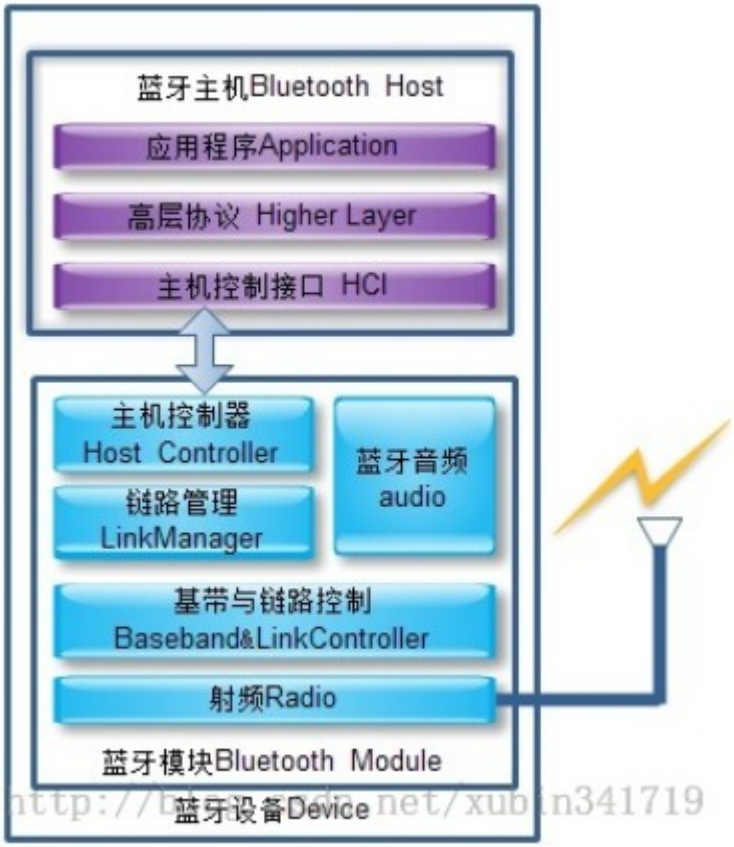
蓝牙宝典

内容目录

蓝牙宝典.....1
一 Bluetooth 的系统构成与主要协议.....2
二 Bluetooth Profile.....5
三 Bluedroid 协议栈.....7
四 bluetooth porting.....8
五 BT enable 过程分析.....10
六 BT 常见问题分析.....20

一 Bluetooth 的系统构成与主要协议

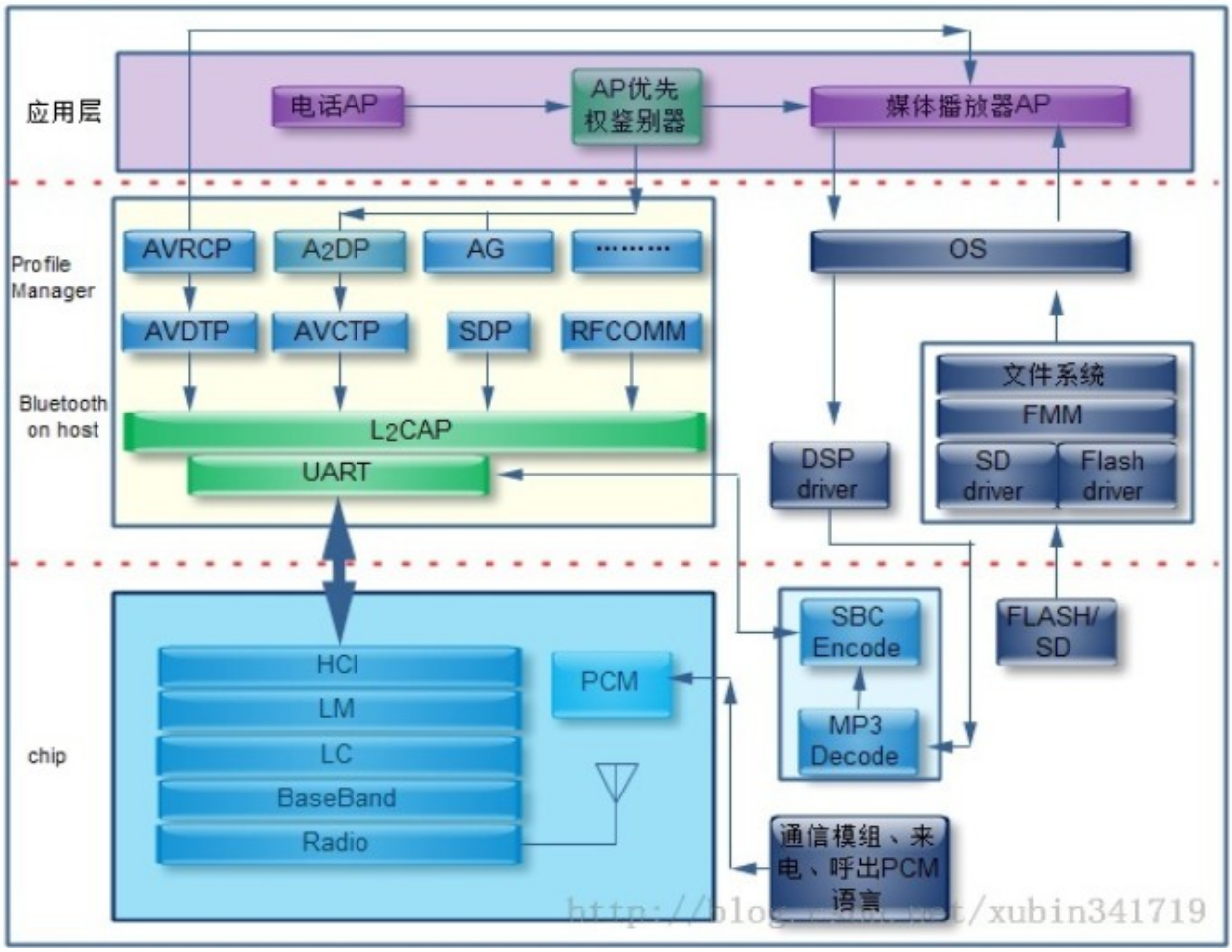


1. 无线射频单元（Radio）:负责数据和语音的发送和接收，特点是短距离，低功耗。 蓝牙天线一般体积小，重量轻，属于微带天线。

2. 基带与链路控制单元（LinkController）：进行射频信号与数字或语音信号的相互转化，实现基带协议和其他的底层连接规程。

3. 链路管理单元（LinkManager）：负责管理蓝牙设备之间的通信，实现链路的建立、验证、链路配置等操作。

4. 蓝牙软件协议实现： 如下图



核心协议：

BR/EDR：Baseband, LMP, L2CAP, SDP

LE: ATT, SMP , L2CAP

上层协议：

RFCOMM， 串口仿真协议， 提供对基于 L2CAP 协议的串口方针

OBEX， 对象交换协议， 实现在不同的设备、不同的平台之间方便、高效的交换信息，基于其实现的 profile 有 MAP，PBAP，OPP 等。

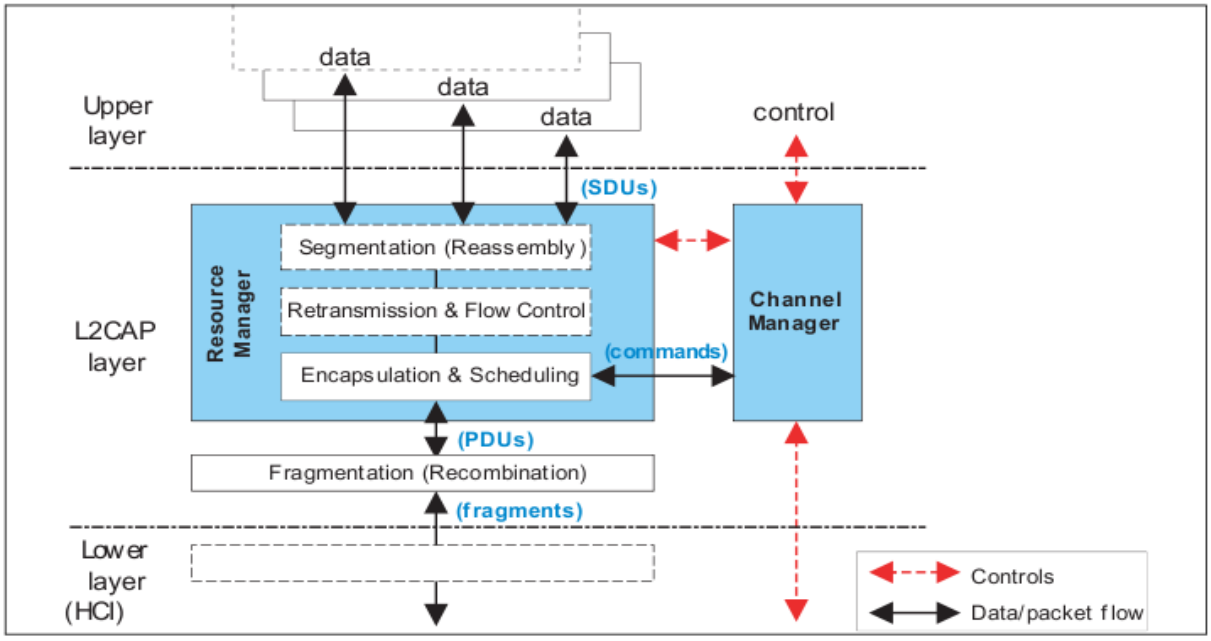
AVDTP ， 音视频分发传输协议， 基于此实现了 AVRCP profile

AVCTP ， 音视频控制传输协议， 基于此实现了 A2DP profile

BNEP， 网络封装协议， PAN 基于此实现

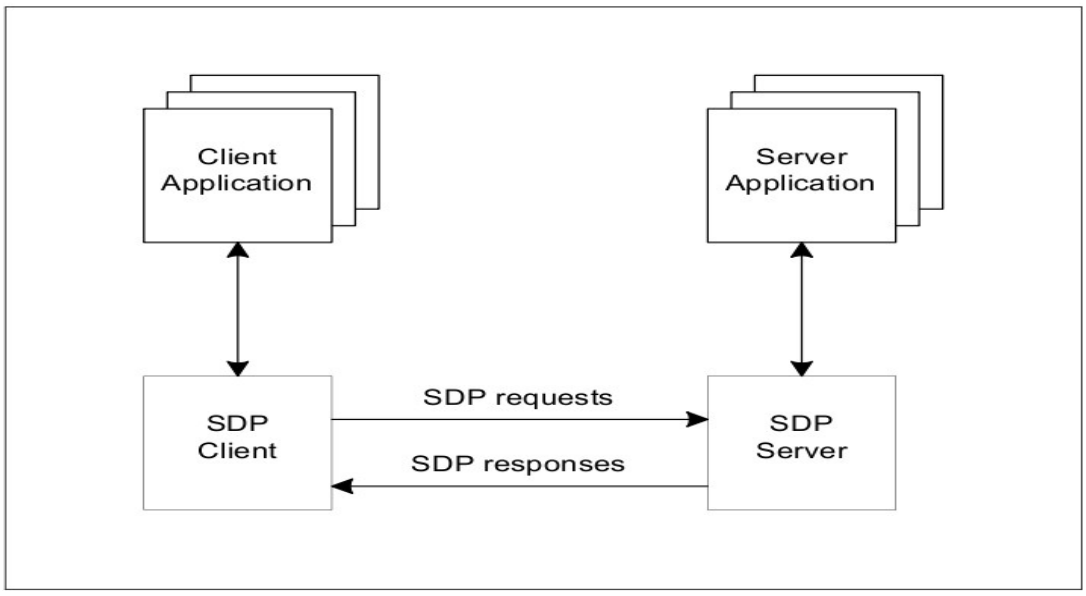
L2CAP Protocol

L2CAP 即 Logik Link Control and Adaptation Protocol, 为上层协议提供多路复用以拆分重组。 控制信道提供对 channel 的管理分配， 数据信道主要将上层传来的 SDU（Service Data Unit）进行处理，然后重组打包以 PDU（Protocol Data Unit）的形式交给逻辑链路层去处理。



SDP Protocol

SDP， Service Discovery Protocol， 发现周围处于可被发现模式的蓝牙设备以及这些蓝牙设备上对应的 service record



ATT Protocol and SMP Protocol

ATT 与 SMP 是 BLE 里面定义的两个 Protocol, 是 BLE 的核心基本协议

ATT Protocol 定义了两个角色，server 和 client, server 端提供系列的 attributes 给 client 端， 这些 attributes 可以被 client 端 discover, read, write； server 端则讲 attributes 指示或通知给 client. Attribute PDU 的形式是 Requests, Responses, Commands, Notifications, Indications, Confirmations, 其中 Requests 与 Responses,Indications, Confirmations 分别互相对应， 采用同步的应答机制。

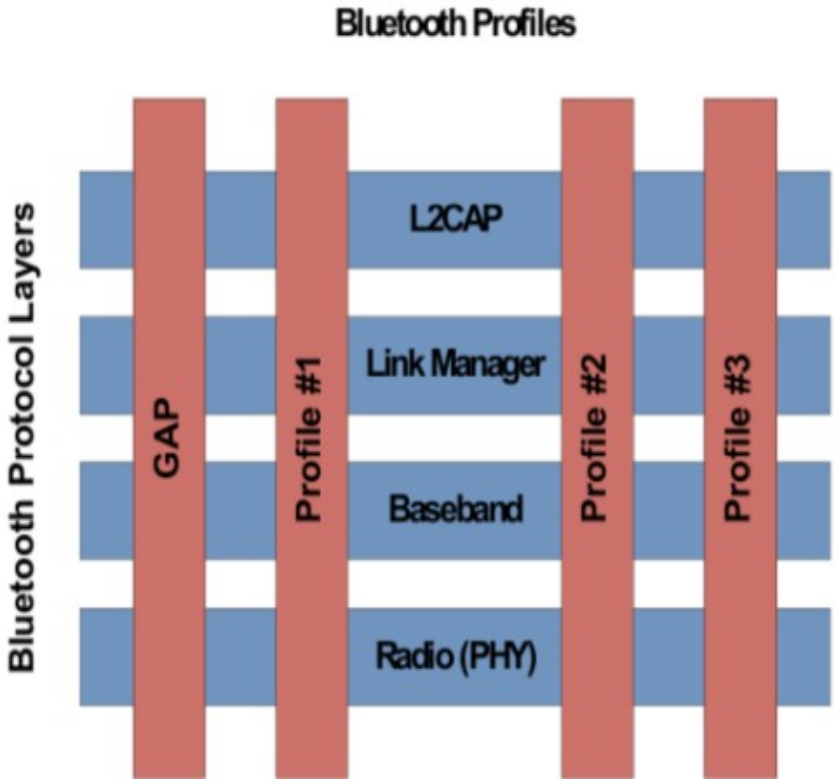
SMP 是 LE 的安全管理协议， 负责 LE 设备配对， 密钥的生成与分配。所有的 SMP 命令都是基于 L2CAP 的固定 channel 来传送的

AMP Protocol

Alternate MAC/PHY 协调 bluetooth 和 wifi 的资源， 在 wifi 空闲的时候来分享 wifi 的资源给 bluetooth 用，从而达到更快的数据吞吐量，在这个过程中，蓝牙模块仅仅是用来创建两台设备之间配对，数据传输本身是通过 WIFI 射频来完成。

二 Bluetooth Profile

What`s profile? Protoco 是一个横向的概念， 而 profile 是一个纵向的、跨越层次的概念



Bluetooth 有很多 profile， 主要有以下几点：

GAP ： Generic Access Profile ， 涵盖蓝牙基本的 profile，定义了蓝牙搜索， 蓝牙链接的链路管理，安全等级等方面的通用流程， bluetooth device 需要实现这个 profile 来保证蓝牙设备之间基本的互操作。 所有的蓝牙设备都必须实现 GAP， 其他上层 profile 都需要基于其来实现。

GATT： Generic Attribute Profile , 对应的 BLE 的基本 profile, LE 的蓝牙设备都遵循 GATT

A2DP： Advanced Audio Distribution Profile , 蓝牙音频传输模型

AVRCP ： Audio/Video Remote Control Profile , 音视频远程控制

HFP， HSP： Handsfree Profile, Headset Profile ， 免提电话

HID ， HOGP： Human Interface Device Profile ， 人机接口设备、蓝牙鼠标、键盘等， 其中 HOGP 是基于 GATT 的 HID。

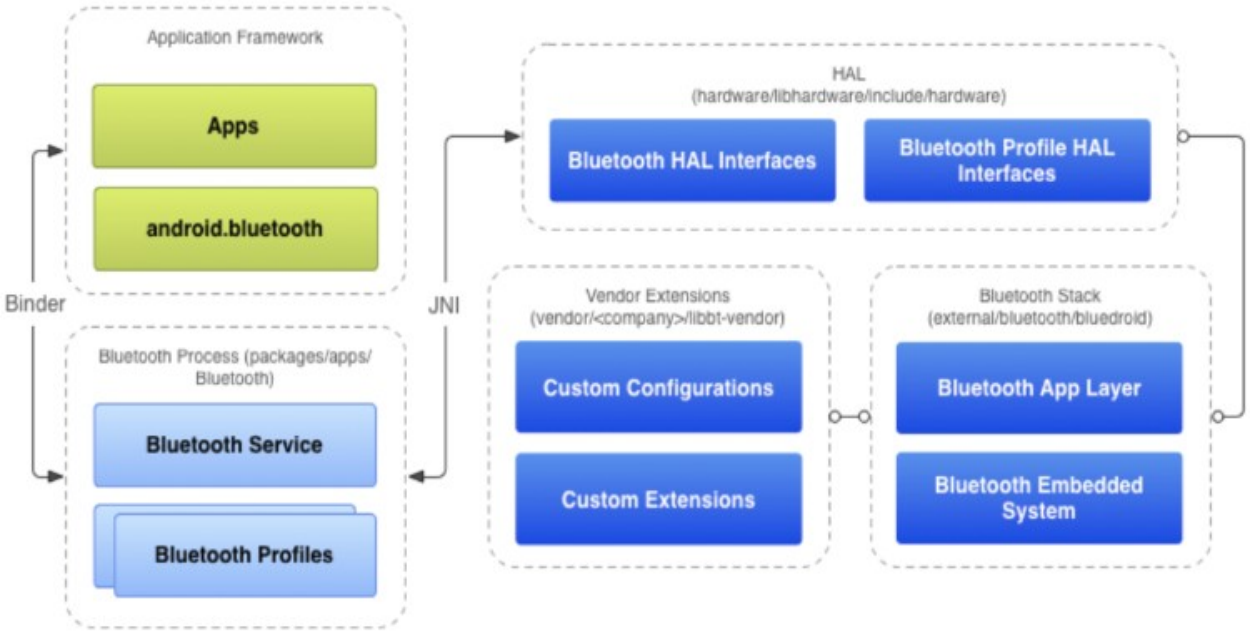
PAN， Personal Area Network Profile , 个人局域网

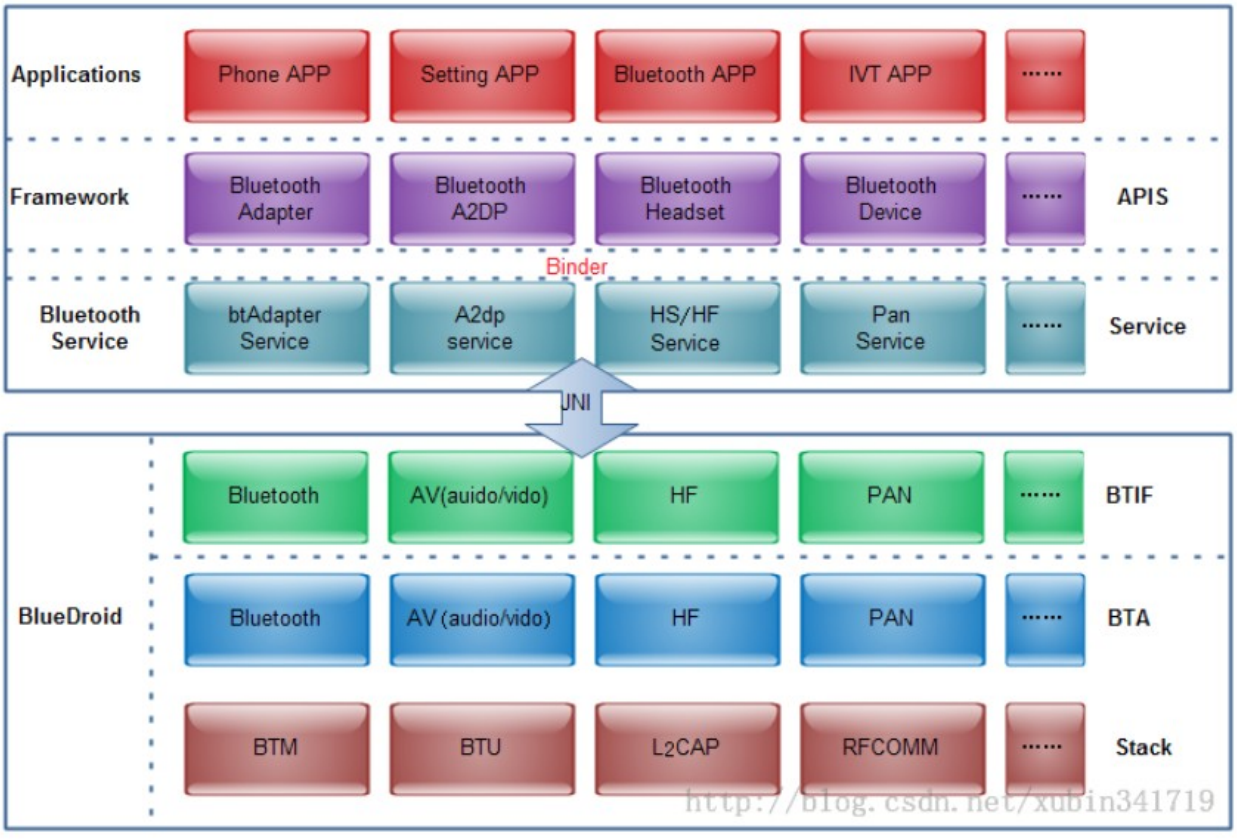
OPP， MAP， PBAP： 分别是 Object Push Profile, Message Access Profile , Phone Book Access Profile , 文件传输，短信，联系人等

SPP， Serial Port Profile， 串口方针， 对应 RFCOMM

HDP， HRP， GLP，BLP ： Health Device Profile, Heart Rate Profile, Glucose Profile, Blood Pressure Profile, 这些都是 LE 相关的， 运动、健康相关的 profile.

三 Bluedroid 协议栈





1. android4.2 之后采用固定的 bluedroid 协议栈 。

（1）、应用程序通过 android.bluetooth package 下的 API 来调用系统的 Bluetooth 功能。

（2）、应用层空间增加了一个名为 Bluetooth 的 App。它做为系统的 bluetooth 核心进程而存在。其内部将通过 JNI 来调用 Bluetooth HAL 层以完成各种蓝牙请求。

（3）、Bluetooth HAL 也属于 Android 4.2 新增模块，它由蓝牙核心规范硬件抽象层和蓝牙应用规范硬件抽象层组成。由于 HAL 层的隔离作用，上层代码可轻松移植到不同芯片平台。
（4）、 作为整个蓝牙服务的核心，Bluetooth Stack 模块则由 Bluetooth Application Layer（缩写为 BTA）和 Bluetooth Embedded System（缩写为 BTE）两大部分组成。BTA 实现了蓝牙设备管理、状态管理及一些应用规范。而 BTE 则通过 HCI 与厂商蓝牙芯片交互以实现了蓝牙协 议栈的通用功能和相关协议。另外，BTE 还包括一个统一内核接口（GKI），蓝牙芯片厂商可借助 GKI 快速轻松得移植蓝牙协议栈到其他操作系统或手机平台 上。
（5）、Vendor Extentions（厂商扩展）：开发者可以添加自定义扩展以实现厂商特定的模块和组件。

2.
Bluedroid 协议细化了 3 个层次，BTIF、BTA、BTE。
BTIF ，即 bluetooth interface ,作为 Bluedroid 与上层 app 交互的接口，上层命令与底层事件回调的双向传递。
BTA ，即 bluetooth application layer, Bluedroid 的应用层，这里的操作实际上是 BTE 来完成的。
BTE， 即 bluetooth embedded system, 这是蓝牙协议栈的真正实现，到这里是 HCI（Host Controller Interface）及以上的部分。

四 bluetooth porting

（1）Kernel 中的 BT driver

I， kernel 中为蓝牙配置驱动的地方，主要涉及 bt enable pin, bt wake host， host wake bt。这些在 bios 里面都会配好。
路径为：
~ /phoenix/kernel/cht/drivers/misc/bcm-lpm

II， 通过
```
static struct acpi_device_id bcm_id_table[] = {
/* ACPI IDs here */
{ "BCM2E1A", 0 },
{ "BCM2E3A", 0 },
{ "OBDA8723", 0},
{ "BCM2E7B", 0 },
{ }
};
```
与 BIOS 中配置好，由 kernel 注册在总线上的 devices 进行匹配。 进而跑我们驱动中的 probe 函数。

III , probe 函数
1.会通过调用
```
bt_lpm_gpiod = gpiod_get_index(&pdev->dev, NULL,
                               gpio_enable_bt_acpi_idx);
bt_lpm.gpio_enable_bt = desc_to_gpio(bt_lpm_gpiod);
```
来从已经注册在总线上的 devices 信息中获取对应的 gpio 号，以便接下来的控制。

2.会通过

```
bt_rfkill = rfkill_alloc("bcm43xx Bluetooth", &pdev->dev,
                RFKILL_TYPE_BLUETOOTH, &bcm43xx_bt_rfkill_ops,
                NULL);
```

来创建一个 rfkill 节点。以后上层通过控制改文件节点来对 bt 进行上下电的管理。

（2）Device 目录

I，在 device 目录中，device.mk 会通过编译将蓝牙所需要的库文件与配置文件放在所需位置。

```
PRODUCT_PACKAGES += \
                bd_prov \
                libbt-vendor \
                audio.a2dp.default \
                rfkill_bt.sh \
                bt_vendor.default.conf \
                BCM4354A2_001.003.015.0053.0147_Wci2In_Gpio6_RC.hcd \
                RVP_BCM4354A2_001.003.015.0053.0148_Wci2In_Gpio4_RC.hcd \
PRODUCT_COPY_FILES +=
frameworks/native/data/etc/android.hardware.bluetooth.xml:system/etc/permissions/android.hardware.bluetooth.xml \
```

II， init.rc 中会创建一些蓝牙所需要的文件目录，并且定义一些上层所需调用的 service.

```
on boot
 # change back to bluetooth from system
  chown bluetooth net_bt_stack /data/misc/Bluetooth
  mkdir /data/misc/bluedroid 0770 bluetooth net_bt_stack
 # UART device
  chmod 0660 /dev/ttyHSU0
  chown bluetooth net_bt_stack /dev/ttyHSU0
 # power up/down interface
  chmod 0660 /sys/class/rfkill/rfkill2/state
  chmod 0660 /sys/class/rfkill/rfkill2/type
  write /sys/class/rfkill/rfkill2/state 0
  chown bluetooth net_bt_stack /sys/class/rfkill/rfkill2/state
  chown bluetooth net_bt_stack /sys/class/rfkill/rfkill2/type
 # bluetooth MAC address programming
  chown bluetooth net_bt_stack ro.bt.bdaddr_path
  chown bluetooth net_bt_stack /system/etc/bluetooth
  chown bluetooth net_bt_stack /data/misc/bluetooth
  setprop ro.bt.bdaddr_path "/data/misc/bluetooth/bdaddr"
```

五 BT enable 过程分析

1. 从 UI 操作开始， 在设置界面中打开蓝牙设备， 这部分代码在 packages/apps/Settings/src/com/android/settings/bluetooth/BluetoothEnabler.java

```
176     @Override
177     public void onSwitchChanged(Switch switchView, boolean isChecked) {
178         // Show toast message if Bluetooth is not allowed in airplane mode
179         if (isChecked &&
180                 !WirelessSettings.isRadioAllowed(mContext, Settings.Global.RADIO_BLUETOOTH)) {
181             Toast.makeText(mContext, R.string.wifi_in_airplane_mode, Toast.LENGTH_SHORT).show();
182             // Reset switch to off
183             switchView.setChecked(false);
184         }
185
186         if (mLocalAdapter != null) {
187             mLocalAdapter.setBluetoothEnabled(isChecked);
188         }
189         mSwitch.setEnabled(false);
190     }
191 }
```

179---184：介绍的是关于飞行模式下蓝牙的状态。

我们重点关注 187 mLocalAdapter.setBluetoothEnabled(isChecked);这个是蓝牙打开与关闭的一个接口。 mlocalAdapter 类的定义为 private final LocalBluetoothAdapter mLocalAdapter;这样我们就调到了 LocalBluetoothAdapter 类中。

2.在 packages/apps/Settings/src/com/android/settings/bluetooth/LocalBluetoothAdapter.java 文件中

```
198     public void setBluetoothEnabled(boolean enabled) {
199         boolean success = enabled
200                 ? mAdapter.enable()
201                 : mAdapter.disable();
202
203         if (success) {
204             setBluetoothStateInt(enabled
205                 ? BluetoothAdapter.STATE_TURNING_ON
206                 : BluetoothAdapter.STATE_TURNING_OFF);
207         } else {
208             if (Utils.V) {
209                 Log.v(TAG, "setBluetoothEnabled call, manager didn't return " +
210                         "success for enabled: " + enabled);
211             }
212
213             syncBluetoothState();
214         }
215     }
```

199---201 boolean success = enabled ? mAdapter.enable() : mAdapter.disable(); 这里有个 mAdapter，在代码中可以找见 private final BluetoothAdapter mAdapter;查找发现在 frameworks/base/core/java/android/bluetooth/BluetoothAdapter.java 下。（由 Settings app 跳到 Frameworks api 中）

3. 进入 BluetoothAdapter.java 中的 public final class BluetoothAdapter 类中。找见

```
568     public boolean enable() {
569         if (isEnabled() == true){
570             if (DBG) Log.d(TAG, "enable(): BT is already enabled..!");
571             return true;
572         }
573         try {
574             return mManagerService.enable();
575         } catch (RemoteException e) {Log.e(TAG, "", e);}
576         return false;
577     }
```

重点关注 574 行 mManagerService.enable(); mManagerService 对象是由 IBluetoothManager 接口实现的。

```
379     private final IBluetoothManager mManagerService;
```

接下来寻找到./services/core/java/com/android/server/BluetoothManagerService.java

4.在./services/core/java/com/android/server/BluetoothManagerService.java 中

```
445     public boolean enable() {
446         if ((Binder.getCallingUid() != Process.SYSTEM_UID) &&
447             (!checkIfCallerIsForegroundUser())) {
448             Log.w(TAG,"enable(): not allowed for non-active and non system user");
449             return false;
450         }
451
452         mContext.enforceCallingOrSelfPermission(BLUETOOTH_ADMIN_PERM,
453                                                 "Need BLUETOOTH ADMIN permission");
454         if (DBG) {
455             Log.d(TAG,"enable():  mBluetooth =" + mBluetooth +
456                     " mBinding = " + mBinding);
457         }
458
459         synchronized(mReceiver) {
460             mQuietEnableExternal = false;
461             mEnableExternal = true;
462             // waive WRITE_SECURE_SETTINGS permission check
463             long callingIdentity = Binder.clearCallingIdentity();
464             persistBluetoothSetting(BLUETOOTH_ON_BLUETOOTH);
465             Binder.restoreCallingIdentity(callingIdentity);
466             sendEnableMsg(false);
467         }
468         return true;
469     }
```

我们找见 enable 函数。 大致看一下，首先判断蓝牙是不是由系统 app 开启的，检查是否有操作蓝牙的权限， 在这我们底层会有两个.xml 权限文件是要加载的。 关键代码在 466 sendEnableMsg(false);

```
1478    private void sendEnableMsg(boolean quietMode) {
1479        mHandler.sendMessage(mHandler.obtainMessage(MESSAGE_ENABLE,
1480                                    quietMode ? 1 : 0, 0));
1481    }
```

在本文件的 1478 行能够找见函数， 用了 Handler 消息机制。其中 arg1=0,arg2=0.直接找 mHandler 这个对象里面的 handleMessage 去看 switch----case 即可。

```
970                 case MESSAGE_ENABLE:
971                     if (DBG) {
972                         Log.d(TAG, "MESSAGE_ENABLE: mBluetooth = " + mBluetooth);
973                     }
974                     mHandler.removeMessages(MESSAGE_RESTART_BLUETOOTH_SERVICE);
975                     mEnable = true;
976                     handleEnable(msg.arg1 == 1);
977                     break;
```

关注在 976 行 handleEnable，这里传的参数是 0。

```
1319                //Enable bluetooth
1320                try {
1321                    if (!mQuietEnable) {
1322                        if(!mBluetooth.enable()) {
1323                            Log.e(TAG,"IBluetooth.enable() returned false");
1324                        }
1325                    }
1326                    else {
1327                        if(!mBluetooth.enableNoAutoConnect()) {
1328                            Log.e(TAG,"IBluetooth.enableNoAutoConnect() returned false");
1329                        }
1330                    }
1331                } catch (RemoteException e) {
1332                    Log.e(TAG,"Unable to call enable()",e);
1333                }
```

其中 mQuietEnable 为我们传进来的参数 0,取反为 1,所以我们就调用了 mBluetooth.enable(). private IBluetooth mBluetooth; mBluetooth 使用了 Ibluetooth 接口，这个 mBluetooth 就是 Bluetooth.apk 里面的 AdapterService 了。去看看代码吧。

5. 打开 packages/apps/Bluetooth/src/com/android/bluetooth/btservice/AdapterService.java（又回到了 packages app 目录中）

```
651         public boolean enable() {
652             if ((Binder.getCallingUid() != Process.SYSTEM_UID) &&
653                 (!Utils.checkCaller())) {
654                 Log.w(TAG, "enable() - Not allowed for non-active user and non system user");
655                 return false;
656             }
657
658             AdapterService service = getService();
659             if (service == null) return false;
660             return service.enable();
661         }
```

据孙超同学所说， service.enable();就是 AdapterSerivice.java 文件中的 enable 函数。

```
1173        boolean enable() {
1174            return enable (false);
1175        }
```

```
1181      public synchronized boolean enable(boolean quietMode) {
1182          enforceCallingOrSelfPermission(BLUETOOTH_ADMIN_PERM,
1183                                         "Need BLUETOOTH ADMIN permission");
1184          debugLog("enable() - Enable called with quiet mode status =  " + mQuietmode);
1185          mQuietmode  = quietMode;
1186          Message m =
1187                  mAdapterStateMachine.obtainMessage(AdapterState.USER_TURN_ON);
1188          mAdapterStateMachine.sendMessage(m);
1189          return true;
1190      }
```

从 1187 行可以看出，用到了状态机， 传了一个 USER_TURN_ON 状态给状态机。

```
131                 case USER_TURN_ON:
132                     if (DBG) Log.d(TAG,"CURRENT_STATE=OFF, MESSAGE = USER_TURN_ON");
133                     notifyAdapterStateChange(BluetoothAdapter.STATE_TURNING_ON);
134                     mPendingCommandState.setTurningOn(true);
135                     transitionTo(mPendingCommandState);
136                     sendMessageDelayed(START_TIMEOUT, START_TIMEOUT_DELAY);
137                     adapterService.processStart();
138                     break;
```

133 ：更新 AdapterService 里的状态为 STATE_TURNING_ON

134 ： 设置 pending 状态下的 Turning on 为打开状态

135 ： 转移到 pending 状态

136 ： 设置超时

137 ： 真正 Enable 蓝牙的地方

再来看看 processStart();

```
398         //Start profile services
399         if (!mProfilesStarted && supportedProfileServices.length >0) {
400             //Startup all profile services
401             setProfileServiceState(supportedProfileServices,BluetoothAdapter.STATE_ON);
402         }else {
403             debugLog("processStart() - Profile Services alreay started");
404             mAdapterStateMachine.sendMessage(mAdapterStateMachine.obtainMessage(AdapterState.STARTED));
405         }
406     }
```

这两个 if 分支最终都会将状态机从 pending 状态变为 STARTED 状态。

```
257             case STARTED:  {
258                 if (DBG) Log.d(TAG,"CURRENT_STATE=PENDING, MESSAGE = STARTED, isTurningOn=" + isTurningOn + ", isTurningOff=" + isTurningOff);
259                 //Remove start timeout
260                 removeMessages(START_TIMEOUT);
261
262                 //Enable
263                 boolean ret = adapterService.enableNative();
264                 if (!ret) {
265                     Log.e(TAG, "Error while turning Bluetooth On");
266                     notifyAdapterStateChange(BluetoothAdapter.STATE_OFF);
267                     transitionTo(mOffState);
268                 } else {
269                     String timeout = SystemProperties.get("bt.enable.timeout.delay");
270                     if (!timeout.isEmpty())
271                         enable_timeout_delay = Integer.parseInt(timeout);
272                     else
273                         enable_timeout_delay = DEFAULT_ENABLE_TIMEOUT_DELAY;
274                     Log.d(TAG, "enable_timeout_delay = " + enable_timeout_delay + " ms");
275                     sendMessageDelayed(ENABLE_TIMEOUT, enable_timeout_delay);
276                 }
277             }
```

263 行 boolean ret = mAdapterService.enableNative();才是真的.....

我们终于从 java 层进入了 Native 层了。

6. 打开文件

packages/apps/Bluetooth/jni/com_android_bluetooth_btservice_AdapterService.cpp，找到 enableNative 函数。

```
794 static jboolean enableNative(JNIEnv* env, jobject obj) {
795     ALOGV("%s:",__FUNCTION__);
796
797     jboolean result = JNI_FALSE;
798     if (!sBluetoothInterface) return result;
799
800     int ret = sBluetoothInterface->enable();
801     result = (ret == BT_STATUS_SUCCESS) ? JNI_TRUE : JNI_FALSE;
802     return result;
803 }
804
```

调用了 sBluetoothInterface 里的 enable().

```
 51 static const bt_interface_t *sBluetoothInterface = NULL;
```

是 bt_interface_t 类型的结构体。

这就到了 bluedroid 协议中。

7.external/bluetooth/bluedroid/btif/src/bluetooth.c 中

```
433 static const bt_interface_t bluetoothInterface = {
434     sizeof(bluetoothInterface),
435     init,
436     enable,
437     disable,
438     cleanup,
439     get_adapter_properties,
440     get_adapter_property,
441     set_adapter_property,
442     get_remote_device_properties,
443     get_remote_device_property,
444     set_remote_device_property,
445     get_remote_service_record,
446     get_remote_services,
447     start_discovery,
448     cancel_discovery
```

调用这里面的 enable 函数

```
145 static int enable( void )
146 {
147     ALOGI("enable");
148
149     /* sanity check */
150     if (interface_ready() == FALSE)
151         return BT_STATUS_NOT_READY;
152
153     return btif_enable_bluetooth();
154 }
155
```

接下来调用 btif_enable_bluetooth();函数

```
528 bt_status_t btif_enable_bluetooth(void)
529 {
530     BTIF_TRACE_DEBUG("BTIF ENABLE BLUETOOTH");
531
532     if (btif_core_state != BTIF_CORE_STATE_DISABLED)
533     {
534         usleep(10000); /* 10 milliseconds */
535         kill(getpid(), SIGKILL);
536         ALOGD("not disabled\n");
537         return BT_STATUS_DONE;
538     }
539
540     btif_core_state = BTIF_CORE_STATE_ENABLING;
541
542     /* Create the GKI tasks and run them */
543     bte_main_enable();
544
545     return BT_STATUS_SUCCESS;
546 }
547
```

接下来是 bte_main_enable();

```
207 void bte_main_enable()
208 {
209     APPL_TRACE_DEBUG("%s", __FUNCTION__);
210
211     /* Initialize BTE control block */
212     BTE_Init();
213
214     lpm_enabled = FALSE;
215
216     GKI_create_task((TASKPTR)btu_task, BTU_TASK, BTE_BTU_TASK_STR,
217                     (UINT16 *) ((UINT8 *)bte_btu_stack + BTE_BTU_STACK_SIZE),
218                     sizeof(bte_btu_stack));
219
220     bte_hci_enable();
221
222     GKI_run();
223 }
224
```

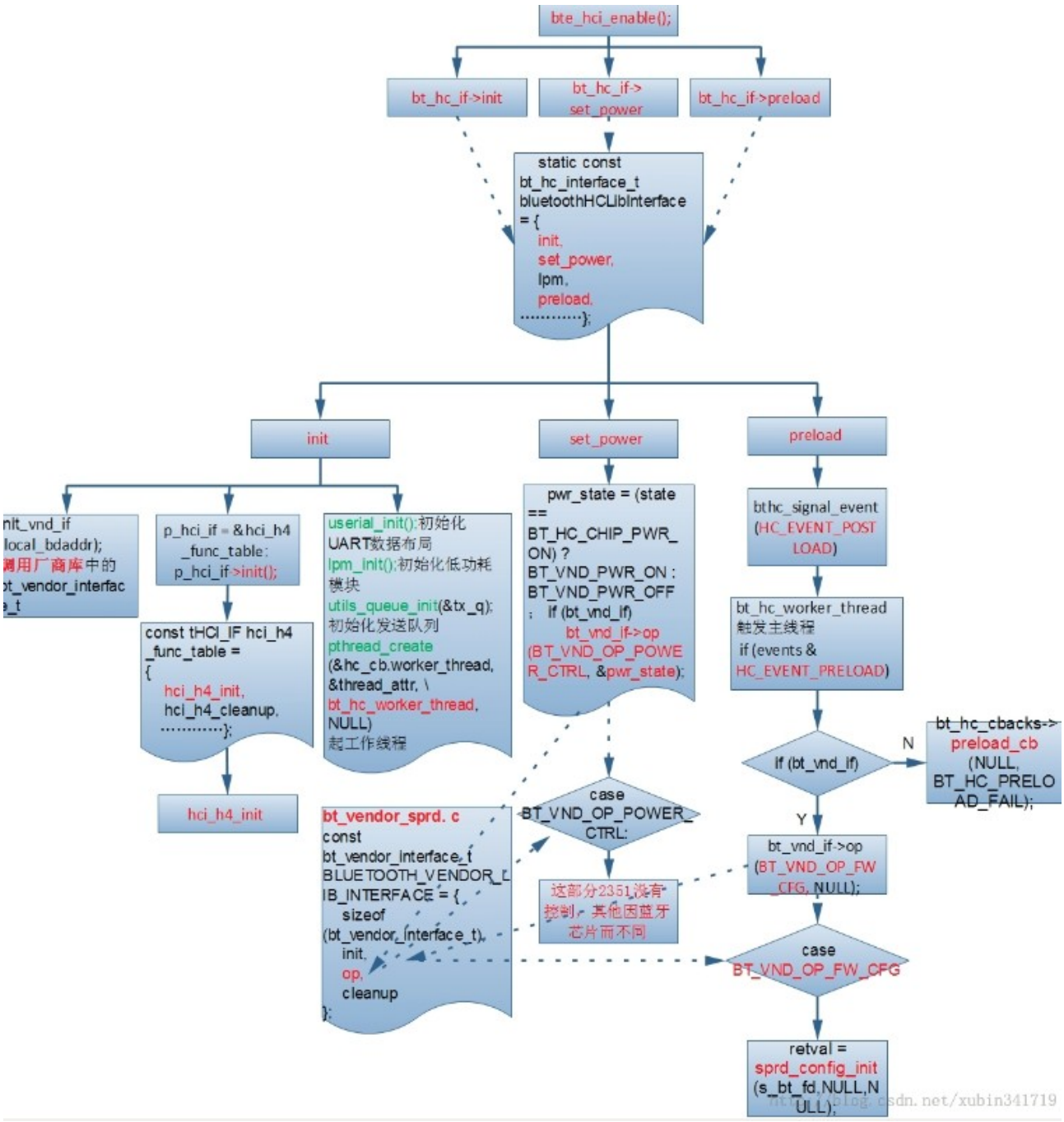
这个函数里面 （1）初始化了 BTE 控制模块；

（2） 创建了 BTU_TASK 进程；

(3) 打开 HCI 和厂商模块;

这 3 个步骤每个都能引申出一大串的东西。 我们这里只关注第三个步骤

8. 我们看 bte_hci_enable 函数。



上图为 bte_hci_enable 函数后的调用关系，主要做了各种东西的初始化工作。

而我们现在只关注其中的 setpower

```
316         bt_hc_if->set_power(BT_HC_CHIP_PWR_ON);
317
```

```
91 static bt_hc_interface_t *bt_hc_if=NULL;
```

我们看到 bt_hc_if 结构体的格式为 bt_hc_interface_t

其中的 set_power 为 external/bluetooth/bluedroid/hci/src/bt_hci_bdroid.c 中的 set_power

```
379 static void set_power(bt_hc_chip_power_state_t state)
380 {
381     int pwr_state;
382
383     BTHCDBG("set_power %d", state);
384
385     /* Calling vendor-specific part */
386     pwr_state = (state == BT_HC_CHIP_PWR_ON) ? BT_VND_PWR_ON : BT_VND_PWR_OFF;
387
388     vendor_send_command(BT_VND_OP_POWER_CTRL, &pwr_state);
389 }
```

再来看 vendor_send_command 函数的实现

```
104 int vendor_send_command(bt_vendor_opcode_t opcode, void *param) {
105   assert(vendor_interface != NULL);
106
107   return vendor_interface->op(opcode, param);
108 }
```

而 vendor_interface 是 bt_vendor_interface_t 类型的结构体指针。

用 vendor_send_command 函数将状态传给了操作函数，这样我们就调到了 vendor 提供的库中。

9. hardware/broadcom/libbt/src/bt_vendor_brcm.c 中

```
138 static int op(bt_vendor_opcode_t opcode, void *param)
139 {
140     int retval = 0;
141
142     BTVNDDBG("op for %d", opcode);
143
144     switch(opcode)
145     {
146         case BT_VND_OP_POWER_CTRL:
147             {
148                 int *state = (int *) param;
149                 if (*state == BT_VND_PWR_OFF)
150                     upio_set_bluetooth_power(UPIO_BT_POWER_OFF);
151                 else if (*state == BT_VND_PWR_ON)
152                     upio_set_bluetooth_power(UPIO_BT_POWER_ON);
153             }
154             break;
```

接下来是 upio_set_bluetooth_power 函数了

```
281     /* check if we have rfkill interface */
282     if (is_rfkill_disabled())
283         return 0;
284
285     if (rfkill_id == -1)
286     {
287         if (init_rfkill())
288             return ret;
289     }
290
291     fd = open(rfkill_state_path, O_WRONLY);
292
293     if (fd < 0)
294     {
295         ALOGE("set_bluetooth_power : open(%s) for write failed: %s (%d)",
296             rfkill_state_path, strerror(errno), errno);
297         return ret;
298     }
299
300     sz = write(fd, &buffer, 1);
301
302     if (sz < 0) {
303         ALOGE("set_bluetooth_power : write(%s) failed: %s (%d)",
304             rfkill_state_path, strerror(errno),errno);
305     }
306     else
307         ret = 0;
308
309     if (fd >= 0)
310         close(fd);
311
312     return ret;
313 }
```

这个函数就会通过我们 kernel 创建的 rfkill 节点来对 bt 芯片进行上电了， 接下来就到了 kernel 中。

10.在 BT 的驱动 kernel/cht/drivers/misc/bcm-lpm/bcm_bt_lpm.c 中

```
216 static const struct rfkill_ops bcm43xx_bt_rfkill_ops = {
217         .set_block = bcm43xx_bt_rfkill_set_power,
218 };
```

通过上面操作函数调用到

```
169 static int bcm43xx_bt_rfkill_set_power(void *data, bool blocked)
170 {
171         /* rfkill_ops callback. Turn transmitter on when blocked is false */
172
173         if (!blocked) {
174 #ifdef LPM_ON
175                 gpio_set_value(bt_lpm.gpio_wake, 1);
176                 /*
177                 * Delay advice by BRCM is min 2.5ns,
178                 * setting it between 10 and 50us for more confort
179                 */
180                 usleep_range(10, 50);
181 #endif
182 #ifdef CONFIG_PF450CL
183                 gpio_set_value(bt_lpm.gpio_reg_on, 1);
184                 gpio_set_value(bt_lpm.gpio_reset, 1);
185 #else
186                 gpio_set_value(bt_lpm.gpio_enable_bt, 1);
187 #endif
188                 if((bt_lpm.tty_dev!=NULL) && (uart_enabled==false))
189                 {
190                         pm_runtime_get(bt_lpm.tty_dev);
191                         uart_enabled=true;
192                         printk("TSOS %s: get tty\n", __func__);
193                 }
194                 printk("TSOS %s: turn BT on\n", __func__);
195         } else {
196 #ifdef CONFIG_PF450CL
197                 gpio_set_value(bt_lpm.gpio_reg_on, 0);
198                 gpio_set_value(bt_lpm.gpio_reset, 0);
199 #else
200                 gpio_set_value(bt_lpm.gpio_enable_bt, 0);
201 #endif
202                 if((bt_lpm.tty_dev!=NULL) && (uart_enabled==true))
203                 {
204                         uart_enabled=false;
205                         pm_runtime_put(bt_lpm.tty_dev);
206                         printk("TSOS %s: put tty\n", __func__);
207                 }
208                 printk("TSOS %s: turn BT off\n", __func__);
209         }
210
211         bt_enabled = !blocked;
212
```

再通过 gpio_set_value 来给 enable 脚上电。

11.这样蓝牙就上电了。

六 BT 常见问题分析

1. Camellia 项目遇见了 BT 有时无法正常打开的 BUG，BT 开关再打开到一半总是会弹回。

解决过程：

（1）通过波形我们发现每次发送 BT 的 conf 包的时候， 数据都会变成 0. 调查发现， BT 上层发包到 kernel 中， uart 通过 dma 的方式进行传输的， 这是指一种高速的数据传输操作，允许在外部设备和存储器之间直接读写数据，既不通过 CPU，也不需要 CPU 干预。dma 内存一般在内存的 0--4G，而我 们板子的内存是 8G 的。通过研究 uart 驱动， 发现在分配内存页的时候并没有申明 dma 内存。所以我们修改为 dma 内存， 观察波形，发现 conf 包能够正确发送了。

（2）解决完上面问题， BT 还是会出现打不开的情况。 观察波形， 发现 host 端 rts 脚被过早的拉高了，因为 rts 是低电平有效， 所以造成 firmware 下载没完成便被中断了。所以在 libbt 里面向底层发送命令的函数加一个 0.2s 的延迟，再观察波形，发现正常了，BT 也能正常 打开了。

就这样，BT 的这个 BUG 就解决了。