bluetoothInterface

作用：结构体中的成员大部分都是函数指针，主要完成函数的挂接！

```
static const bt_interface_t bluetoothInterface = {
    sizeof(bluetoothInterface),
    init,
    initq,
    enable,
    disable,
    cleanup,
    ssrcleanup,
    get_adapter_properties,
    get_adapter_property,
    set_adapter_property,
    get_remote_device_properties,
    get_remote_device_property,
    set_remote_device_property,
    get_remote_service_record,
    get_remote_services,
    start_discovery,
    cancel_discovery,
    create_bond,
    remove_bond,
    cancel_bond,
    get_connection_state,
    pin_reply,
    ssp_reply,
    get_profile_interface,
    dut_mode_configure,
    dut_mode_send,
#if HCI_RAW_CMD_INCLUDED == TRUE
    hci_cmd_send,
#else
    NULL,
#endif
#if BLE_INCLUDED == TRUE
    le_test_mode,
#else
    NULL,
#endif
    config_hci_snoop_log,
    set_os_callouts,
    read_energy_info,
#if TEST_APP_INTERFACE == TRUE
    get_testapp_interface,
#else
    NULL,
#endif
    bt_le_lpp_write_rssi_threshold,
    bt_le_lpp_enable_rssi_monitor,
    bt_le_lpp_read_rssi_threshold,
};
```

init

作用：

```
static int init(bt_callbacks_t* callbacks )
{
    ALOGI("init");

    /* sanity check */
    if (interface_ready() == TRUE)
        return BT_STATUS_DONE;

    /* store reference to user callbacks */
    bt_hal_cbacks = callbacks;

    /* add checks for individual callbacks ? */

    bt_utils_init();

    /* init btif */
    btif_init_bluetooth();
```

```
    return BT_STATUS_SUCCESS;
}
```

interface_ready()

作用：检查接口函数是否准备好

```
static uint8_t interface_ready(void)
{
    /* add checks here that would prevent API calls other than init to be executed */
    if (bt_hal_cbacks == NULL)
        return FALSE;

    return TRUE;
}
```

bt_utils_init()

作用：工具集初始化，初始化一些互斥锁。

```
void bt_utils_init() {
    int i;
    pthread_mutexattr_t lock_attr;

    for(i = 0; i < TASK_HIGH_MAX; i++) {
        g_DoSchedulingGroupOnce[i] = PTHREAD_ONCE_INIT;
        g_DoSchedulingGroup[i] = TRUE;
        g_TaskIDs[i] = INVALID_TASK_ID;
    }
    pthread_mutexattr_init(&lock_attr);
    pthread_mutex_init(&gIdxLock, &lock_attr);
}
```

btif_init_bluetooth()

作用：创建btif任务，准备蓝牙，开启相关调度程序。初始化蓝牙接口blueinterface

```
bt_status_t btif_init_bluetooth()
{
    UINT8 status;
    btif_config_init();
    bte_main_boot_entry();

    /* As part of the init, fetch the local BD ADDR */
    memset(&btif_local_bd_addr, 0, sizeof(bt_bdaddr_t));
    btif_fetch_local_bdaddr(&btif_local_bd_addr);

    /* start btif task */
    status = GKI_create_task(btif_task, BTIF_TASK, BTIF_TASK_STR,
                (UINT16 *) ((UINT8 *)btif_task_stack + BTIF_TASK_STACK_SIZE),
                sizeof(btif_task_stack));

    if (status != GKI_SUCCESS)
        return BT_STATUS_FAIL;
    btif_core_state = BTIF_CORE_STATE_INITIALIZED;
    return BT_STATUS_SUCCESS;
}
```

btif_config_init

作用：

```
int btif_config_init()
{
    static int initialized;
    bdld("in initialized:%d", initialized);
    if(!initialized)
    {
        initialized = 1;
        struct stat st;
        if(stat(CFG_PATH, &st) != 0)
            bdle("%s does not exist, need provision", CFG_PATH);
        btsock_thread_init();
        init_slot_lock(&slot_lock);
        lock_slot(&slot_lock);
        root.name = "Bluedroid";
        alloc_node(&root, CFG_GROW_SIZE);
        dump_node("root", &root);
```

```
        pth = btsock_thread_create(NULL, cfg_cmd_callback);
        load_cfg();
        unlock_slot(&slot_lock);
        #ifdef UNIT_TEST
            cfg_test_write();
            //cfg_test_read();
            exit(0);
        #endif
    }
    return pth >= 0;
}
```

bte_main_boot_entry

作用：

```
void bte_main_boot_entry(void)
{
    /* initialize OS */
    GKI_init();                 //GKI初始化

    bte_main_in_hw_init();      //初始化结构体：通过 bt_hc_get_interface判断API实例参数是否为空，如果不为空就将控制块结构体清空为0 function for chip hardware init

    bte_load_conf(BTE_STACK_CONF_FILE);   //Reads the stack configuration file and populates global variables with the contents of the file.
#if (defined(BLE_INCLUDED) && (BLE_INCLUDED == TRUE))
    bte_load_ble_conf(BTE_BLE_STACK_CONF_FILE);
#endif

#if (BTTRC_INCLUDED == TRUE)
    /* Initialize trace feature */
    BTTRC_TraceInit(MAX_TRACE_RAM_SIZE, &BTE_TraceLogBuf[0], BTTRC_METHOD_RAM);
#endif

    pthread_mutex_init(&cleanup_lock, NULL);

}
```

GKI_init()

作用：初始化一些信号量和互斥锁

```
/*******************************************************************************
** Function         GKI_init
** Description      This function is called once at startup to initialize
**                  all the timer structures.
** Returns          void
*******************************************************************************/

void GKI_init(void)
{
    pthread_mutexattr_t attr;
    tGKI_OS             *p_os;

    memset (&gki_cb, 0, sizeof (gki_cb));

    gki_buffer_init();      //initialize all buffers and free buffer pools
    gki_timers_init();      //initialize all the timer structures.
    alarm_service_init();

    gki_cb.com.OSTicks = (UINT32) times(0);

    pthread_mutexattr_init(&attr);

#ifndef __CYGWIN__
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE_NP);
#endif
    p_os = &gki_cb.os;
    pthread_mutex_init(&p_os->GKI_mutex, &attr);
    pthread_mutex_init(&p_os->gki_timerupdate_mutex, &attr);
    /* pthread_mutex_init(&GKI_sched_mutex, NULL); */
#if (GKI_DEBUG == TRUE)
    pthread_mutex_init(&p_os->GKI_trace_mutex, NULL);
#endif
    /* pthread_mutex_init(&thread_delay_mutex, NULL); */  /* used in GKI_delay */
```

```
    /* pthread_cond_init (&thread_delay_cond, NULL); */

    struct sigevent sigevent;
    memset(&sigevent, 0, sizeof(sigevent));
    sigevent.sigev_notify = SIGEV_THREAD;
    sigevent.sigev_notify_function = (void (*)(union sigval))bt_alarm_cb;
    sigevent.sigev_value.sival_ptr = NULL;
    if (timer_create(CLOCK_REALTIME, &sigevent, &posix_timer) == -1) {
        ALOGE("%s unable to create POSIX timer: %s", __func__, strerror(errno));
        timer_created = false;
    } else {
        timer_created = true;
    }
}
```

bte_main_in_hw_init()

作用：Internal helper function for chip hardware init

```
static void bte_main_in_hw_init(void)
{
    if ( (bt_hc_if = (bt_hc_interface_t *) bt_hc_get_interface()) == NULL)
    {
        APPL_TRACE_ERROR("!!! Failed to get BtHostControllerInterface !!!");
    }

    memset(&preload_retry_cb, 0, sizeof(bt_preload_retry_cb_t));
}
```

bte_load_conf

作用：

```
// Reads the stack configuration file and populates global variables with
// the contents of the file.
void bte_load_conf(const char *path) {
  assert(path != NULL);

  ALOGI("%s attempt to load stack conf from %s", __func__, path);

  config_t *config = config_new(path);
  if (!config) {
    ALOGI("%s file >%s< not found", __func__, path);
    return;
  }

  strlcpy(hci_logfile, config_get_string(config, CONFIG_DEFAULT_SECTION, "BtSnoopFileName", ""), sizeof
(hci_logfile));
  hci_logging_enabled = config_get_bool(config, CONFIG_DEFAULT_SECTION, "BtSnoopLogOutput", false);
  hci_ext_dump_enabled = config_get_bool(config, CONFIG_DEFAULT_SECTION, "BtSnoopExtDump", false);
  hci_save_log = config_get_bool(config, CONFIG_DEFAULT_SECTION, "BtSnoopSaveLog", false);
  trace_conf_enabled = config_get_bool(config, CONFIG_DEFAULT_SECTION, "TraceConf", false);

  bte_trace_conf_config(config);
            config_free(config);
          }
```

cleanup

作用：

```
static void cleanup( void )
{
    /* sanity check */
    if (interface_ready() == FALSE)
        return;

    btif_shutdown_bluetooth();

    /* hal callbacks reset upon shutdown complete callback */

    return;
}
```

disable

作用：

```
static int disable(void)
```

```
{
    /* sanity check */
    if (interface_ready() == FALSE)
        return BT_STATUS_NOT_READY;

    return btif_disable_bluetooth();
}
```

enable

作用：

```
static int enable( void )
{
    ALOGI("enable");

    /* sanity check */
    if (interface_ready() == FALSE)
        return BT_STATUS_NOT_READY;

    return btif_enable_bluetooth();
}
```

initq

作用：

```
static int initq(bt_callbacks_t* callbacks)
{
    ALOGI("initq");
    if(interface_ready()==FALSE)
        return BT_STATUS_NOT_READY; //halbacks have not been initialized for the interface yet, by the adapterservice
    bt_hal_cbacks->le_lpp_write_rssi_thresh_cb   = callbacks->le_lpp_write_rssi_thresh_cb;
    bt_hal_cbacks->le_lpp_read_rssi_thresh_cb    = callbacks->le_lpp_read_rssi_thresh_cb;
    bt_hal_cbacks->le_lpp_enable_rssi_monitor_cb = callbacks->le_lpp_enable_rssi_monitor_cb;
    bt_hal_cbacks->le_lpp_rssi_threshold_evt_cb  = callbacks->le_lpp_rssi_threshold_evt_cb;
    return BT_STATUS_SUCCESS;
}
```

bluetoothCallback

说明：在andriod上层应用开启蓝牙流程主要为：Java上层 -> 调用Java本地接口 -> 调用C语言底层驱动程序

AdapterState

作用：

(摘选：AdapterState.java 第223行开始)

```
/**
 * This state machine handles Bluetooth Adapter State.
 * States:
 *      {@link OnState} : Bluetooth is on at this state
 *      {@link OffState}: Bluetooth is off at this state. This is the initial
 *      state.
 *      {@link PendingCommandState} : An enable / disable operation is pending.
 * TODO(BT): Add per process on state.
 */

public boolean processMessage(Message msg) {

    boolean isTurningOn= isTurningOn();
    boolean isTurningOff = isTurningOff();

    AdapterService adapterService = mAdapterService;
    AdapterProperties adapterProperties = mAdapterProperties;
    if ((adapterService == null) || (adapterProperties == null)) {
        Log.e(TAG,"receive message at Pending State after cleanup:" +
                  msg.what);
        return false;
    }

    switch (msg.what) {
        case USER_TURN_ON:
            if (DBG) Log.d(TAG,"CURRENT_STATE=PENDING, MESSAGE = USER_TURN_ON"
                    + ", isTurningOn=" + isTurningOn + ", isTurningOff=" + isTurningOff);
            if (isTurningOn) {
                Log.i(TAG,"CURRENT_STATE=PENDING: Alreadying turning on bluetooth... Ignoring USER_TURN_ON...");
```

```
                    } else {
                        Log.i(TAG,"CURRENT_STATE=PENDING: Deferring request USER_TURN_ON");
                        deferMessage(msg);
                    }
                    break;
                case USER_TURN_OFF:
                    if (DBG) Log.d(TAG,"CURRENT_STATE=PENDING, MESSAGE = USER_TURN_ON"
                            + ", isTurningOn=" + isTurningOn + ", isTurningOff=" + isTurningOff);
                    if (isTurningOff) {
                        Log.i(TAG,"CURRENT_STATE=PENDING: Alreadying turning off bluetooth... Ignoring USER_TURN_OFF...");
                    } else {
                        Log.i(TAG,"CURRENT_STATE=PENDING: Deferring request USER_TURN_OFF");
                        deferMessage(msg);
                    }
                    break;
                case STARTED:    {
                    if (DBG) Log.d(TAG,"CURRENT_STATE=PENDING, MESSAGE = STARTED, isTurningOn=" + isTurningOn + ",
isTurningOff=" + isTurningOff);
                    //Remove start timeout
                    removeMessages(START_TIMEOUT);

                    //Enable
                    boolean ret = adapterService.enableNative();        //在这里调用enableNative()函数，这个函数的具体实现是在
                     //com_andriod_bluetooth_btservice_AdapterService.cpp 属于JNI具体实现
                    if (!ret) {
                        Log.e(TAG, "Error while turning Bluetooth On");
                        notifyAdapterStateChange(BluetoothAdapter.STATE_OFF);
                        transitionTo(mOffState);
                    } else {
                        sendMessageDelayed(ENABLE_TIMEOUT, ENABLE_TIMEOUT_DELAY);
                    }
                }
                    break;

                case ENABLED_READY:
                    if (DBG) Log.d(TAG,"CURRENT_STATE=PENDING, MESSAGE = ENABLE_READY, isTurningOn=" + isTurningOn + ",
isTurningOff=" + isTurningOff);
                    removeMessages(ENABLE_TIMEOUT);
                    adapterProperties.onBluetoothReady();
                    mPendingCommandState.setTurningOn(false);
                    transitionTo(mOnState);
                    notifyAdapterStateChange(BluetoothAdapter.STATE_ON);
                    break;

                case SET_SCAN_MODE_TIMEOUT:
                     Log.w(TAG,"Timeout will setting scan mode..Continuing with disable...");
                     //Fall through
                case BEGIN_DISABLE: {
                    if (DBG) Log.d(TAG,"CURRENT_STATE=PENDING, MESSAGE = BEGIN_DISABLE, isTurningOn=" + isTurningOn + ",
isTurningOff=" + isTurningOff);
                    removeMessages(SET_SCAN_MODE_TIMEOUT);
                    sendMessageDelayed(DISABLE_TIMEOUT, DISABLE_TIMEOUT_DELAY);
                    boolean ret = adapterService.disableNative();
                    if (!ret) {
                        removeMessages(DISABLE_TIMEOUT);
                        Log.e(TAG, "Error while turning Bluetooth Off");
                        //FIXME: what about post enable services
                        mPendingCommandState.setTurningOff(false);
                        notifyAdapterStateChange(BluetoothAdapter.STATE_ON);
                    }
                }
```

enableNative

作用：通过调用C语言完成对蓝牙设备的控制操作。

知识点：什么是JNI函数？JNI函数是Jave Native Interface的缩写，中文名称：Java本地接口，它提供若干个API实现了Java和其他语言的通信，语言主要是C和C++两种语言。

Java的设计目的：标准的Java类库可能不支持你的程序所需的特性，或许你已经有了一个用其他语言写成的库或者程序，而你希望在Java程序中使用它。可能需要用底层语言实现一

个小型的时间比较敏感代码，比如使用汇编，而后在你的Java程序中调用。

(摘选：com_andriod_bluetooth_btservice_AdapterService.cpp 文件中的第877行)

省略……

```
static jboolean enableNative(JNIEnv* env, jobject obj) {
    ALOGV("%s:",__FUNCTION__);

    jboolean result = JNI_FALSE;
    if (!sBluetoothInterface) return result;

    int ret = sBluetoothInterface->enable();      //通过调用C语言中的函数实现蓝牙的使能。
    result = (ret == BT_STATUS_SUCCESS) ? JNI_TRUE : JNI_FALSE;          //通过返回指来判断此刻蓝牙的打开还是关闭。
    return result;
}
```

省略……

```
//-------------------------------------------------------------------------------------------------------------------------
-------------------------
```

//看到上面的代码存在的疑问有：sBluetoothInterface什么东西？为什么它可以调用enable(),下面便是它的来源:

(摘选：com_andriod_bluetooth_btservice_AdapterService.cpp 文件中的第732行)

```
static void classInitNative(JNIEnv* env, jclass clazz) {
    int err;
    hw_module_t* module;     //定义一个hw_module_t结构体类型的指针，容器

    jclass jniCallbackClass =
    env->FindClass("com/android/bluetooth/btservice/JniCallbacks");
sJniCallbacksField = env->GetFieldID(clazz, "mJniCallbacks",
    "Lcom/android/bluetooth/btservice/JniCallbacks;");

method_stateChangeCallback = env->GetMethodID(jniCallbackClass, "stateChangeCallback", "(I)V");

method_adapterPropertyChangedCallback = env->GetMethodID(jniCallbackClass,
                                                         "adapterPropertyChangedCallback",
                                                         "([I[[B)V");
method_discoveryStateChangeCallback = env->GetMethodID(jniCallbackClass,
                                                       "discoveryStateChangeCallback", "(I)V");

method_devicePropertyChangedCallback = env->GetMethodID(jniCallbackClass,
                                                         "devicePropertyChangedCallback",
                                                         "([B[I[[B)V");
method_deviceFoundCallback = env->GetMethodID(jniCallbackClass, "deviceFoundCallback", "([B)V");
method_pinRequestCallback = env->GetMethodID(jniCallbackClass, "pinRequestCallback",
                                             "([B[BIZ)V");
method_sspRequestCallback = env->GetMethodID(jniCallbackClass, "sspRequestCallback",
                                             "([B[BIII)V");

method_bondStateChangeCallback = env->GetMethodID(jniCallbackClass,
                                                  "bondStateChangeCallback", "(I[BI)V");

method_aclStateChangeCallback = env->GetMethodID(jniCallbackClass,
                                                 "aclStateChangeCallback", "(I[BI)V");
    method_setWakeAlarm = env->GetMethodID(clazz, "setWakeAlarm", "(JZ)Z");
    method_acquireWakeLock = env->GetMethodID(clazz, "acquireWakeLock", "(Ljava/lang/String;)Z");
    method_releaseWakeLock = env->GetMethodID(clazz, "releaseWakeLock", "(Ljava/lang/String;)Z");
    method_deviceMasInstancesFoundCallback = env->GetMethodID(jniCallbackClass,
                                                              "deviceMasInstancesFoundCallback",
                                                              "(I[B[Ljava/lang/String;[I[I[I)V");
    method_energyInfo = env->GetMethodID(clazz, "energyInfoCallback", "(IIJJJJ)V");

    char value[PROPERTY_VALUE_MAX];
    property_get("bluetooth.mock_stack", value, "");

    const char *id = (strcmp(value, "1")? BT_STACK_MODULE_ID : BT_STACK_TEST_MODULE_ID);      //这边的两个宏是定义在Bluetooth.h
中的
                                                                                             //BT_STACK_MODULE_ID => #define
BT_STACK_MODULE_ID "bluetooth"
                                                                                  //BT_STACK_TEST_MODULE_ID => #define
BT_STACK_TEST_MODULE_ID "bluetooth_test"


    //传入的id就相当于传入一个字符串 "xxxxxx"，第二传入的参数是一个二级指针，也就是这个指针的地址，那么它存储的
    //内容也只能是一个指针而已
    //补充(二次学习)：hw_get_module函数的功能是根据模块ID寻找硬件模块动态库(王振丽:84页)，在里面然后调用load打开动态链接库，并从
中获取硬件模块结构体地址。
```

```
    err = hw_get_module(id, (hw_module_t const**)&module);   //这句话的作用：根据传入的字符串id 去获取一个模块的地址，并保存在指针变量module中，以后访问modul就相当                                                                    //于访问这个模块了,具体函数实现请参照下一级目录

    if (err == 0) {                        //获取模块成功了，就可以使用modle所挂接的函数了 ^_^ 到这里不要忘记我们是来做什么的，我们是来获取蓝牙接口的，如果它给我们接口了，我们要
                                //怎样接收呢，其实这个时候我们是需要给它一个变量(可以比喻成一个空盒子)，让把地址放进去就可以了，我们知道存放指针的变量，只能够申明成指针
                                //变量，所以有 hw_device_t* abstraction;
        hw_device_t* abstraction;
        err = module->methods->open(module, id, &abstraction);          //可能会想这里好奇怪，为什么要&取地址，其实传入的就是指针变量的地址，至于具体这个methods是在                                                                    //哪里完成挂接的，现在还不知道。？？？？？？？？？
        if (err == 0) {      //又成功了
            bluetooth_module_t* btStack = (bluetooth_module_t *)abstraction;       //现在abstraction内已不是没有用的数据，而是一个我们获取bluetooth接口的通道，强转                                                                    //换一下,在这里有一个疑问 为什么在刚开始申明类型的时候不申请bluetooth_module_t
                    //后来向后面查看发现open函数传入的就是这个类型。但是你只要知道你指向的是什么样类型
                                                                    //很关键
            sBluetoothInterface = btStack->get_bluetooth_interface();
        } else {
           ALOGE("Error while opening Bluetooth library");
        }
    } else {
        ALOGE("No Bluetooth Library found");
    }
}

//-------------------------------------------------------------------------------------------------------------------------------
------------------------

//重点说明：我们现在所看到的东西都是JNI层的实现代码，在com_andriod_bluetooth_btservice_AdapterService.cpp 这个文件中有很多的封装好的“功能”函数，例如：
//classInitNative,initNative,cleanupNative等，那么问题来了，在java上层的实现这些“功能”函数的时候都会用到动态库，怎样知道我所调用的函数就是我想要的比如
//说在上层使用的是 _init_()就是JNI层的mokoid_init()，在这里有一个方法就是完成JNI层方法注册，可以理解为对应方法的挂接。
//为了识别方便，这里的Java上层调用的方法名称和JNI层实现的方法名相同。(详细请参考(Android底层开发技术实战详解：90页 王振丽))

(摘选：com_andriod_bluetooth_btservice_AdapterService.cpp 第1393行)

//JNI NativeMethod是JNI层注册方法，

省略……
static JNINativeMethod sMethods[] = {
    /* name, signature, funcPtr */
    {"classInitNative", "()V", (void *) classInitNative},
    {"initNative", "()Z", (void *) initNative},
    {"cleanupNative", "()V", (void*) cleanupNative},
    {"ssrcleanupNative", "(Z)V", (void*) ssrcleanupNative},
    {"enableNative", "()Z",  (void*) enableNative},
    {"disableNative", "()Z",  (void*) disableNative},
    {"setAdapterPropertyNative", "(I[B)Z", (void*) setAdapterPropertyNative},
    {"getAdapterPropertiesNative", "()Z", (void*) getAdapterPropertiesNative},
    {"getAdapterPropertyNative", "(I)Z", (void*) getAdapterPropertyNative},
    {"getDevicePropertyNative", "([BI)Z", (void*) getDevicePropertyNative},
    {"setDevicePropertyNative", "([BI[B)Z", (void*) setDevicePropertyNative},
    {"startDiscoveryNative", "()Z", (void*) startDiscoveryNative},
    {"cancelDiscoveryNative", "()Z", (void*) cancelDiscoveryNative},
    {"createBondNative", "([BI)Z", (void*) createBondNative},
    {"removeBondNative", "([B)Z", (void*) removeBondNative},
    {"cancelBondNative", "([B)Z", (void*) cancelBondNative},
    {"getConnectionStateNative", "([B)I", (void*) getConnectionStateNative},
    {"pinReplyNative", "([BZI[B)Z", (void*) pinReplyNative},
    {"sspReplyNative", "([BIZI)Z", (void*) sspReplyNative},
    {"getRemoteServicesNative", "([B)Z", (void*) getRemoteServicesNative},
    {"getRemoteMasInstancesNative", "([B)Z", (void*) getRemoteMasInstancesNative},
    {"connectSocketNative", "([BI[BII)I", (void*) connectSocketNative},
    {"createSocketChannelNative", "(ILjava/lang/String;[BII)I",
     (void*) createSocketChannelNative},
    {"configHciSnoopLogNative", "(Z)Z", (void*) configHciSnoopLogNative},
    {"alarmFiredNative", "()V", (void *) alarmFiredNative},
    {"readEnergyInfo", "()I", (void*) readEnergyInfo},
    {"getSocketOptNative", "(III[B)I", (void*) getSocketOptNative},
    {"setSocketOptNative", "(III[BI)I", (void*) setSocketOptNative}
```

```
    /*[FEATURE]-Add-BEGIN by TCTNB.(Qianbo Pan), 2013/09/09, for FR512357 Add Bluetooth test mode api*/
    ,{"setTestModeNative", "(I)I", (void *)setTestModeNative},
    {"setBtChannelNative","(I)I",(void *)setBtChannelNative}
    /*[FEATURE]-Add-END by TCTNB.(Qianbo Pan)*/
};
```

省略……

hw_get_module

作用：返回一个模块的地址，通过module返回过去

(摘选：hardware.c 第197行)

```
int hw_get_module(const char *id, const struct hw_module_t **module)
{
    return hw_get_module_by_class(id, NULL, module);      //调用另外一个函数具体函数实现方式参照如下：
}
```

(摘选：hardware.c 第145行)

```
int hw_get_module_by_class(const char *class_id, const char *inst,
                           const struct hw_module_t **module)
{
    int i;
    char prop[PATH_MAX];
    char path[PATH_MAX];
    char name[PATH_MAX];
    char prop_name[PATH_MAX];

    if (inst)
        snprintf(name, PATH_MAX, "%s.%s", class_id, inst);
    else
        strlcpy(name, class_id, PATH_MAX);                //将id字符串拷贝到name数组中

    /*
     * Here we rely on the fact that calling dlopen multiple times on
     * the same .so will simply increment a refcount (and not load
     * a new copy of the library).
     * We also assume that dlopen() is thread-safe.
     */

    //下面使用多种方式去寻找匹配的模块，属性，配置，默认

    /* First try a property specific to the class and possibly instance */
    snprintf(prop_name, sizeof(prop_name), "ro.hardware.%s", name);
    if (property_get(prop_name, prop, NULL) > 0) {
        if (hw_module_exists(path, sizeof(path), name, prop) == 0) {
                                                //注意这里的name等价于class_id，不需要怀疑。请参考 strlcpy(name,
class_id, PATH_MAX);
            goto found;  //发现
        }
    }

    /* Loop through the configuration variants looking for a module */
    for (i=0 ; i<HAL_VARIANT_KEYS_COUNT; i++) {
        if (property_get(variant_keys[i], prop, NULL) == 0) {
            continue;
        }
        if (hw_module_exists(path, sizeof(path), name, prop) == 0) {
            goto found;  //发现
        }
    }

    /* Nothing found, try the default */
    if (hw_module_exists(path, sizeof(path), name, "default") == 0) {
        goto found;                  //发现
    }

    //如果没有找到只能返回一个错误码
    return -ENOENT;

found:
    /* load the module, if this fails, we're doomed, and we should not try
```

```
     * to load a different variant. */

    //如果找到了相对应的模块
    /**
      * @class_id: 传入的字符串，“bluetooth”
      * @path:
      * @module:专门用来保存模块的返回地址的指针变量，提供给上面的函数
     **/
    return load(class_id, path, module);
}
```

load

作用：**这是很关键的一步！因为在这之前所做的努力，最终就是能够打开动态库，将模块的地址返回过去**

(摘选：hardware.c　　第61行)

这个函数在上面传过来的参数为：class_id、 path,、module

```
/**
 * Load the file defined by the variant and if successful
 * return the dlopen handle and the hmi.
 * @return 0 = success, !0 = failure.
 */
static int load(const char *id,const char *path,const struct hw_module_t **pHmi)
{
    int status;
    void *handle;
    struct hw_module_t *hmi;                          //临时变量

    /*
     * load the symbols resolving undefined symbols before
     * dlopen returns. Since RTLD_GLOBAL is not or'd in with
     * RTLD_NOW the external symbols will not be global
     */

    //注意这里，根据路径去打开一个动态库

    handle = dlopen(path, RTLD_NOW);                  //重点！作用通过路径->获取句柄，这个地方以前重点学习过不再赘述
    if (handle == NULL) {
        char const *err_str = dlerror();
        ALOGE("load: module=%s\n%s", path, err_str?err_str:"unknown");
        status = -EINVAL;
        goto done;
    }

    /* Get the address of the struct hal_module_info. */
    /* 获取结构体hal_module_info 地址*/

    const char *sym = HAL_MODULE_INFO_SYM_AS_STR;
    hmi = (struct hw_module_t *)dlsym(handle, sym); //重点！作用通过句柄->获取地址，根据这个地方以前重点学习过不再赘述

    if (hmi == NULL) {
        ALOGE("load: couldn't find symbol %s", sym);
        status = -EINVAL;
        goto done;
    }

    /* Check that the id matches */
    if (strcmp(id, hmi->id) != 0) {
        ALOGE("load: id=%s != hmi->id=%s", id, hmi->id);
        status = -EINVAL;
        goto done;
    }

    hmi->dso = handle;                                //句柄保留

    /* success */
    status = 0;

    done:
    if (status != 0) {
```

```
        hmi = NULL;
        if (handle != NULL) {
            dlclose(handle);
            handle = NULL;
        }
    } else {
        ALOGV("loaded HAL id=%s path=%s hmi=%p handle=%p",
                id, path, *pHmi, handle);
    }

    *pHmi = hmi;            //走到这里就是我们期待已久的结果，因为我们确实通过是动态库获取一个机构体指针，很不容易！！

    return status;
}
```

hw_module_exists

作用：根据名称，组合一个新路径，然后判断这个路径是否可以访问，如果可以访问返回值为0，新路径保存在path中

(摘选：hardware.c　　第125行)

```
/*
 * Check if a HAL with given name and subname exists, if so return 0, otherwise
 * otherwise return negative.  On success path will contain the path to the HAL.
 */
static int hw_module_exists(char *path, size_t path_len, const char *name,
                            const char *subname)
{
    snprintf(path, path_len, "%s/%s.%s.so", //组合新路径，有没有发现亮点看到这个我们应该知道他想做什么了，就是通过动态库找到对应的结构体指针。
             HAL_LIBRARY_PATH2, name, subname);
    if (access(path, R_OK) == 0)            //探测这个路径是否可以去访问，我们要尝试访问一下，看究竟可不可以成功。
        return 0;

    snprintf(path, path_len, "%s/%s.%s.so",
             HAL_LIBRARY_PATH1, name, subname);
    if (access(path, R_OK) == 0)
        return 0;

    return -ENOENT;
}
```

module->methods->open

作用：

(摘选：hardwart.h　　第86行)

module->methods->open

```
//这个结构体是上面module的原型，不过这个结构体我们从上一个函数已经得到了，所以就可以直接去方法methods了

typedef struct hw_module_t {
    /** tag must be initialized to HARDWARE_MODULE_TAG */
    uint32_t tag;
    uint16_t module_api_version;
#define version_major module_api_version
    uint16_t hal_api_version;
#define version_minor hal_api_version

    /** Identifier of module */
    const char *id;

    /** Name of this module */
    const char *name;

    /** Author/owner/implementor of the module */
    const char *author;

    /** Modules methods */
    struct hw_module_methods_t* methods;

    /** module's dso */
```

```
    void* dso;

#ifdef __LP64__
    uint64_t reserved[32-7];
#else
    /** padding to 128 bytes, reserved for future use */
    uint32_t reserved[32-7];
#endif

} hw_module_t;

//-------------------------------------------------------------------------------------------------------------------
```

//这个时候我们可能会想，methods方法的挂接在哪一个具体函数里面的呢？
//我暂时理解，从上面获取的module就是下面这个结构体，并且实现了对应的函数，所以我们可以继续跟下去。

//并且认为这样的结构体已经完成了初始化工作！

```
struct hw_module_t HAL_MODULE_INFO_SYM = {
    .tag = HARDWARE_MODULE_TAG,
    .version_major = 1,
    .version_minor = 0,
    .id = BT_HARDWARE_MODULE_ID,
    .name = "Bluetooth Stack",
    .author = "The Android Open Source Project",
    .methods = &bt_stack_module_methods                    //挂接的函数请参照下面：
};

//-------------------------------------------------------------------------------------------------------------------

static struct hw_module_methods_t bt_stack_module_methods = {
    .open = open_bluetooth_stack,
};

//-------------------------------------------------------------------------------------------------------------------
```

//函数直接对应open挂接的方法，这里主要实现open函数的具体内容。

```
static int open_bluetooth_stack (const struct hw_module_t* module, char const* name,
                                 struct hw_device_t** abstraction)
{
    UNUSED(name);

    bluetooth_device_t *stack = malloc(sizeof(bluetooth_device_t) );
    if (stack)
    {
        memset(stack, 0, sizeof(bluetooth_device_t) );
        stack->common.tag = HARDWARE_DEVICE_TAG;
        stack->common.version = 0;
        stack->common.module = (struct hw_module_t*)module;
        stack->common.close = close_bluetooth_stack;
        stack->get_bluetooth_interface = bluetooth__get_bluetooth_interface;    //坑人啊~！辛辛苦苦获取模块，最后就得到这个
```

结构体指针，这个结构体内容
是指针，里面挂接了好多函数，你可以找一下！bluetooth__get_bluetooth_interface就是函数，函数返回的就是一个结构体地址。bluetooth.c第796行

```
        }
        else
        {
            ALOGE("%s: malloc() returned NULL", __FUNCTION__);
        }
        *abstraction = (struct hw_device_t*)stack;
        return 0;
    }
```

enable

作用：这部分主要是C语言部分的实现。

(摘选：bluetooth.c 第208行)

```
static int enable( void )
{
    ALOGI("enable");
```

```
    /* sanity check */
    if (interface_ready() == FALSE)     //判断bt_hal_cbacks接口信息是否准备完成，如果没有准备完成则返回 BT_STATUS_NOT_READY
        return BT_STATUS_NOT_READY;     //如果定义在结构体中的函数指针挂接已经完成，那么就使能蓝牙。
    return btif_enable_bluetooth();
}
```

btif_enable_bluetooth

作用：Performs chip power on and kickstarts OS scheduler 执行打开电源操作，并且启动操作系统调度。

(摘选：btif_core.c 第546行)

```
bt_status_t btif_enable_bluetooth(void)
{
    BTIF_TRACE_DEBUG("BTIF ENABLE BLUETOOTH");

    if (btif_core_state != BTIF_CORE_STATE_DISABLED &&
        btif_core_state != BTIF_CORE_STATE_INITIALIZED)
    {
        ALOGD("not disabled\n");
        return BT_STATUS_DONE;
    }

    btif_core_state = BTIF_CORE_STATE_ENABLING;//设定为开启状态

    bt_disabled = FALSE;

    init_slot_lock(&mutex_bt_disable);
    /* Create the GKI tasks and run them */
    bte_main_enable();

    return BT_STATUS_SUCCESS;
}
```

bte_main_enable

作用：是enable函数的具体实现，

```
/*******************************************************************************
** Function         bte_main_enable
** Description      BTE MAIN API - Creates all the BTE tasks. Should be called
**                  part of the Bluetooth stack enable sequence
** Returns          None
*******************************************************************************/
void bte_main_enable()
{
    APPL_TRACE_DEBUG("%s", __FUNCTION__);

    /* Initialize BTE control block */
    //初始化BTE控制模块
    BTE_Init();

    lpm_enabled = FALSE;

    //创建BTU_TASK进程
    GKI_create_task((TASKPTR)btu_task, BTU_TASK, BTE_BTU_TASK_STR,
                    (UINT16 *) ((UINT8 *)bte_btu_stack + BTE_BTU_STACK_SIZE),
                    sizeof(bte_btu_stack));

    //打开HCI 和 厂商模块控制
    bte_hci_enable();

    GKI_run();
}
```

BTE_Init

作用：

```
/*******************************************************************************
** Function         BTE_Init
** Description      Initializes the BTU control block.
**                  NOTE: Must be called before creating any tasks
**                      (RPC, BTU, HCIT, APPL, etc.)
** Returns          void
```

```
*****************************************************************************/
void BTE_Init(void)
{
    int i = 0;

    memset (&btu_cb, 0, sizeof (tBTU_CB));
    btu_cb.hcit_acl_pkt_size = BTU_DEFAULT_DATA_SIZE + HCI_DATA_PREAMBLE_SIZE;
#if (BLE_INCLUDED == TRUE)
    btu_cb.hcit_ble_acl_pkt_size = BTU_DEFAULT_BLE_DATA_SIZE + HCI_DATA_PREAMBLE_SIZE;
#endif
    btu_cb.trace_level = HCI_INITIAL_TRACE_LEVEL;

    for ( i = 0; i < BTU_MAX_LOCAL_CTRLS; i++ ) /* include BR/EDR */
        btu_cb.hci_cmd_cb[i].cmd_window = 1;
}
```

bte_hci_enable

作用：

```
/*****************************************************************************
** Function         bte_hci_enable
** Description      Enable HCI & Vendor modules
** Returns          None
*****************************************************************************/
static void bte_hci_enable(void)
{
    APPL_TRACE_DEBUG("%s", __FUNCTION__);

    if (bt_hc_if)
    {
        int result = bt_hc_if->init(&hc_callbacks, btif_local_bd_addr.address);
        APPL_TRACE_EVENT("libbt-hci init returns %d", result);

        assert(result == BT_HC_STATUS_SUCCESS);

        if (hci_logging_enabled == TRUE || hci_logging_config == TRUE)
            bt_hc_if->logging(BT_HC_LOGGING_ON, hci_logfile, hci_save_log);

#if (defined (BT_CLEAN_TURN_ON_DISABLED) && BT_CLEAN_TURN_ON_DISABLED == TRUE)
        APPL_TRACE_DEBUG("%s  Not Turninig Off the BT before Turninig ON", __FUNCTION__);

        /* Do not power off the chip before powering on  if BT_CLEAN_TURN_ON_DISABLED flag
         is defined and set to TRUE to avoid below mentioned issue.

         Wingray kernel driver maintains a combined  counter to keep track of
         BT-Wifi state. Invoking  set_power(BT_HC_CHIP_PWR_OFF) when the BT is already
         in OFF state causes this counter to be incorrectly decremented and results in undesired
         behavior of the chip.

         This is only a workaround and when the issue is fixed in the kernel this work around
         should be removed. */
#else
        /* toggle chip power to ensure we will reset chip in case
           a previous stack shutdown wasn't completed gracefully */
        //这里的bt_hc_if是通过bt_hc_get_interface得到bluetoothHCLibInterface结构体的指针，并传递给bt_hc_if
        //所以在执行set_power操作的时候，等价于执行bluetoothHCLibInterface对应的函数。
        bt_hc_if->set_power(BT_HC_CHIP_PWR_OFF);
#endif
        bt_hc_if->set_power(BT_HC_CHIP_PWR_ON);

        preload_start_wait_timer();
        bt_hc_if->preload(NULL);
    }
}
```

bt_hc_if的由来

作用：获取的bluetoothHCLibInterface结构体的首地址。

(摘选：bt_hci_bdroid.c 第518行)

```
//这个结构体绑定了通过使用函数指针，挂接了很多函数。
//在后面的使用中只要获取到这个结构体的地址就可以轻松
//访问到里面的相对应的函数。
static const bt_hc_interface_t bluetoothHCLibInterface = {
```

```
    sizeof(bt_hc_interface_t),
    init,
    set_power,
    lpm,
    preload,
    postload,
    transmit_buf,
    logging,
    cleanup,
    tx_hc_cmd,
    ssr_cleanup
};

/*******************************************************************************
** Function        bt_hc_get_interface
** Description     Caller calls this function to get API instance
** Returns         API table
*******************************************************************************/
//获取接口，将结构体 bluetoothHCLibInterface 挂载的函数指针返回出去
const bt_hc_interface_t *bt_hc_get_interface(void)
{
    return &bluetoothHCLibInterface;
}
```

bt_hc_if->set_power

作用：蓝牙芯片电源控制程序

(摘选：bt_hci_bdroid.c 第376行)

```
/** Chip power control */
//调用这个函数传入的参数有：BT_VND_PWR_ON 或者 BT_VND_PWR_OFF
static void set_power(bt_hc_chip_power_state_t state)
{
    int pwr_state;

    BTHCDBG("set_power %d", state);

    /* Calling vendor-specific part */
    pwr_state = (state == BT_HC_CHIP_PWR_ON) ? BT_VND_PWR_ON : BT_VND_PWR_OFF;

    vendor_send_command(BT_VND_OP_POWER_CTRL, &pwr_state);          //将传入的指令通过vendor_send_command发送出去
}
```

vendor_send_command

作用：

(摘选：vendor.c 第109行)

```
//Sends a vendor-specific command to the library.
//含义：发送厂商指令的命令到一个库里。
int vendor_send_command(bt_vendor_opcode_t opcode, void *param) {
  assert(vendor_interface != NULL);

  if (vendor_interface)
    return vendor_interface->op(opcode, param);       //看到这里可能会思考，vendor_interface的指针对应的实例操作是怎么来的?
  else                                                //这个指针的获取可以追溯到vendor_open函数，程序只有调用这个函数，我们
    return -1;                                        //才可以获取这个指针，这个函数的初始化调用详细 参见：bt_hci_bdroid.c
}                                                     //第337行，这个函数的调用是在init函数中被完成。

//-----------------------------------------------------------------------------
```

(摘选：vendor.c 第062行)

```
bool vendor_open(const uint8_t *local_bdaddr) {
  assert(lib_handle == NULL);

  //知识点：void* dlopen(const char *pathname, int mode)
  //功能：按照指定的模式打开动态库文件，并返回一个句柄给调用进程
  //使用dlclose()关闭一个已经打开的动态库
  //RTLD_LAZY 暂缓决定
```

```
  //RTLD_NOW  立即决定
  lib_handle = dlopen(VENDOR_LIBRARY_NAME, RTLD_NOW);
  if (!lib_handle) {
    ALOGE("%s unable to open %s: %s", __func__, VENDOR_LIBRARY_NAME, dlerror());
    goto error;
  }

  //知识点：void* dlsym(void* phandle, const char* symbol)
  //功能：根据动态链接库操作句柄(phandle)与符号(symbol),返回符号对应的地址
  //例如：在so库中定义了一个void myteset()函数，在使用这个函数的时候我们需要先定义一个指针变量 void
(*p_mytest)()然后将返回值赋值给它
  //在这里我们获取 BLUETOOTH_VENDOR_LIB_INTERFACE 对应的接口地址
  vendor_interface = (bt_vendor_interface_t *)dlsym(lib_handle, VENDOR_LIBRARY_SYMBOL_NAME);
  if (!vendor_interface) {
    ALOGE("%s unable to find symbol %s in %s: %s", __func__, VENDOR_LIBRARY_SYMBOL_NAME, VENDOR_LIBRARY_NAME,
dlerror());
    goto error;
  }

  //调用动态库函数中的init的函数完成厂商中回调函数的挂接，通过分析可以知道这部分函数主要功能是：当发送指定厂商
指令给动态库时，会反馈结果
  //表示动态库完成某一种功能操作后，对命令的执行一些情况( 暂时这么理解 )
  int status = vendor_interface->init(&vendor_callbacks, (unsigned char *)local_bdaddr);
  if (status) {
    ALOGE("%s unable to initialize vendor library: %d", __func__, status);
    goto error;
  }

  return true;

error:;
  vendor_interface = NULL;
  if (lib_handle)
    dlclose(lib_handle);
  lib_handle = NULL;
  return false;
}

//-----------------------------------------------------------------------------
```

(摘选：vendor.c 第062行)

```
//回调函数初始化int status = vendor_interface->init(&vendor_callbacks, (unsigned char *)local_bdaddr);
//挂接的结构体参数其实是一些函数指针，通过对这部分函数指针的访问我们蓝牙模块获取命令的执行情况，在这些函数里
//可能我们还有做点其他的事情。
static const bt_vendor_callbacks_t vendor_callbacks = {
  sizeof(vendor_callbacks),
  firmware_config_cb,
  sco_config_cb,
  low_power_mode_cb,
  sco_audiostate_cb,
  buffer_alloc,
  buffer_free,
  transmit_cb,
  epilog_cb
};
```

opt

作用：发送控制命令的vendor_interface->op(opcode, param);在前面的代码中没有找到对应的函数。
在另外一个文件中发现它是以另外一种形式出现。

(摘选：bt_vendor_lib.h 第379行)

```
extern const bt_vendor_interface_t BLUETOOTH_VENDOR_LIB_INTERFACE;

//通过这一条语句则定义了一个bt_vendor_interface_t结构体类型的变量，变量名称为：
BLUETOOTH_VENDOR_LIB_INTERFACE
//对于结构体成语的赋值在另外一个文件。
```

(摘选：bt_vendor_qcom.c 第243行)

```
// Entry point of DLib
const bt_vendor_interface_t BLUETOOTH_VENDOR_LIB_INTERFACE = {
```

```
    sizeof(bt_vendor_interface_t),
    init,                       //初始化函数
    op,             //op函数 对于这部分代码的实现 如下：
    cleanup         //
};

//-----------------------------------------------------------------------
```

说明：Requested operations请求操作的代码相对比较长，根据传递的操作码进行相应的操作。

(摘选：bt_vendor_qcom.c 第576行)

```
/** Requested operations */
static int op(bt_vendor_opcode_t opcode, void *param)
{
    int retval = 0;
    int nCnt = 0;
    int nState = -1;
    bool is_ant_req = false;
    char wipower_status[PROPERTY_VALUE_MAX];

    ALOGV("bt-vendor : op for %d", opcode);

    switch(opcode)
    {
        case BT_VND_OP_POWER_CTRL:
            {
                nState = *(int *) param;            //将传递的参数赋值给nState
                ALOGI("bt-vendor : BT_VND_OP_POWER_CTRL: %s",
                        (nState == BT_VND_PWR_ON)? "On" : "Off" );

                switch(btSocType)                   //根据蓝牙SOC类型确定操作
                {
                    case BT_SOC_DEFAULT:
                        if (readTrpState())
                        {
                           ALOGI("bt-vendor : resetting BT status");
                           hw_config(BT_VND_PWR_OFF);
                        }
                        retval = hw_config(nState);
                        if(nState == BT_VND_PWR_ON
                           && retval == 0
                           && is_hw_ready() == TRUE){
                            retval = 0;
                        }
                        else {
                            retval = -1;
                        }
                        break;
                    case BT_SOC_ROME:
                    case BT_SOC_AR3K:
                        /* BT Chipset Power Control through Device Tree Node */
                        retval = bt_powerup(nState);
                     default:
                        break;
                }
            }
            break;

        case BT_VND_OP_FW_CFG:
            {
                // call hciattach to initalize the stack
                if(bt_vendor_cbacks){
                   ALOGI("Bluetooth Firmware and transport layer are initialized");
                   bt_vendor_cbacks->fwcfg_cb(BT_VND_OP_RESULT_SUCCESS);
                }
                else{
                   ALOGE("bt_vendor_cbacks is null");
                   ALOGE("Error : hci, smd initialization Error");
                   retval = -1;
                }
            }
            break;
```

```
        case BT_VND_OP_SCO_CFG:
            {
                if (bt_vendor_cbacks)
                    bt_vendor_cbacks->scocfg_cb(BT_VND_OP_RESULT_SUCCESS); //dummy
            }
            break;
#ifdef BT_SOC_TYPE_ROME
        case BT_VND_OP_ANT_USERIAL_OPEN:
                ALOGI("bt-vendor : BT_VND_OP_ANT_USERIAL_OPEN");
                is_ant_req = true;
                //fall through
#endif
          //这里进行串口的初始化 波特率，奇偶校验
       case BT_VND_OP_USERIAL_OPEN:
            {
                int (*fd_array)[] = (int (*)[]) param;
                int idx, fd;
                ALOGI("bt-vendor : BT_VND_OP_USERIAL_OPEN");
                switch(btSocType)
                {
                    case BT_SOC_DEFAULT:
                        {
                            if(bt_hci_init_transport(pFd) != -1){
                                int (*fd_array)[] = (int (*) []) param;
                                (*fd_array)[CH_CMD] = pFd[0];
                                (*fd_array)[CH_EVT] = pFd[0];
                                (*fd_array)[CH_ACL_OUT] = pFd[1];
                                (*fd_array)[CH_ACL_IN] = pFd[1];
                            }
                            else {
                                retval = -1;
                                break;
                            }
                            retval = 2;
                        }
                    break;
                case BT_SOC_AR3K:
                    {
                        fd = userial_vendor_open((tUSERIAL_CFG *) &userial_init_cfg);
                        if (fd != -1) {
                            for (idx=0; idx < CH_MAX; idx++)
                                (*fd_array)[idx] = fd;
                                 retval = 1;
                        }
                        else {
                            retval = -1;
                            break;
                        }

                        /* Vendor Specific Process should happened during userial_open process
                            After userial_open, rx read thread is running immediately,
                            so it will affect VS event read process.
                        */
                        if(ath3k_init(fd,3000000,115200,NULL,&vnd_userial.termios)<0)
                            retval = -1;
                    }
                    break;
                case BT_SOC_ROME:
                    {
                        if (!is_soc_initialized()) {
                            fd = userial_vendor_open((tUSERIAL_CFG *) &userial_init_cfg);
                            if (fd < 0) {
                                ALOGE("userial_vendor_open returns err");
                                retval = -1;
                            } else {
                                /* Clock on */
                                userial_clock_operation(fd, USERIAL_OP_CLK_ON);
                                ALOGD("userial clock on");
                                property_get("ro.bluetooth.wipower", wipower_status, false);
                                if(strcmp(wipower_status, "true") == 0) {
                                   /* wait for embedded mode startup */
                                    usleep(WAIT_TIMEOUT);
                                    check_embedded_mode(fd);
```

```
                } else {
                    ALOGI("Wipower not enabled");
                }
                ALOGV("rome_soc_init is started");
                property_set("wc_transport.soc_initialized", "0");
                /* Always read BD address from NV file */
                if(!bt_vendor_nv_read(1, vnd_local_bd_addr))
                {
                   /* Since the BD address is configured in boot time We should not be
here */
                   ALOGI("Failed to read BD address. Use the one from bluedroid
stack/ftm");
                }
                if(rome_soc_init(fd,vnd_local_bd_addr)<0) {
                    retval = -1;
                    userial_clock_operation(fd, USERIAL_OP_CLK_OFF);
                } else {
                    ALOGV("rome_soc_init is completed");
                    property_set("wc_transport.soc_initialized", "1");
                    userial_clock_operation(fd, USERIAL_OP_CLK_OFF);
                    /*Close the UART port*/
                    close(fd);
                }
            }
            }

            property_set("wc_transport.clean_up","0");
            if (retval != -1) {
#ifdef BT_SOC_TYPE_ROME
                start_hci_filter();
                if (is_ant_req) {
                    ALOGV("connect to ant channel");
                    ant_fd = fd = connect_to_local_socket("ant_sock");
                }
                else
#endif
                {
                    ALOGV("connect to bt channel");
                    vnd_userial.fd = fd = connect_to_local_socket("bt_sock");
                }

                if (fd != -1) {
                    ALOGV("%s: received the socket fd: %d is_ant_req: %d\n",
                                                __func__, fd, is_ant_req);
                    for (idx=0; idx < CH_MAX; idx++)
                        (*fd_array)[idx] = fd;
                        retval = 1;
                    }
                else {
                    retval = -1;
                }
                        } else {
                            if (fd >= 0)
                                close(fd);
                        }
                    }
                    break;
                default:
                    ALOGE("Unknown btSocType: 0x%x", btSocType);
                    break;
                }
            }
            break;
#ifdef BT_SOC_TYPE_ROME
        case BT_VND_OP_ANT_USERIAL_CLOSE:
            {
                ALOGI("bt-vendor : BT_VND_OP_ANT_USERIAL_CLOSE");
                property_set("wc_transport.clean_up","1");
                if (ant_fd != -1) {
                    ALOGE("closing ant_fd");
                    close(ant_fd);
                    ant_fd = -1;
                }
            }
```

```
                break;
#endif
        case BT_VND_OP_USERIAL_CLOSE:
            {
                ALOGI("bt-vendor : BT_VND_OP_USERIAL_CLOSE btSocType: %d", btSocType);
                switch(btSocType)
                {
                    case BT_SOC_DEFAULT:
                         bt_hci_deinit_transport(pFd);
                         break;

                     case BT_SOC_ROME:
                     case BT_SOC_AR3K:
                        property_set("wc_transport.clean_up","1");
                        userial_vendor_close();
                        break;
                    default:
                        ALOGE("Unknown btSocType: 0x%x", btSocType);
                        break;
                }
            }
            break;

        case BT_VND_OP_GET_LPM_IDLE_TIMEOUT:
            if (btSocType ==  BT_SOC_AR3K) {
                uint32_t *timeout_ms = (uint32_t *) param;
                *timeout_ms = 1000;
            }
            break;

        case BT_VND_OP_LPM_SET_MODE:
            if(btSocType ==  BT_SOC_AR3K) {
                uint8_t *mode = (uint8_t *) param;

                if (*mode) {
                    lpm_set_ar3k(UPIO_LPM_MODE, UPIO_ASSERT, 0);
                }
                else {
                    lpm_set_ar3k(UPIO_LPM_MODE, UPIO_DEASSERT, 0);
                }
                if (bt_vendor_cbacks )
                    bt_vendor_cbacks->lpm_cb(BT_VND_OP_RESULT_SUCCESS);
            }
            else {
                if (bt_vendor_cbacks)
                    bt_vendor_cbacks->lpm_cb(BT_VND_OP_RESULT_SUCCESS); //dummy
            }
            break;

        case BT_VND_OP_LPM_WAKE_SET_STATE:
            {
                switch(btSocType)
                {
                    case BT_SOC_ROME:
                        {
                            uint8_t *state = (uint8_t *) param;
                            uint8_t wake_assert = (*state == BT_VND_LPM_WAKE_ASSERT) ? \
                                BT_VND_LPM_WAKE_ASSERT : BT_VND_LPM_WAKE_DEASSERT;

                            if (wake_assert == 0)
                                ALOGV("ASSERT: Waking up BT-Device");
                            else if (wake_assert == 1)
                                ALOGV("DEASSERT: Allowing BT-Device to Sleep");

#ifdef QCOM_BT_SIBS_ENABLE
                            if(bt_vendor_cbacks){
                                ALOGI("Invoking HCI H4 callback function");
                               bt_vendor_cbacks->lpm_set_state_cb(wake_assert);
                            }
#endif
                        }
                        break;
                    case BT_SOC_AR3K:
                        {
```

```
                                uint8_t *state = (uint8_t *) param;
                                uint8_t wake_assert = (*state == BT_VND_LPM_WAKE_ASSERT) ? \
                                                        UPIO_ASSERT : UPIO_DEASSERT;
                                lpm_set_ar3k(UPIO_BT_WAKE, wake_assert, 0);
                            }
                        case BT_SOC_DEFAULT:
                            break;
                        default:
                            ALOGE("Unknown btSocType: 0x%x", btSocType);
                            break;
                        }
                    }
                    break;
                case BT_VND_OP_EPILOG:
                    {
#if (HW_NEED_END_WITH_HCI_RESET == FALSE)
                        if (bt_vendor_cbacks)
                        {
                            bt_vendor_cbacks->epilog_cb(BT_VND_OP_RESULT_SUCCESS);
                        }
#else
                        switch(btSocType)
                        {
                          case BT_SOC_ROME:
                            {
                                char value[PROPERTY_VALUE_MAX] = {'\0'};
                                property_get("wc_transport.hci_filter_status", value, "0");
                                if(is_soc_initialized()&& (strcmp(value,"1") == 0))
                                {
                                   hw_epilog_process();
                                }
                                else
                                {
                                  if (bt_vendor_cbacks)
                                    {
                                      ALOGE("vendor lib epilog process aborted");
                                      bt_vendor_cbacks->epilog_cb(BT_VND_OP_RESULT_SUCCESS);
                                    }
                                }
                            }
                                break;
                          default:
                                hw_epilog_process();
                                break;
                        }
#endif
                    }
                    break;
                case BT_VND_OP_GET_LINESPEED:
                    {
                        retval = -1;
                        switch(btSocType)
                        {
                            case BT_SOC_ROME:
                                if(!is_soc_initialized()) {
                                    ALOGE("BT_VND_OP_GET_LINESPEED: error"
                                    " - transport driver not initialized!");
                                }else {
                                    retval = 3000000;
                                }
                                break;
                            default:
                                retval = userial_vendor_get_baud();
                                break;
                         }
                        break;
                    }
            }

            return retval;
        }
```

uart_init

作用：蓝牙在通信的过程中，究竟是用的USB还是UART进行信息传输的很是让人疑惑，

这一节主要讲述蓝牙HCI接口通信部分的初始化工作。

(摘选：bt_vendor_qcom.c 第649行)

```
        case BT_VND_OP_USERIAL_OPEN:
            {
                int (*fd_array)[] = (int (*)[]) param;          //fd_array是一个指向数组的指针，数组的类型是int型
                int idx, fd;                                    //指针param是一个void* 类型的
                ALOGI("bt-vendor : BT_VND_OP_USERIAL_OPEN");
                switch(btSocType)
                {
                    case BT_SOC_DEFAULT:
                        {
                            if(bt_hci_init_transport(pFd) != -1){
                                int (*fd_array)[] = (int (*) []) param;

                                    (*fd_array)[CH_CMD] = pFd[0];
                                    (*fd_array)[CH_EVT] = pFd[0];
                                    (*fd_array)[CH_ACL_OUT] = pFd[1];
                                    (*fd_array)[CH_ACL_IN] = pFd[1];
                            }
                            else {
                                retval = -1;
                                break;
                            }
                            retval = 2;
                        }
                        break;
                case BT_SOC_AR3K:
                    {
                        fd = userial_vendor_open((tUSERIAL_CFG *) &userial_init_cfg);
                        if (fd != -1) {
                            for (idx=0; idx < CH_MAX; idx++)        //CH_MAX最大通道数量
                                (*fd_array)[idx] = fd;              //数组的成员是int类型，将fd保存在数组中
                                 retval = 1;                        //返回值设置为1
                        }
                        else {
                            retval = -1;                            //失败
                            break;
                        }

                        /* Vendor Specific Process should happened during userial_open process
                            After userial_open, rx read thread is running immediately,
                            so it will affect VS event read process.
                        */
                        if(ath3k_init(fd,3000000,115200,NULL,&vnd_userial.termios)<0)
                            retval = -1;
                    }
                    break;
                case BT_SOC_ROME:
                    {
                        if (!is_soc_initialized()) {
                            fd = userial_vendor_open((tUSERIAL_CFG *) &userial_init_cfg);
                            if (fd < 0) {
                                ALOGE("userial_vendor_open returns err");
                                retval = -1;
                            } else {
                                /* Clock on */
                                userial_clock_operation(fd, USERIAL_OP_CLK_ON);
                                ALOGD("userial clock on");
                                property_get("ro.bluetooth.wipower", wipower_status, false);
                                if(strcmp(wipower_status, "true") == 0) {
                                   /* wait for embedded mode startup */
                                    usleep(WAIT_TIMEOUT);
                                    check_embedded_mode(fd);
                                } else {
                                    ALOGI("Wipower not enabled");
                                }
                                ALOGV("rome_soc_init is started");
                                property_set("wc_transport.soc_initialized", "0");
                                /* Always read BD address from NV file */
                                if(!bt_vendor_nv_read(1, vnd_local_bd_addr))
                                {
                                   /* Since the BD address is configured in boot time We should not be here */
```

```
                    ALOGI("Failed to read BD address. Use the one from bluedroid stack/ftm");
                }
                if(rome_soc_init(fd,vnd_local_bd_addr)<0) {
                    retval = -1;
                    userial_clock_operation(fd, USERIAL_OP_CLK_OFF);
                } else {
                    ALOGV("rome_soc_init is completed");
                    property_set("wc_transport.soc_initialized", "1");
                    userial_clock_operation(fd, USERIAL_OP_CLK_OFF);
                    /*Close the UART port*/
                    close(fd);
                }
            }
        }

        property_set("wc_transport.clean_up","0");
        if (retval != -1) {
#ifdef BT_SOC_TYPE_ROME
            start_hci_filter();
            if (is_ant_req) {
                ALOGV("connect to ant channel");
                ant_fd = fd = connect_to_local_socket("ant_sock");
            }
            else
#endif
            {
                ALOGV("connect to bt channel");
                vnd_userial.fd = fd = connect_to_local_socket("bt_sock");
            }

            if (fd != -1) {
                ALOGV("%s: received the socket fd: %d is_ant_req: %d\n",
                                                __func__, fd, is_ant_req);
                for (idx=0; idx < CH_MAX; idx++)
                    (*fd_array)[idx] = fd;
                retval = 1;
            }
            else {
                retval = -1;
            }
        } else {
          if (fd >= 0)
            close(fd);
        }
    }
    break;
default:
    ALOGE("Unknown btSocType: 0x%x", btSocType);
    break;
}
}
```

userial_vendor_open

作用：使用给定的配置去打开一个串口，串口的的配置是通过这个函数传递进去的

传入的参数为：数据格式和波特率，结构体参照如下：

```
/*
 typedef struct
 {
    uint16_t fmt;       /* Data format */
    uint8_t  baud;      /* Baud rate */
  } tUSERIAL_CFG;
*/
```

(摘选：hci_uart.c 第226行)

```
/*******************************************************************************
** Function         userial_vendor_open
** Description      Open the serial port with the given configuration
** Returns          device fd
*******************************************************************************/

int userial_vendor_open(tUSERIAL_CFG *p_cfg)
{
    uint32_t baud;                  //波特率
```

```
uint8_t data_bits;    //传输数据位数
uint16_t parity;      //奇偶校验位
uint8_t stop_bits;    //停止位

vnd_userial.fd = -1;

//转换波特率， helper function converts USERIAL baud rates into TCIO conforming baud rates
if (!userial_to_tcio_baud(p_cfg->baud, &baud))
{
    return -1;                  //如果波特率设置不对，会报错！
}

//检测数据格式
if(p_cfg->fmt & USERIAL_DATABITS_8)
    data_bits = CS8;
else if(p_cfg->fmt & USERIAL_DATABITS_7)
    data_bits = CS7;
else if(p_cfg->fmt & USERIAL_DATABITS_6)
    data_bits = CS6;
else if(p_cfg->fmt & USERIAL_DATABITS_5)
    data_bits = CS5;
else
{
    ALOGE("userial vendor open: unsupported data bits");
    return -1;
}

//判断是否有奇偶校验
if(p_cfg->fmt & USERIAL_PARITY_NONE)
    parity = 0;
else if(p_cfg->fmt & USERIAL_PARITY_EVEN)
    parity = PARENB;
else if(p_cfg->fmt & USERIAL_PARITY_ODD)
    parity = (PARENB | PARODD);
else
{
    ALOGE("userial vendor open: unsupported parity bit mode");
    return -1;
}

//判断停止位个数
if(p_cfg->fmt & USERIAL_STOPBITS_1)
    stop_bits = 0;
else if(p_cfg->fmt & USERIAL_STOPBITS_2)
    stop_bits = CSTOPB;
else
{
    ALOGE("userial vendor open: unsupported stop bits");
    return -1;
}

ALOGI("userial vendor open: opening %s", vnd_userial.port_name);

//实现说明一下，vnd_userial变量的声明，包括后面初始化操作都是在本文件中进行的。
//对于详细的方法和过程，你可以查看下一级文件对它的具体说明。
if ((vnd_userial.fd = open(vnd_userial.port_name, O_RDWR|O_NOCTTY)) == -1)
{
    ALOGE("userial vendor open: unable to open %s", vnd_userial.port_name);
    return -1;
}

//清除串口读写缓存区
tcflush(vnd_userial.fd, TCIOFLUSH);

tcgetattr(vnd_userial.fd, &vnd_userial.termios);
cfmakeraw(&vnd_userial.termios);

/* Set UART Control Modes */
vnd_userial.termios.c_cflag |= CLOCAL;
vnd_userial.termios.c_cflag |= (CRTSCTS | stop_bits);

tcsetattr(vnd_userial.fd, TCSANOW, &vnd_userial.termios);

/* Set input/output baudrate */
```

```
    cfsetospeed(&vnd_userial.termios, baud);
    cfsetispeed(&vnd_userial.termios, baud);
    tcsetattr(vnd_userial.fd, TCSANOW, &vnd_userial.termios);

    tcflush(vnd_userial.fd, TCIOFLUSH);

#if (BT_WAKE_VIA_USERIAL_IOCTL==TRUE)
    userial_ioctl_init_bt_wake(vnd_userial.fd);
#endif

    ALOGI("device fd = %d open", vnd_userial.fd);

    return vnd_userial.fd;          //返回一个串口文件描述符
}
```

vnd_userial->port_name

作用：看到这个打卡串口的函数你可能感觉很奇怪，它是根据什么打开特定的串口的
这个函数的成员是在什么时候初始化完成的？以系列的问题有待解决！！！

(摘选：hci_uart.c 第202行)

```
//vnd_userial.port_name这个变量是全局变量，它的初始化是通过下列函数完成的。
/*****************************************************************************
** Function        userial_vendor_init
** Description     Initialize userial vendor-specific control block
** Returns         None
*****************************************************************************/
void userial_vendor_init(void)
{
    vnd_userial.fd = -1;
    snprintf(vnd_userial.port_name, VND_PORT_NAME_MAXLEN, "%s", BT_HS_UART_DEVICE);
}
```

HCI init

作用：在上一条enable里有直接调用hci动态库函数，但是对于动态库中的函数实现一无所知
现在来分析初始化函数究竟做了哪些事情。

(摘选：bt_hci_bdroid.c 第316行)

```
/*****************************************************************************
**   BLUETOOTH HOST/CONTROLLER INTERFACE LIBRARY FUNCTIONS
*****************************************************************************/
static int init(const bt_hc_callbacks_t* p_cb, unsigned char *local_bdaddr)
{
    int result;

    ALOGI("init");

    if (p_cb == NULL)                           //如果指定的结构体为NULL，那么就可以认为无法使用指定的回调函数
    {
        ALOGE("init failed with no user callbacks!");
        return BT_HC_STATUS_FAIL;
    }

    hc_cb.epilog_timer_created = false;         //线程控制块
    fwcfg_acked = false;
    has_cleaned_up = false;

    pthread_mutex_init(&hc_cb.worker_thread_lock, NULL);

    /* store reference to user callbacks */
    bt_hc_cbacks = (bt_hc_callbacks_t *) p_cb;  //将传送过来的指针转化为全局变量，保存回调函数参数

    vendor_open(local_bdaddr);                  //打开厂商提供的静态库，完成初始化操作。

    utils_init();                               //工具集初始化
#ifdef HCI_USE_MCT
    extern tHCI_IF hci_mct_func_table;
    p_hci_if = &hci_mct_func_table;
#else
    extern tHCI_IF hci_h4_func_table;
    p_hci_if = &hci_h4_func_table;
```

```
#endif

    p_hci_if->init();
    userial_init();                                          //串口初始化，这个需要注意一下，在后面会用到的。
    lpm_init();                                  //低功耗LPM初始化
    utils_queue_init(&tx_q);                     //初始化缓存队列

    if (hc_cb.worker_thread)
    {
        ALOGW("init has been called repeatedly without calling cleanup ?");
    }

    // Set prio here and let hci worker thread inherit prio
    // remove once new thread api (thread_set_priority() ?)
    // can switch prio
    raise_priority_a2dp(TASK_HIGH_HCI_WORKER);

    hc_cb.worker_thread = thread_new("bt_hc_worker");
    if (!hc_cb.worker_thread) {
        ALOGE("%s unable to create worker thread.", __func__);
        return BT_HC_STATUS_FAIL;
    }

    return BT_HC_STATUS_SUCCESS;
}
```

vendor_open

作用：主要是打开蓝牙芯片生产商给定的动态库文件。
完成回调函数的挂接

(摘选：vendor.c 第62行)

```
static const char *VENDOR_LIBRARY_NAME = "libbt-vendor.so";                    //芯片生产商给定动态库名称
static const char *VENDOR_LIBRARY_SYMBOL_NAME = "BLUETOOTH_VENDOR_LIB_INTERFACE";

bool vendor_open(const uint8_t *local_bdaddr) {
  assert(lib_handle == NULL);

  //打开动态库
  lib_handle = dlopen(VENDOR_LIBRARY_NAME, RTLD_NOW);
  if (!lib_handle) {
    ALOGE("%s unable to open %s: %s", __func__, VENDOR_LIBRARY_NAME, dlerror());
    goto error;
  }

  //加载动态库
  vendor_interface = (bt_vendor_interface_t *)dlsym(lib_handle, VENDOR_LIBRARY_SYMBOL_NAME);
  if (!vendor_interface) {
    ALOGE("%s unable to find symbol %s in %s: %s", __func__, VENDOR_LIBRARY_SYMBOL_NAME, VENDOR_LIBRARY_NAME, dlerror());
    goto error;
  }

  //初始化厂商提供的库，这里主要是实现函数回调函数的挂接
  int status = vendor_interface->init(&vendor_callbacks, (unsigned char *)local_bdaddr);
  if (status) {
    ALOGE("%s unable to initialize vendor library: %d", __func__, status);
    goto error;
  }

  return true;

error:;
  vendor_interface = NULL;
  if (lib_handle)
    dlclose(lib_handle);
  lib_handle = NULL;
  return false;
}
```

p_hci_if->init

作用：根据条件编译挂接不同的回调函数

(摘选：bt_hci_bdroid.c 第340行)

```
#ifdef HCI_USE_MCT
    extern tHCI_IF hci_mct_func_table;
    p_hci_if = &hci_mct_func_table;
#else
    extern tHCI_IF hci_h4_func_table;
    p_hci_if = &hci_h4_func_table;
#endif

    p_hci_if->init();
```

hci_h4_func_table

作用：完成回调函数的挂接

(摘选：hci_h4.c 第1053行)

```
const tHCI_IF hci_h4_func_table =
{
    hci_h4_init,                //在结构体中这部分挂接在init函数指针上，函数具体实现的内容参照如下！初始化H4模块
    hci_h4_cleanup,
    hci_h4_send_msg,
    hci_h4_send_int_cmd,
    hci_h4_get_acl_data_length,
    hci_h4_receive_msg
};

//--------------------------------------------------------------------------

//下面是hci_h4_init完成初始化所完成的内容。实际上初始化的时候就是完成这样的
//结构体
typedef struct
{
    HC_BT_HDR *p_rcv_msg;          /* Buffer to hold current rx HCI message */
    uint16_t rcv_len;               /* Size of current incoming message */
    uint8_t rcv_msg_type;           /* Current incoming message type */
    tHCI_H4_RCV_STATE rcv_state;    /* Receive state of current rx message */
    uint16_t hc_acl_data_size;      /* Controller's max ACL data length */
    uint16_t hc_ble_acl_data_size;  /* Controller's max BLE ACL data length */
    BUFFER_Q acl_rx_q;      /* Queue of base buffers for fragmented ACL pkts */
    uint8_t preload_count;          /* Count numbers of preload bytes */
    uint8_t preload_buffer[6];      /* HCI_ACL_PREAMBLE_SIZE + 2 */
    int int_cmd_rsp_pending;        /* Num of internal cmds pending for ack */
    uint8_t int_cmd_rd_idx;         /* Read index of int_cmd_opcode queue */
    uint8_t int_cmd_wrt_idx;        /* Write index of int_cmd_opcode queue */
    tINT_CMD_Q int_cmd[INT_CMD_PKT_MAX_COUNT]; /* FIFO queue */
} tHCI_H4_CB;
```

init()

作用：

(摘选：)

```
/*******************************************************************************
** Function         hci_h4_init
** Description      Initialize H4 module 初始化H4模块
** Returns          None
*******************************************************************************/
void hci_h4_init(void)
{
    HCIDBG("hci_h4_init");

    memset(&h4_cb, 0, sizeof(tHCI_H4_CB)); //这部分主要清空h4_cb类型的结构体
    utils_queue_init(&(h4_cb.acl_rx_q));

    /* Per HCI spec., always starts with 1 */
    num_hci_cmd_pkts = 1;

    /* Give an initial values of Host Controller's ACL data packet length
     * Will update with an internal HCI(_LE)_Read_Buffer_Size request
     */
    h4_cb.hc_acl_data_size = 1021;
    h4_cb.hc_ble_acl_data_size = 27;
}
```

lpm_init

作用：初始化低功耗模式

(摘选：lpm.c 第250行)

```
/*******************************************************************************
** Function         lpm_init
** Description      Init LPM
** Returns          None
*******************************************************************************/
void lpm_init(void)
{
    memset(&bt_lpm_cb, 0, sizeof(bt_lpm_cb_t));

    //这里发送指令让蓝牙模块工作于低功耗模式
    vendor_send_command(BT_VND_OP_GET_LPM_IDLE_TIMEOUT, &bt_lpm_cb.timeout_ms);
}
```

bluetoothoppLauncherActivity

说明：当在文件端发送文件时，发送端先来到这里。这里只是提取文件的中转站，主要分为两个分支。

(摘选：bluetoothoppLauncherActivity.java 第93行)

```
if (action.equals(Intent.ACTION_SEND) || action.equals(Intent.ACTION_SEND_MULTIPLE)) {
    //Check if Bluetooth is available in the beginning instead of at the end
     //通过isBluetoothAllowed()是否处于飞行模式,
    if (!isBluetoothAllowed()) {
        Intent in = new Intent(this, BluetoothOppBtErrorActivity.class);
        in.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        in.putExtra("title", this.getString(R.string.airplane_error_title));
        in.putExtra("content", this.getString(R.string.airplane_error_msg));
        startActivity(in);
        finish();
        return;
    }

    /*
     * Other application is trying to share a file via Bluetooth,
     * probably Pictures, videos, or vCards. The Intent should contain
     * an EXTRA_STREAM with the data to attach.
     */
    if (action.equals(Intent.ACTION_SEND)) {
        // TODO: handle type == null case
        final String type = intent.getType();
        final Uri stream = (Uri)intent.getParcelableExtra(Intent.EXTRA_STREAM);
        CharSequence extra_text = intent.getCharSequenceExtra(Intent.EXTRA_TEXT);
        // If we get ACTION_SEND intent with EXTRA_STREAM, we'll use the
        // uri data;
        // If we get ACTION_SEND intent without EXTRA_STREAM, but with
        // EXTRA_TEXT, we will try send this TEXT out; Currently in
// Browser, share one link goes to this case;
if (stream != null && type != null) {
    if (V) Log.v(TAG, "Get ACTION_SEND intent: Uri = " + stream + "; mimetype = "
                + type);
    // Save type/stream, will be used when adding transfer
    // session to DB.
    Thread t = new Thread(new Runnable() {
        public void run() {
            BluetoothOppManager.getInstance(BluetoothOppLauncherActivity.this)
                .saveSendingFileInfo(type,stream.toString(), false);
        }
    });
    t.start();
    //Done getting file info..Launch device picker and finish this activity
      //launchDevicePicker();里面同样会判断蓝牙是否已经打开
    launchDevicePicker();
    finish();
    return;
} else if (extra_text != null && type != null) {
    if (V) Log.v(TAG, "Get ACTION_SEND intent with Extra_text = "
                + extra_text.toString() + "; mimetype = " + type);
    final Uri fileUri = creatFileForSharedContent(this, extra_text);
    if (fileUri != null) {
```

```
                Thread t = new Thread(new Runnable() {
                    public void run() {
                        BluetoothOppManager.getInstance(BluetoothOppLauncherActivity.this)
                            .saveSendingFileInfo(type,fileUri.toString(), false);
                    }
                });
                t.start();
                //Done getting file info..Launch device picker
                //and finish this activity
                launchDevicePicker();
                finish();
                return;
            } else {
                Log.w(TAG,"Error trying to do set text...File not created!");
                finish();
                return;
            }
        } else {
            Log.e(TAG, "type is null; or sending file URI is null");
            finish();
            return;
        }
    } else if (action.equals(Intent.ACTION_SEND_MULTIPLE)) {
        final String mimeType = intent.getType();
        final ArrayList<Uri> uris = intent.getParcelableArrayListExtra(Intent.EXTRA_STREAM);
        if (mimeType != null && uris != null) {
            if (V) Log.v(TAG, "Get ACTION_SHARE_MULTIPLE intent: uris " + uris + "\n Type= "
                    + mimeType);
            Thread t = new Thread(new Runnable() {
                public void run() {
                    BluetoothOppManager.getInstance(BluetoothOppLauncherActivity.this)
                        .saveSendingFileInfo(mimeType,uris, false);
                    //Done getting file info
                }
            });
            t.start();
            //Launch device picker after thread is started to avoid delay
            //caused by saving file information during multiple file share scenarios
            //which may cause ANR.
            launchDevicePicker();
            finish();
            return;
        } else {
            Log.e(TAG, "type is null; or sending files URIs are null");
            finish();
            return;
        }
    }
}
```

launchDevicePicker

作用：在这个函数里又询问了是否在飞行模式

(摘选：bluetoothopplauncheractivity.java 第205行)

```
/**
 * Turns on Bluetooth if not already on, or launches device picker if Bluetooth is on
 * @return
 */
private final void launchDevicePicker() {
    // TODO: In the future, we may send intent to DevicePickerActivity
    // directly,
    // and let DevicePickerActivity to handle Bluetooth Enable.
    if (!BluetoothOppManager.getInstance(this).isEnabled()) {
        if (V) Log.v(TAG, "Prepare Enable BT!! ");
        Intent in = new Intent(this, BluetoothOppBtEnableActivity.class);
        in.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        startActivity(in);
    } else {
        if (V) Log.v(TAG, "BT already enabled!! ");
        Intent in1 = new Intent(BluetoothDevicePicker.ACTION_LAUNCH);
        in1.setFlags(Intent.FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS);
        in1.putExtra(BluetoothDevicePicker.EXTRA_NEED_AUTH, false);
        in1.putExtra(BluetoothDevicePicker.EXTRA_FILTER_TYPE,
```

```
                BluetoothDevicePicker.FILTER_TYPE_TRANSFER);
            in1.putExtra(BluetoothDevicePicker.EXTRA_LAUNCH_PACKAGE,
                    Constants.THIS_PACKAGE_NAME);
            in1.putExtra(BluetoothDevicePicker.EXTRA_LAUNCH_CLASS,
                    BluetoothOppReceiver.class.getName());
            if (V) {Log.d(TAG,"Launching " +BluetoothDevicePicker.ACTION_LAUNCH );}
            startActivity(in1);
        }
    }
```

startLeScan

作用：

(摘选：bluetoothadapter.java 第1739行)

```
public boolean startLeScan(LeScanCallback callback) {
        return startLeScan(null, callback);
    }

    /**
     * Starts a scan for Bluetooth LE devices, looking for devices that
     * advertise given services.
     *
     * <p>Devices which advertise all specified services are reported using the
     * {@link LeScanCallback#onLeScan} callback.
     *
     * <p>Requires {@link android.Manifest.permission#BLUETOOTH_ADMIN} permission.
     *
     * @param serviceUuids Array of services to look for
     * @param callback the callback LE scan results are delivered
     * @return true, if the scan was started successfully
     * @deprecated use {@link BluetoothLeScanner#startScan(List, ScanSettings, ScanCallback)}
     *             instead.
     */
    @Deprecated
    public boolean startLeScan(final UUID[] serviceUuids, final LeScanCallback callback) {
        if (DBG) Log.d(TAG, "startLeScan(): " + serviceUuids);
        if (callback == null) {
            if (DBG) Log.e(TAG, "startLeScan: null callback");
            return false;
        }
        BluetoothLeScanner scanner = getBluetoothLeScanner();
        if (scanner == null) {
            if (DBG) Log.e(TAG, "startLeScan: cannot get BluetoothLeScanner");
            return false;
        }

        synchronized(mLeScanClients) {
            if (mLeScanClients.containsKey(callback)) {
                if (DBG) Log.e(TAG, "LE Scan has already started");
                return false;
            }

            try {
                IBluetoothGatt iGatt = mManagerService.getBluetoothGatt();
                if (iGatt == null) {
                    // BLE is not supported
                    return false;
                }

                ScanCallback scanCallback = new ScanCallback() {
                    @Override
                    public void onScanResult(int callbackType, ScanResult result) {
                        if (callbackType != ScanSettings.CALLBACK_TYPE_ALL_MATCHES) {
                            // Should not happen.
                            Log.e(TAG, "LE Scan has already started");
                            return;
                        }
                        ScanRecord scanRecord = result.getScanRecord();
                        if (scanRecord == null) {
                            return;
                        }
                        if (serviceUuids != null) {
                            List<ParcelUuid> uuids = new ArrayList<ParcelUuid>();
```

```
                for (UUID uuid : serviceUuids) {
                    uuids.add(new ParcelUuid(uuid));
                }
                List<ParcelUuid> scanServiceUuids = scanRecord.getServiceUuids();
                if (scanServiceUuids == null || !scanServiceUuids.containsAll(uuids)) {
                    if (DBG) Log.d(TAG, "uuids does not match");
                    return;
                }
            }
            callback.onLeScan(result.getDevice(), result.getRssi(),
                    scanRecord.getBytes());
        }
};
ScanSettings settings = new ScanSettings.Builder()
    .setCallbackType(ScanSettings.CALLBACK_TYPE_ALL_MATCHES)
    .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY).build();

List<ScanFilter> filters = new ArrayList<ScanFilter>();
if (serviceUuids != null && serviceUuids.length > 0) {
    // Note scan filter does not support matching an UUID array so we put one
    // UUID to hardware and match the whole array in callback.
    ScanFilter filter = new ScanFilter.Builder().setServiceUuid(
            new ParcelUuid(serviceUuids[0])).build();
    filters.add(filter);
                }
                scanner.startScan(filters, settings, scanCallback);

                mLeScanClients.put(callback, scanCallback);
                return true;

            } catch (RemoteException e) {
                Log.e(TAG,"",e);
            }
        }
        return false;
    }
```

Bluetooth->finding->conect->Devices

作用：
使用BluetoothAdapter可以通过设备搜索或查询配对设备找到远程Bluetooth设备。
Device discovery（设备搜索）是一个扫描搜索本地已使能Bluetooth设备并且从搜索到的设备请求一些信息的过程（有时候会收到类似“discovering”，“inquiring”或“scanning”）。但是，搜索到的本地Bluetooth设备只有在打开被发现功能后才会响应一个discovery请求，响应的信息包括设备名，类，唯一的MAC地址。发起搜寻的设备可以使用这些信息来初始化跟被发现的设备的连接。

发现设备 -> 连接设备 -> 管理设备 -> 工作策略

Finding Devices

简介：
在搜索设备前，需要事先查询一下我们需要的设备是否已经存在
需要调用的函数为：getBondedDevices()，返回的结果是一个设备集合，bluetoothDevice对象中需要使用MAC地址来初始化一个连接。

1、Querying paired devices

作用：查找配对

```
    private void populatePairedDevices() {
        mPairedDevicesAdapter.clear();
        Set<BluetoothDevice> pairedDevices = mBluetoothAdapter.getBondedDevices();
        for (BluetoothDevice device : pairedDevices) {
            mPairedDevicesAdapter.add(Device.fromBluetoothDevice(device));
        }
    }
```

2、Discovering devices

作用：通过调用startDiscovery()

（摘选:AdapterService.java 第868行）

```
        public boolean startDiscovery() {
```

```
        if (!Utils.checkCaller()) {
            Log.w(TAG, "startDiscovery() - Not allowed for non-active user");
            return false;
        }

        AdapterService service = getService();
        if (service == null) return false;
        return service.startDiscovery();
    }
```

3、Enabling discoverability

Connect Devices

Managing Devices

Working Profiles