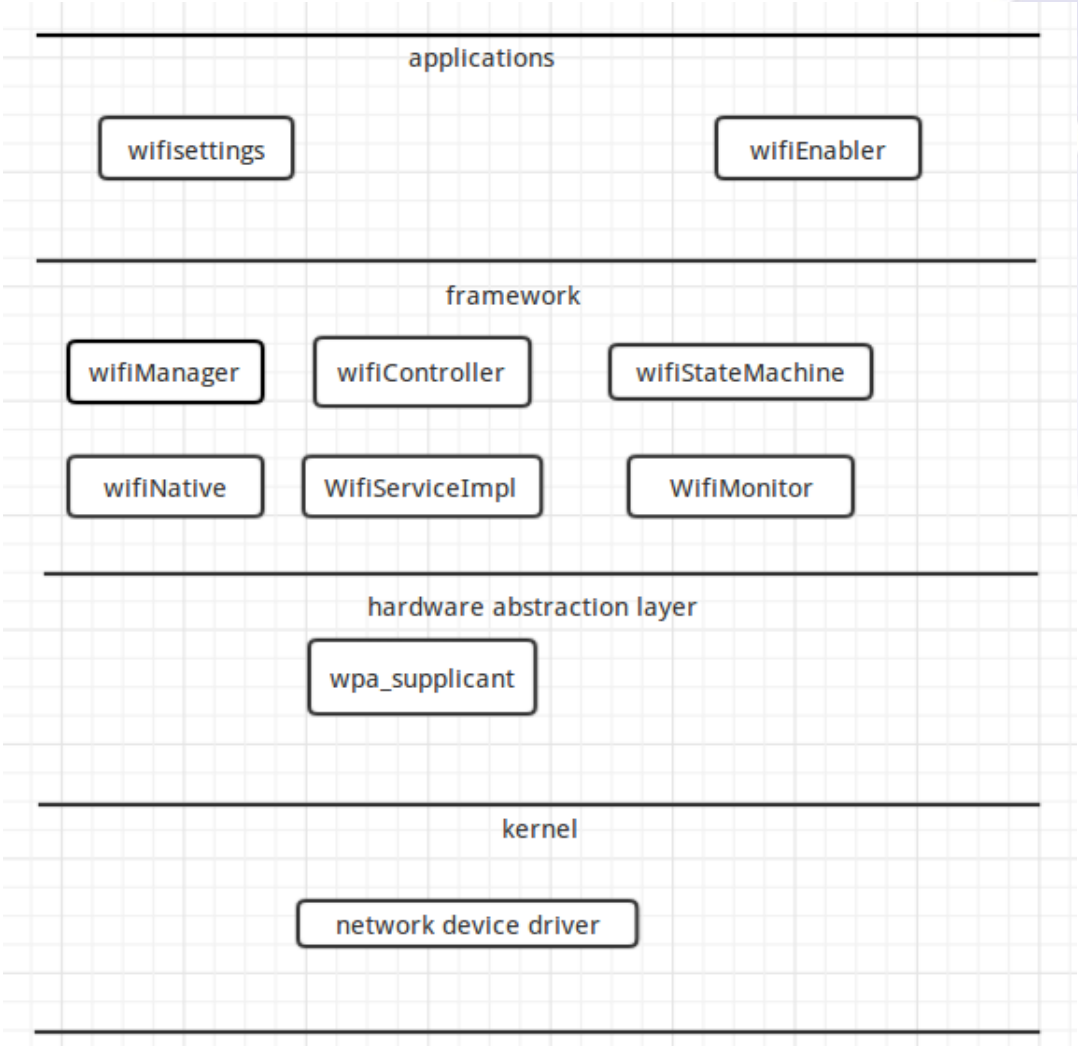
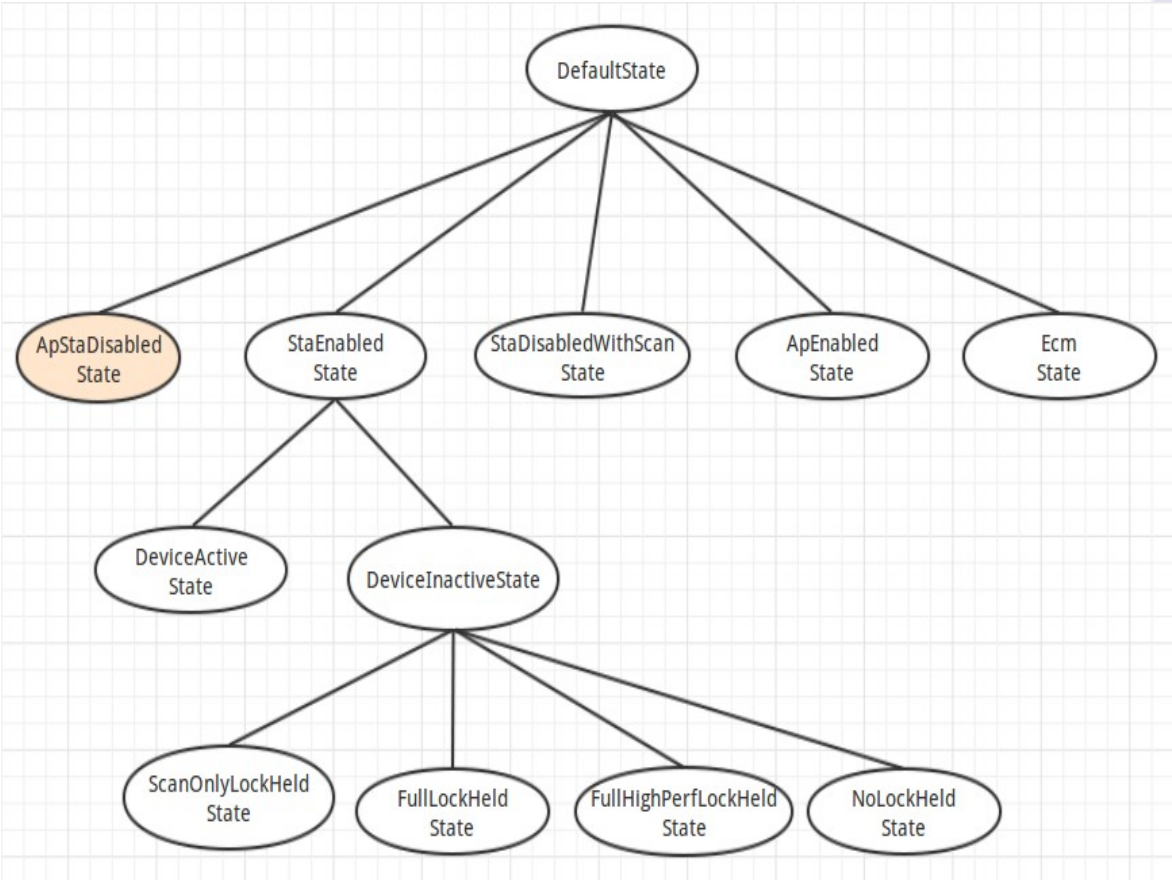
wifi

by 成達玉
darview_cheng@asus.com

总体介绍

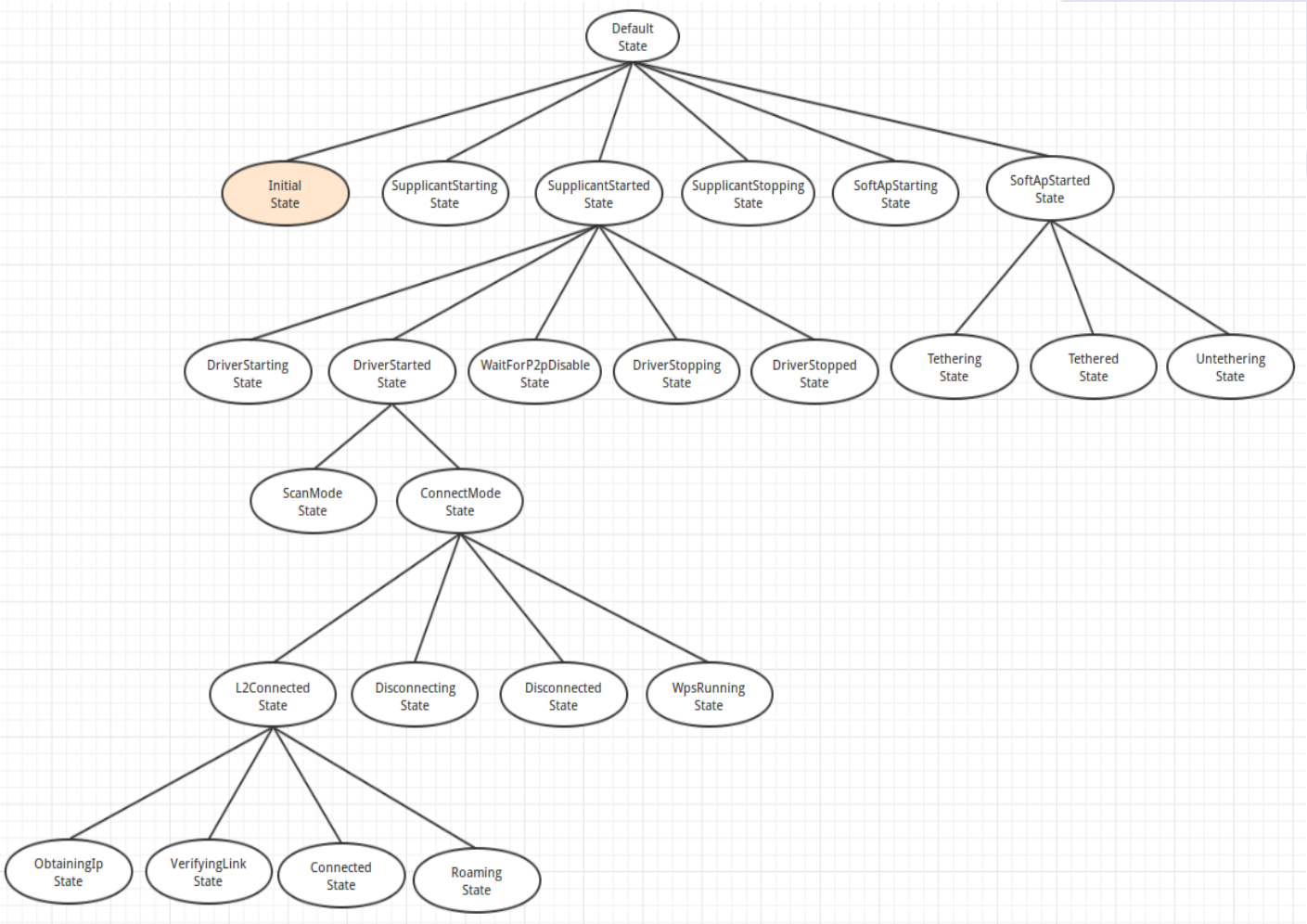


总体介绍



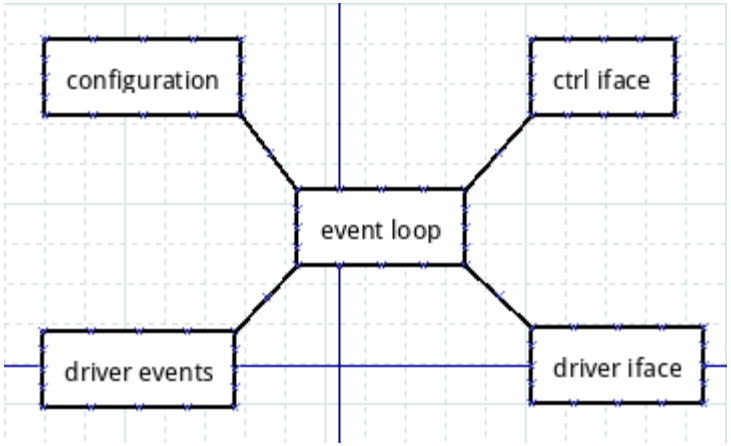
wifiController 是一個裝體機，它的初始狀態是 apStaDisabledState

总体介绍



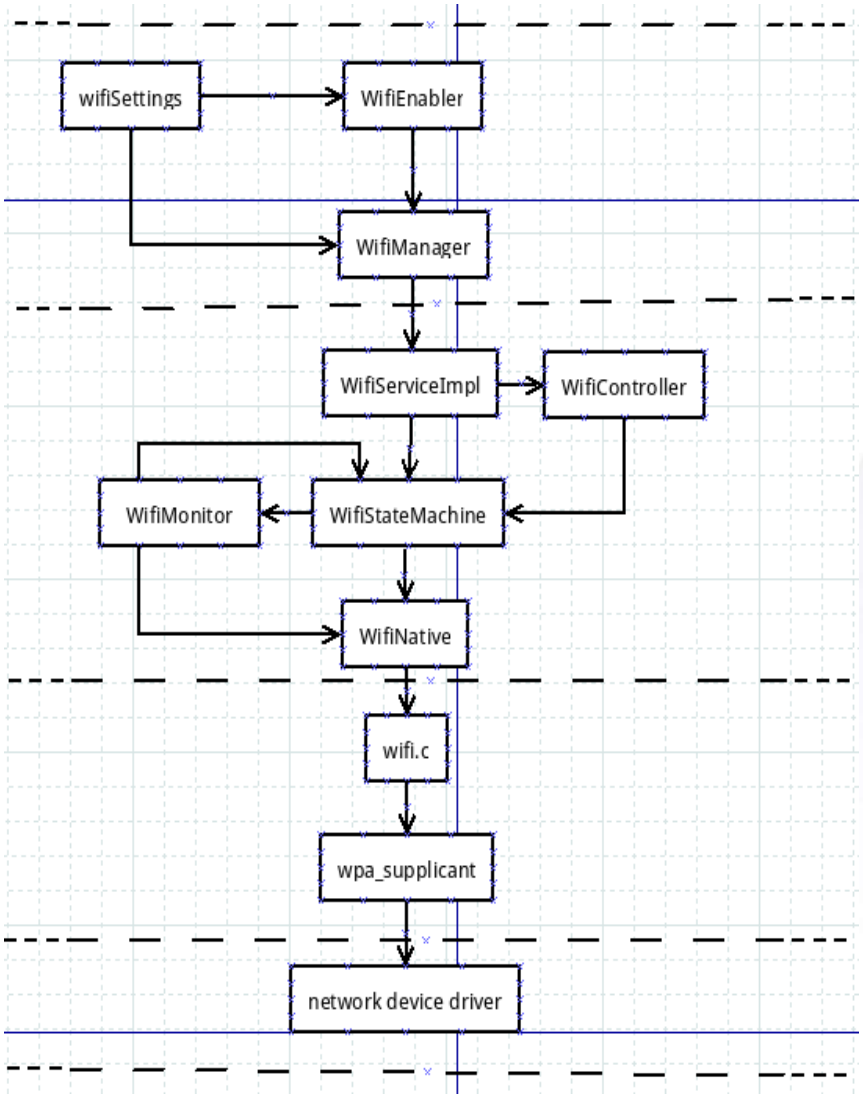
WifiStateMachine 是一個狀態機，它的初始狀態是 initialState

总体介绍



WPAS 所有工作都围绕事件 (event loop 模块) 展开。它是基于事件驱动的
configuration 模块主要完成配置参數的处理
Ctrl iface 模塊主要處理來自 client 端的 control 消息
driver iface 接口模块用于隔离和底层驱动直接交互的那些 driver 控制模块
driver wrapper 经常要返回一些信息给上层 WPAS 中 , 这些信息将通过 driver events 的方式反馈给 WPAS 其他模块进行处理

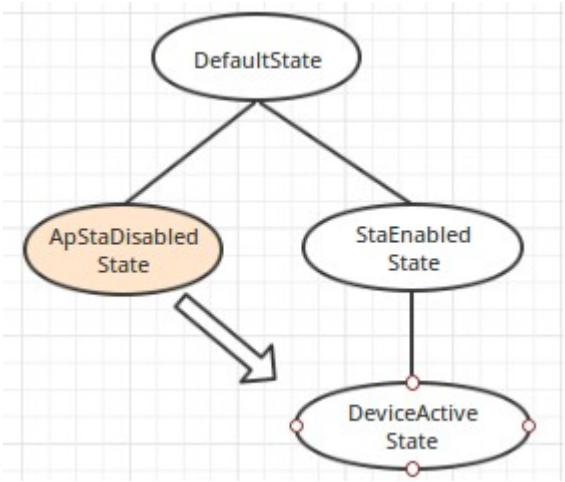
总体介绍

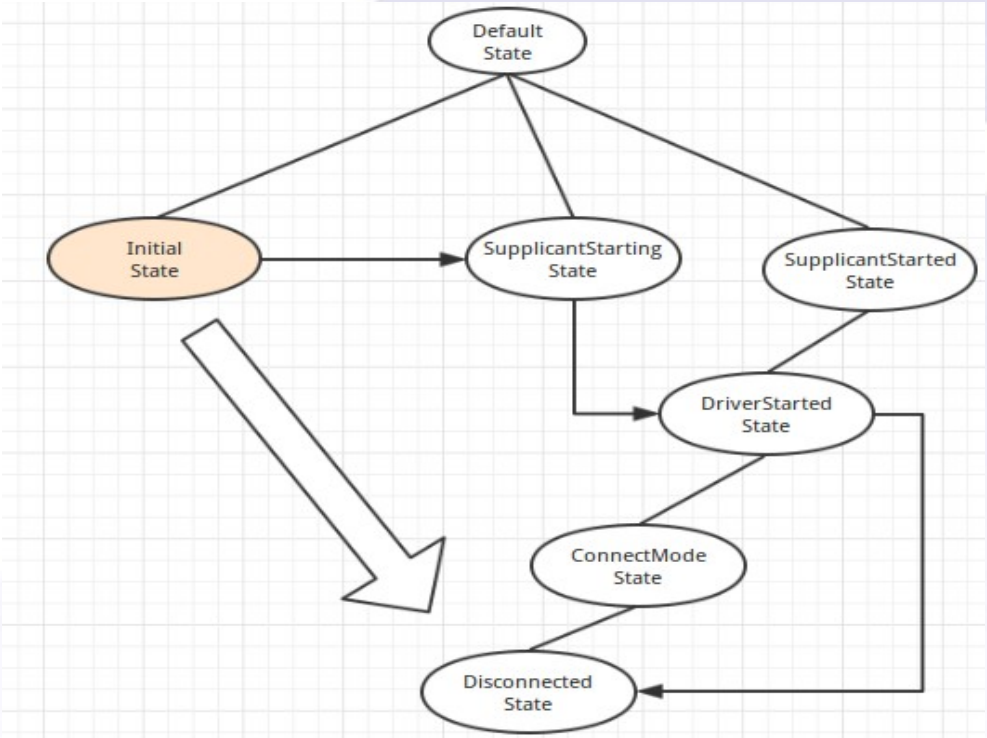


wifiSettings 會引用 wifiEnabler,wifiManager
WifiManager 是 WifiService 的客户端。它通过成员变量 mService 和 WifiService 进行 Binder 交互
WifiServiceImpl 會引用 wifiStateMachine 對 wifi 進行控制
WifiController 引用 wifiStateMachine 對 wifi 進行控制
WifiStateMachine 引用 wifiMonitor 來監控 wpas 事件
WifiMonitor 會將事件分發 , 調用 wifiStateMachine 處理
WifiNative 用于和 WPAS 通信 , 其内部定义了较多的 native 方法
wpa_supplicant 是一个开源软件项目 , 它实现了 Station 对无线网络进行管理和控制的功能

1. 打開 wifi

用戶打開 wifi ，最終將 wificontroller 的狀態轉換到 DeviceActiveState, 將 wifiStateMachine 的狀態轉換到 DisconnectedState





當用戶打開 wifi 開關， WifiEnabler 會通過 wifiManager 調用 wifiServiceImpl 的 setWifiStateEnabled 方法
WifiServiceImpl 的 setWifiStateEnabled 向 wifiController 發送 CMD_WIFI_TOGGLED ，由初始狀態 ApStaDisabledState 處理 , 會將 wifiController 的狀態轉換到 DeviceActiveState
在 WifiController 狀態轉換過程中向 WifiStateMachine 發送了四條消息 *`wificontroller`*
CMD_START_SUPPLICANT-->CMD_SET_OPERATIONAL_MODE-->CMD_START_DRIVER-->CMD_SET_HIGH_PERF_MODE
`InitialState` *處理 CMD_START_SUPPLICANT 時会讓 wifiStateMachine 再轉到 SupplicantStartingState*
SupplicantStartingState 處理 *`SUP_CONNECTION_EVENT`* 時將 *`WifiStateMachine`* 狀態轉換到 *`DriverStartedState`*, *`DriverStartedState`* 會直接將 *`WifiStateMachine`* 的狀態轉換到 *`DisconnectedState`*

1. 打開 wifi

InitialState 處理 CMD_START_SUPPLICANT 時会加載驅動，啓動 wpas, 和 wpas 建立聯繫，向 wifiStateMachine 發送 SUP_CONNECTION_EVENT ，然後啓動一個監聽程序監聽 wpas 事件 , 最後將 WifiStateMachine 的狀態轉換到 SupplicantStartingState

```
case CMD_START_SUPPLICANT:
if (mWifiNative.loadDriver())// 加载 wifi driver
{
        //reload STA firmware
}
if(mWifiNative.startSupplicant(mP2pSupported))// 启动 wpa_supplicant
{               setWifiState(WIFI_STATE_ENABLING);
                mWifiMonitor.startMonitoring();
                // 和 wpa_supplicant 建立 socket 连接并不断的从 wpa_supplicant 收 event
                // 在连接建立之后向 WifiStateMachine  发送 SUP_CONNECTION_EVENT
                // 新建 MonitorThread  不断的从 wpa_supplicant 收 event
                transitionTo(mSupplicantStartingState);
}                                                       貼
```

CMD_SET_OPERATIONAL_MODE-->CMD_START_DRIVER-->CMD_SET_HIGH_PERF_MODE--> SUP_CONNECTION_EVENT

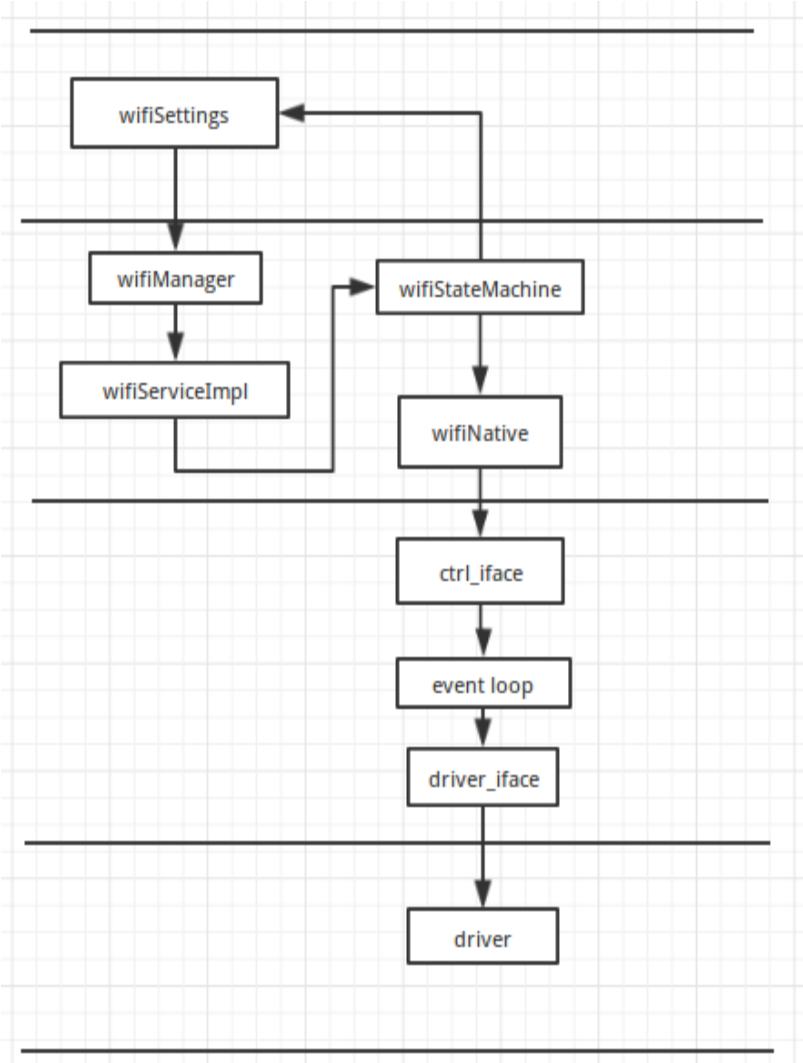
CMD_SET_OPERATIONAL_MODE, CMD_START_DRIVER 被延迟处理，CMD_SET_HIGH_PERF_MODE 会被 DefaultState 处理 SupplicantStartingState 處理 CMD_SET_HIGH_PERF_MODE 時會先發送 WIFI_STATE_CHANGED_ACTION 廣播，然後將 WifiStateMachine 的狀態轉換到 DriverStartedState, 而 DriverStartedState 會直接將 WifiStateMachine 的狀態切到 DisconnectedState

```
case WifiMonitor.SUP_CONNECTION_EVENT:
      setWifiState(WIFI_STATE_ENABLED);
      ….
      InitializeWpsDetails();// 初始化 Wps 信息
      sendSupplicantConnectionChangedBroadcast(true);
      transitionTo(mDriverStartedState);
```

1. 打開 wifi

```
public synchronized void startMonitoring(String iface)
{
        if (mConnected)
        {
              ….
        }
        else
         {
             int connectTries = 0;
             while (true)
             {
                if (mWifiNative.connectToSupplicant()) //和 wpa_supplicant 建立 socket 连接
                {
                    m.mMonitoring = true;
                    m.mStateMachine.sendMessage(SUP_CONNECTION_EVENT);// 向  WifiStateMachine  发送
                                                                      //SUP_CONNECTION_EVENT
                    new MonitorThread(mWifiNative, this).start();// 开启事件监听线程
                    mConnected = true;
                    break;
                 }
             }
        }
}
public void run()
{
        for (;;)
        {
              String eventStr = mWifiNative.waitForEvent();// 接受来自 wpa_s 的事件
              if (mWifiMonitorSingleton.dispatchEvent(eventStr)) // 有 event 才调,, 分发来自 wpa_s 的底层 event
              {
                 break;
              }
      }
}
```

2.setWifiState(WIFI_STATE_ENABLED)



setWifiState(WIFI_STATE_ENABLED); 發送 *WIFI_STATE_CHANGED_ACTION* 廣播

WifiSettings 收到廣播後通過 *WifiManager* 調用 *WifiServiceImpl* 的 *startScan*

WifiServiceImpl 會調用 *wifiStateMachine* 的 *startScan*

wifiStateMachine 发送 *CMD_START_SCAN* 给自己最終由 *DriverStartedState* 處理

DisconnectedState 處理 *CMD_START_SCAN* 返回 *NOT_HANDLED*

```
{
case CMD_START_SCAN:
      判断扫描条件，关闭后台扫描
      交由父状态 (ConnectModeState) 处理
}
```

由于 *ConnectModeState* 无法处理，所以交由祖父状态 *(DriverStartedState)* 处理

```
DriverStartedState
{
   case CMD_START_SCAN:

handleScanRequest(WifiNative.SCAN_WITHOUT_CONNECTION_SETUP,
message);
                        break;
}
```

handleScanRequest 會調用 *wifiNative* 的 *scan* 方法通過 *jni* 將掃描請求發送給 *ctrl_iface*

ctrl_iface 會設置掃描結果處理器，然後最終通過 *event loop* 調用 *driver wrapper* 的 *scan2* 發送掃描命令給 *driver*

3.wpas 向驱动发送扫描命令

```
static void wpas_ctrl_scan(struct wpa_supplicant *wpa_s, char *params,
                    char *reply, int reply_size, int *reply_len)
{
      if (params)
      {
            if (os_strncasecmp(params, "TYPE=ONLY", 9) == 0)
                  {
                  wpa_s->scan_res_handler = scan_only_handler;
                  }...
            }
      ....
      if (!wpa_s->sched_scanning && !wpa_s->scanning &&
                ((wpa_s->wpa_state <= WPA_SCANNING) ||
                 (wpa_s->wpa_state == WPA_COMPLETED)))
      {     ….
            wpa_supplicant_req_scan(wpa_s, 0, 0);
            ….
      }
}
```

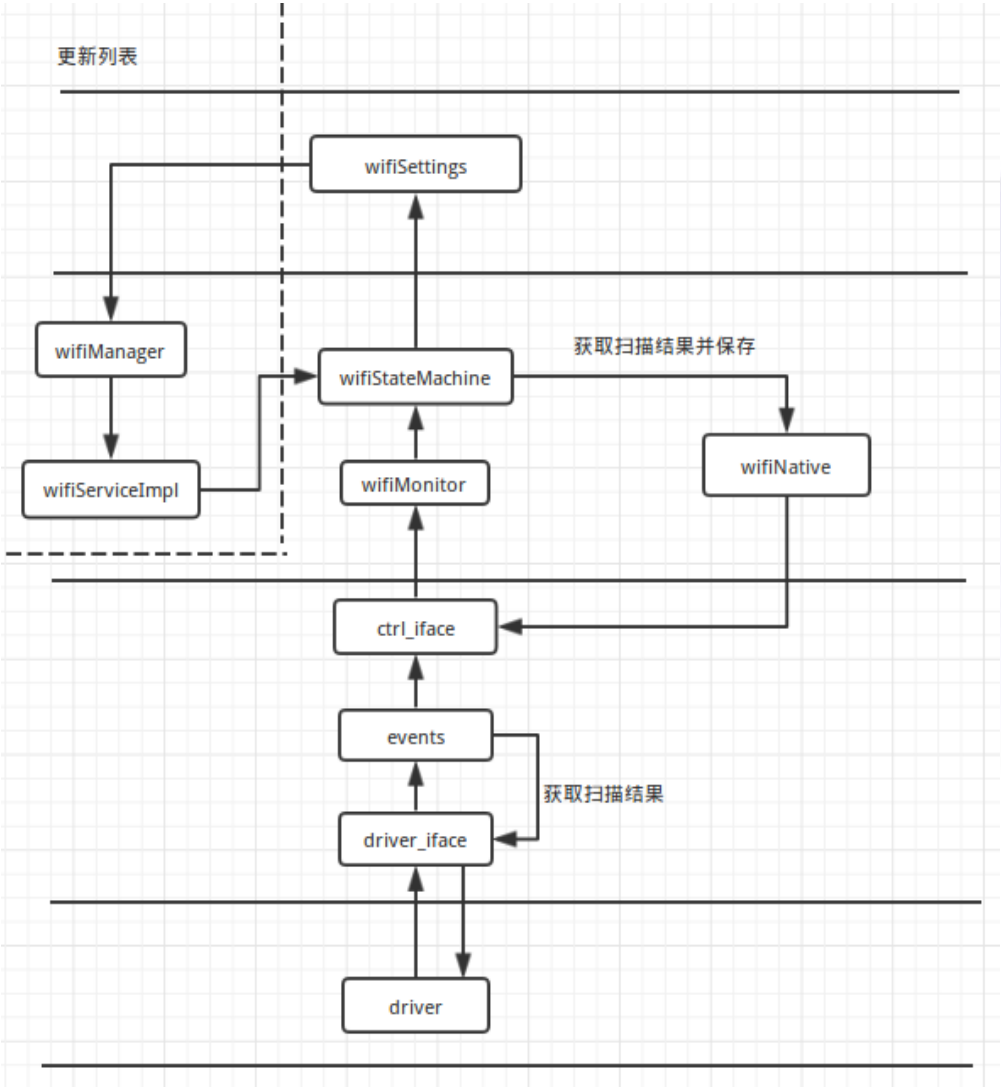
3.wpas 向驱动发送扫描命令

```
void wpa_supplicant_req_scan(struct wpa_supplicant *wpa_s, int sec, int usec)
{
      int res = eloop_deplete_timeout(sec, usec, wpa_supplicant_scan, wpa_s,
                                      NULL);
      if (res == 1) {…}
      ...
      else {
            eloop_register_timeout(sec, usec, wpa_supplicant_scan, wpa_s, NULL);
      }
}
void wpa_supplicant_scan(void *eloop_ctx, void *timeout_ctx)
{
      prev_state = wpa_s->wpa_state;
      if (wpa_s->wpa_state == WPA_DISCONNECTED ||
          wpa_s->wpa_state == WPA_INACTIVE)
            wpa_supplicant_set_state(wpa_s, WPA_SCANNING);
      ….
      ret = wpa_supplicant_trigger_scan(wpa_s, scan_params);
}
```

wpa_supplicant_trigger_scan 最終調用 driver interface 中的 scan2 既 wpa_driver_nl80211_scan
wpa_driver_nl80211_scan 方法調用 send_and_recv_msgs 向 wlan 驱动發送命令

```
int wpa_driver_nl80211_scan(struct i802_bss *bss,
                            struct wpa_driver_scan_params *params)
{
      ret = send_and_recv_msgs(drv, msg, NULL, NULL);
      /*
      发送请求给 wlan 驱动。返回值只是表示该命令是否正确发送给了驱动。扫描结束事件将通过
      driver event 返回给 WPAS 。
      */
}
```

4.wpa 處理掃描結果



driver 掃描完成後發消息給 *driver Wrapper*
wpas 通過 *events* 模塊獲取掃描結果，並向 *wifimonitor* 通知。
wifiMonitor 收到通知，向 *wifiStateMachine* 發消息，*wifiStateMachine* 會向 *wpas* 去获取結果並保存，同时发送结果可用的通知
wifiSettings 收到通知，更新 *ap* 列表。

4.wpa 處理掃描結果

```
 int _wpa_supplicant_event_scan_results(struct wpa_supplicant *wpa_s,
                                        union wpa_event_data *data,
                                        int own_request)
{
      scan_res = wpa_supplicant_get_scan_results(wpa_s,
                                          data ? &data->scan_info :
                                          NULL, 1);
      if (own_request && wpa_s->scan_res_handler &&
          (wpa_s->own_scan_running || !wpa_s->external_scan_running)) {
            void (*scan_res_handler)(struct wpa_supplicant *wpa_s,
                                     struct wpa_scan_results *scan_res);
            scan_res_handler = wpa_s->scan_res_handler;
            wpa_s->scan_res_handler = NULL;
            scan_res_handler(wpa_s, scan_res);// 調用 scan_only_handler(/scan.c)
            ret = -2;
            goto scan_work_done;
      }
scan_work_done:
      wpa_scan_results_free(scan_res);
      if (wpa_s->scan_work) {
            struct wpa_radio_work *work = wpa_s->scan_work;
            wpa_s->scan_work = NULL;
            radio_work_done(work);
      }
      return ret;
}
```

scan_only_handler 通過調用 wpa_msg_ctrl 向 wifiMonitor 發送消息

5.wifiMonitor 處理消息

```
 void handleEvent(int event, String remainder)
{
 case SCAN_RESULTS:
                mStateMachine.sendMessage(SCAN_RESULTS_EVENT);
                break;
}

SupplicantStartedState 從 wpa_s 获取扫描结果并保存，然后发送结果可用的广播
{
 case WifiMonitor.SCAN_RESULTS_EVENT:
                    closeRadioScanStats();
                    noteScanEnd();
                    setScanResults();// 从 wpa_s 获取扫描结果，并保存
                    if (mIsFullScanOngoing) {
                       sendScanResultsAvailableBroadcast();
                    }

                    break;
}
```

Thank You!