Design a WLAN Mini Access Point in the Android Platform

By

Huaqiang He and Xiaowen Guo

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree Master of Science in Information and Communication Technology

Faculty of Engineering and Science

University of Agder

Grimstad

May 2010

Abstract

Mobile as a computing platform is becoming more and more popular. The amount of such devices shipped every year is growing rapidly, more than 1.2 billion in 2009. At the same time the WLAN is being widely adapted at various locations like campuses, meeting rooms, stations, etc. Currently almost all smart phones come with the support for the WLAN. However, most the mobile devices can only behavior as a client in the WLAN. It would be a remarkable feature if the mobile device is able to function as an Access Point (AP) and a modem which forwards data between the 3G network and the WLAN. Android designed for handheld devices has become a popular and powerful platform in both the industry and amateur developer community. Presently there is no WLAN AP mode supported in the Android platform, therefore it's an interesting task for us to implement such a function.

We start with studying the software AP `hostapd`. We set up a WLAN with `hostapd` running in a Ubuntu Linux platform, instead of a hardware AP. By doing this we figure out the elements needed to achieve the software AP functionality. Next we explore the Android building system, understand the mechanism the building system works, and learn the way add new modules that we prepare to add into the platform. With these basics we take all the elements needed into Android source code hierarchy and build them into the final executables. Testing cases are given both in Ubuntu Linux platform and the Android platform. To make the user experience better we design an application in the Android platform for controlling the AP built from `hostapd` and other components.

Through the process we have done many experiments and have gained rich experience and knowledge in the Linux operating system, Linux wireless implementation, wireless drivers, Android building system, and Android application development. Some of them are enhancement to the existing knowledge in various websites, and some are new to all the members in the development community. These are all recorded in the thesis. For the final testing we succeed in both steps. First, the peripheral stations can discover the AP in the Android platform and all stations are able to connect to it. There is no difference between connection to the AP in the Android platform and connection to a normal hardware AP device. Secondly, the data packets are successfully transmitted between stations, which means there is no barrier in the AP in the Android platform for providing data service. From the view of networking layering, we conclude that we succeed in both link layer and application layer.

Preface

This thesis project concludes our two-year master study program in Information and Communication Technology at University of Agder, Grimstad, Norway. The work is organized in the course of IKT 590, with the credit of 30 ECTs. This project is proposed by ST-Ericsson Norway. It began on 1 January 2010, and ended on 25 May 2010.

First, we would like to thank our external supervisors Roger Frøysaa and Andreas Ludviksen in ST-Ericsson, for providing us valuable technical advice and assistance. They constantly direct us the focus of our ongoing work to keep our project on the right track. Also they give us extensive suggestions on the solution of practical problems. Secondly, we would like to thank our project manager Jonny Ervik in ST-Ericsson, for supporting us with comfortable office and required equipments.

Additionally, we would like to thank our internal supervisor Frank Li at University of Agder, for promoting our work forward so as to guarantee our work completed on schedule. Also he gives us valuable comments on thesis composing, which greatly improve our quality of writing.

All in all, we appreciate everyone's positive attitudes and efforts, which make our work process well and fulfilled.

Grimstad, 25 May 2010

Huaqiang He

Xiaowen Guo

Table of Contents

1. Introduction 1

1.1. Background and motivation 1
1.2. Problem statement 1
1.3. Contribution to knowledge 2
1.4. Approach 2
1.5. Thesis structure 3

2. Hostapd in Linux 4

2.1. Software AP in Linux 4
2.2. System architecture 5
2.2.1. Hostapd 6
2.2.1.1. Drivers and WLAN card 6
2.2.1.2. Security mechanisms in hostapd 8
2.2.1.3. Compiling configuration file 9
2.2.1.4. Running configuration file 10
2.3. An example for hostapd building in Ubuntu 9.10 system 10
2.3.1. Compiling hostapd with a minimal configuration 11
2.3.2. Bridge utility 11
2.3.3. Bridge setup 12
2.3.4. Wireless utilities 12
2.3.5. Running hostapd in Ubuntu 13
2.3.5.1. Testing environment 13
2.3.5.2. Configuration file 13
2.3.5.3. Testing 14

3. Android building system 19

3.1. Android source code 19
3.1.1. Tools working on source code 19
3.1.2. Source code branches 19
3.1.3. Downloading the source code 19
3.2. Build environment on Ubuntu platform 21
3.2.1. Ubuntu Linux (32-bit x86) 22
3.3. Cross compiling 23
3.4. Cross compiling tool chains for Android in Ubuntu system 23
3.4.1. Cross compiling ABI 23
3.4.2. Cross linker 24
3.5. Build variables and build process 24
3.5.1. Build for a new product 24
3.5.2. Default settings and build process 25
3.5.3. Adding a new packages for building 25
3.5.3.1. Template makefiles in the building system 25
3.5.3.2. Adding a new module and writing the Android.mk 27
3.6. Out directory 28
3.7. Details about other directories like system and prebuilt 28
3.7.1. external/ directory 28
3.7.2. prebuilt/ directory 28
3.7.3. bootable/ directory 29

4. Hostapd in Android 30

4.1. Configuration for building hostapd 30
4.2. Openssl in Android 31
4.3. Configuration for building libnl 32
4.4. Configuration for building bridge-utils 33
4.5. Wireless card and the driver 35
4.6. Launching hostapd with configuration file 37

5. Testing 39

5.1. Expected results 39
5.2. Testing environment 39
5.3. Network topology 40
5.4. Setting up the network 40
5.4.1. Running hostapd and connecting the hosts 40
5.4.2. Configuration of hosts 42
5.5. Packets transmission 43
5.6. Result analysis 44

6. User interface 45

6.1. Android application framework 45
6.1.1. Four essential components 45
6.1.2. Intent and intent filter 46
6.1.3. Android User interface design 46
6.1.4. The AndroidManifest.xml 47
6.1.5. Develop with Eclipse 48
6.1.6. Important APIs 52
6.2. GUI design through hostapd control interface 53
6.2.1. Hostapd GUI design overview 53
6.2.2. Functionality design diagram 54
6.2.2.1. Hostapd GUI screen design 54
6.2.2.2. Hostapd GUI use case design 56
6.2.2.3. Hostapd control interface 58
6.2.2.4. Hostapd GUI class design 61
6.2.3. Implementation approach 63
6.3. UI design through hostapd configuration file 66
6.3.1. The concept of design 66
6.3.2. Write an application to run hostapd 67
6.3.3. Write an application to edit hostapd configuration file 68
6.3.4. Comparison between two designs 70

7. Contributions 71

8. Discussions 72

8.1. How to run hostapd in the Linux system 72
8.2. How to understand the Android building system 72
8.3. How to port hostapd from Linux to the Android system 73
8.4. How to control hostapd from the application layer 73

9. Conclusions and future work 75

References 76

Appendix A 77

Appendix B Android.mk of hostapd 80

Index of Figures

Figure 1 WLAN with hardware AP 4
Figure 2 WLAN with software AP hostapd 5
Figure 3 Software AP system architecture 5
Figure 4 Hostapd implementation architecture 6
Figure 5 Linux wireless stack 8
Figure 6 Interfaces in up state 14
Figure 7 Addresses configuration of interfaces 16
Figure 8 Hostapd running with its configuration file 16
Figure 9 A station connect to hostapd 17
Figure 10 Hostapd discovered at vista host machine 17
Figure 11 Connection state of vista host machine 18
Figure 12 Address configuration of vista host machine 18
Figure 13 ping google.com from vista host machine 18
Figure 14 donut-x86 branch source code overview at the top 20
Figure 15 Hostapd source code overview at the top 31
Figure 16 successful running of hostapd in Android 31
Figure 17 Source code overview of libnl in the lib subdir 32
Figure 18 Source code overview of bridge utils at the top 34
Figure 19 successful running of bridge utils in Android 35
Figure 20 Modules currently running in Android 37
Figure 21 Hostapd launched with its configuration file in Android 38
Figure 22 Network topology in the testing case of hostapd in Android 40
Figure 23 Eee PC boots from the USB stick and Android is running 41
Figure 24 Hostapd is running and stations are connected 42
Figure 25 Available network interfaces in the Android 42
Figure 26 Network address configuration in Sony Ericson W715 mobile phone 43
Figure 27 ping Android host from Windows vista 43
Figure 28 ping Sony Erricson W715 from Windows vista 44
Figure 29 Android tree-structured UI [14] 47
Figure 30 A simple AndroidManifest.xml file 48
Figure 31 Create an Android project with Eclipse 49
Figure 32 Android project directory structure in Eclipse 49

Figure 33 A simple layout xml file with two elements 50
Figure 34 The screen result of Fig. 33..... 50
Figure 35 An example of Android emulator..... 51
Figure 36 Running result shown in the Android emulator 51
Figure 37 Popup an external window using `AlertDialog` class 52
Figure 38 `Hostapd` GUI main window 54
Figure 39 Network setting window 55
Figure 40 Authentication window..... 55
Figure 41 `Hostapd` GUI adding window 56
Figure 42 List view of currently connecting with the AP 56
Figure 43 Use case diagram for the AP..... 57
Figure 44 Use case diagram for `hostapd` GUI 58
Figure 45 Sequence diagram for `hostapd` control interface 60
Figure 46 Class diagram for `hostapd` GUI..... 62
Figure 47 JNI call process 64
Figure 48 Steps of writing and running `HelloWorld` program [10] 65
Figure 49 Result of running a command 68
Figure 50 Running result for editing conf file 69

Index of tables

Table 1 Wireless card supported by hostapd 7
Table 2 Elements and functions in setting window 62
Table 3 Elements and functions in adding window 63
Table 4 Elements and functions in main window 63
Table 5 Methods and method description in class Process 77
Table 6 Methods and method description in class Runtime 77
Table 7 Description of method getAssets() 77
Table 8 Methods and method description in class InputStream 78
Table 9 Description of method setHorizontallyScrolling() 78
Table 10 Description of method setOnClickListener() 78
Table 11 Description of method openFileOutput() 79

1. Introduction

1.1. Background and motivation

Mobile as a computing platform is becoming more and more popular. The amount of such devices shipped every year is growing rapidly, more than 1.2 billion in 2009. At the same time WLAN is being widely adapted at various locations like campuses, meeting rooms, stations, etc. due to its features of low cost, fast deployment, and high data transmission efficiency. For now almost all smart phones come with the support for WLAN. However, most mobile devices can only behavior as a client in the WLAN. This means that such devices can just ask for data service via the AP, but cannot provide data services for other devices that have WLAN access capability built in.

It would be a remarkable feature if the mobile device is able to function as an AP and a modem which forwards data between the 3G network and the WLAN. Then a lot of fantastic things would become possible and easy. For example, a WLAN can be setup at any places where the 3G network is accessible and a bunch of other devices with WLAN support can connect to the mobile device and enjoy the Internet. Besides, if the WLAN is already available the mobile device can also connect to the AP and access the Internet. For future applications, the feature of AP capability in the mobile device can be used to set up P2P networks in which the device at a better position to provide the Internet service can take over the task of being an AP in the network.

Android designed for handheld devices has become a popular and powerful platform in both the industry and amateur developer community. Presently there is no WLAN AP mode supported in the Android platform, therefore it's an interesting task for us to implement such a function at the challenging time when there is not much documentation we can refer to. This project was initially proposed by Ericsson Company for commercial use in the smart phone based on the Android platform. The success of this Master thesis project provides useful implementation information for developers of the mobile product.

1.2. Problem statement

The main objective of this thesis project is to design and implement a function so that an ordinary station can act as an AP in a wireless network and this design must be based on the Android platform. The project includes four main aspects where our efforts should be put: AP, the Android platform, the association between them and GUI for the AP. More specifically, the following problems need to be solved:

How does an AP work in Linux circumstance?

In this project, we choose `hostapd` as our logical AP, which is not a physical device but a user space daemon for AP. As an initial step towards the goal, our first concentration is put under Linux circumstance. We need to figure out what pre-conditions is required by `hostapd` in order to work in Linux circumstance, including both hardware and software requirements. And what functions those dependencies of `hostapd` support. We also need to know how they work interdependently with `hostapd`. In order to run `hostapd` in Linux system, we need to configure and compile it for the host environment. To test `hostapd` AP functionality, we need to create a bridge which can transfer package between wired and wireless network.

How is the Android building system constructed?

Due to the fact that the C libraries in Android is different from that in Linux, all shared libraries and utilities need to be recompiled for the support of `hostapd` when porting it to the Android system. Thus, to understand and analyze how the Android building system is constructed and to be able to add new blocks into the Android system is a crucial part of our work. This includes Android source code structure, build variables, build process, etc. In the building process, we are going to build up both the Android platform and various applications which make the process even more complicated. In addition, we need to write our own makefiles for each application to be built up.

How does hostapd work in the Android environment?

After mastering how to add external packages for building, we need to put `hostapd` together with its dependencies solved in the first problem into Android. This includes specific configuration, such as target, compiler, linker, etc. And also we need to configure for Linux kernel. The most complex part in this phase is that the interdependency between `hostapd` and those elements required for the support of `hostapd` takes time to explore. We must also consider the conflict of two or more drivers resulting from their existing driver supports in the Android system.

How to control and configure hostapd from the Android application layer?

In order to control the `hostapd` service, a starting point is wpa_supplicant whose interface exists in Android. Hence it is possible that we can in a similar way access the control interface of `hostapd` from Android. Then first it is necessary to find the vertical call chain of wpa_supplicant. We also need to consider a bridge that can provide connection from java application to call a native C library. Thus, a possible way is to create an Android GUI application to control the `hostapd` that uses the call chain down to the `hostapd` process down in the Linux environment. The application to start/stop `hostapd` service could be feasible, and later it can be extended with more specific functionalities.

Moreover, other activities such as testing of the implemented function will be also performed in our project.

1.3. Contribution to knowledge

In this thesis, our task is to achieve the goal that an ordinary laptop based on the Android system can function as an AP for wireless access. We have not only proven the feasibility of implementing `hostapd` into the Android operating system but also introduced a completed process to realize this goal, including build-up from the scratch and the controlled behavior from the top application. In addition, we have presented extensive experience of the Android building system and putting `hostapd` in Android, which can be shared by other developers in the Android community.

The task we face in this thesis work is challenging and our achievement is innovative due to two reasons: the first one is that so far nobody else has implemented this function; and the other is there is little previous work that can be referred since Android is developed very recently. Furthermore, this work should also be possibly compatible with Android phones, if the hardware configuration requirements are satisfied. Hence we hope our work can provide guidelines for Android developers and in the future there will be Android phones available as an AP to facilitate the mobile users. All in all we hope our efforts will lead to an increased number of R&D efforts on this topic.

1.4. Approach

To reach the project goal, we have adopted the following steps as our approach in this thesis work:

- To accomplish the objective of this thesis we first choose a WLAN card that supports AP function and find the card driver that is supported by `hostapd`. Then we build up `hostapd`,

change the configuration file for it, and we can get `hostapd` running with this AP compatible WLAN driver. We use a laptop installed `hostapd` as an AP to test, it providing the wireless network service.

- Next we work on the Android building system as it is essential for the following steps. We learn how the source codes are structured in Android directories, how to set up the build variables, how the build process is organized, how to add external packages, etc.
- Based on the knowledge we have learned about Android building system, porting `hostapd` from Linux system to Android circumstance is processed. In this phase, we configure the Linux kernel for the support of `hostapd`, build `hostapd` and its dependencies in term of software into the Android roots, set up configuration files for `hostapd`, etc. Besides, we solve the conflict problem of two existing drivers, which arises when we add our own WLAN driver to Android.
- In order to control `hostapd` behavior from Android application layer, we finally provide two alternatives to increase the user experience. One way is to design an Android GUI application to control `hostapd` using its control interface. It provides other than the basic start-and-stop service, and also some extended functionalities. The method we use is to refer to the existing wpa_supplicant interface in Android, so that in a similar way we can access the control interface of `hostapd` from Android. The second alternative is to create an Android application that can run `hostapd` application as a whole instead of talking with its control interface.

1.5. Thesis structure

To have a clear statement of the whole work, we organize the Master thesis report into 9 chapters. The focus of each chapter is given as follows in brief statement.

- Chapter 1 introduces the background, motivation, goal of the Master thesis, as well as the approaches used in the thesis project.
- Chapter 2 shows the procedure of setting up a software AP in Ubuntu Linux platform. It gives clear indication about what elements are needed to build up the AP, which is the reference for the work in the Android platform
- Chapter 3 discusses the Android building system which we must first understand before diving into the modification of the system. It explains the various aspects of the building system and the building process.
- Chapter 4 demonstrates the work done in the Android platform to build up the software AP, which is the central goal of the Master thesis.
- Chapter 5 shows the testing case which proves the functionality achievement of the software AP in the Android platform.
- Chapter 6 is mainly about the design of application in the Android platform. The design is to build up graphic user interfaces of software AP.
- Chapter 7 shows the contribution of this Master thesis.
- Chapter 8 states some of the main issues we encounter during the Master thesis together with how we solve them and suggestions for other developers.
- Chapter 9 concludes the work of the Master thesis and suggests the future work.

2. Hostapd in Linux

`Hostapd` is first developed under Linux system. It is guaranteed that `hostapd` can fully function under Linux circumstance. The Android platform can be considered as a Linux kernel plus a virtual machine on top which Java application is running. Therefore, first having a look at `hostapd` working in Linux is a good reference for the following work. The binary `hostapd` in Linux should have small difference from the binary `hostapd` in Android for the reason that the C libraries are different at some points. The C library in Android is a simplified version of that in Linux. Since C library is fundamental to applications, all shared libraries and utilities need to be recompiled for the support of `hostapd` in Android. The recompiling will be given in chapter 4. But before that section 2.3 will give an example of compiling under Ubuntu Linux. With the results shown, it proves that `hostapd` can properly working in Linux circumstance.

2.1. Software AP in Linux

In a wireless network there are two types of devices, an AP and peripheral STAtions (STAs). The AP provides access services for all STAs via shared wireless media. STAs sharing the same media compete in each time slot for the right of data transferring. Recently wireless network access service has been expanded for multiple sorts of devices, far more beyond laptops. In most cases the laptops, desktops, and other computing devices with a WLAN card are only functioning as an STA, while the AP is a separate device bought from some vender. However, with the open source movement there comes the possibility that ordinary customs can build their own wireless network without buying such an AP device.

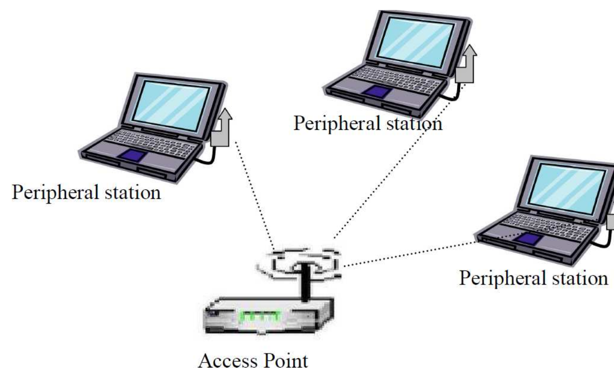

Figure 1 WLAN with hardware AP

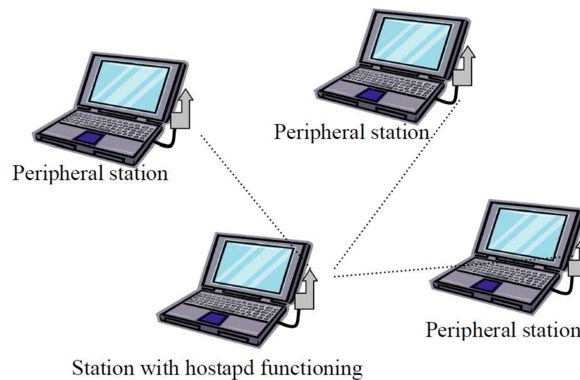

Figure 2 WLAN with software AP `hostapd`

In the open source software community there is an application called `hostapd`. It can fully function just like a separate AP device.

2.2. System architecture

The host machine running `hostapd` has no special hardware requirement provided it has a WLAN card supported by `hostapd`. But some software utilities are necessary for `hostapd` to function in the Linux operating system. These utilities include `openssl`, `libnl`, and the WLAN card driver.

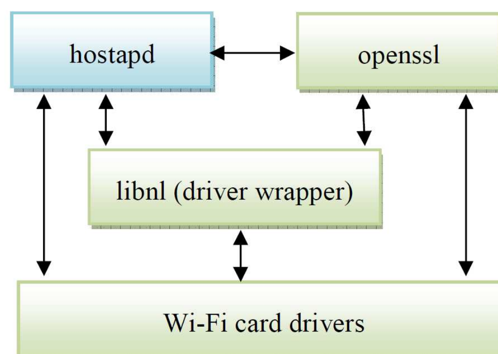

Figure 3 Software AP system architecture

Wireless network has different security considerations from wired network. In wired network the information carried over network media is encapsulated in the wire itself, therefore any device without physical connection to the wire cannot eavesdrop the meaningful content. The user information is secured by the physical connection itself. However in the wireless channel any device with the radio receiver can sense the signal. Such characteristic of wireless channel requires the data transmitted to be encrypted and only the two participants in communication should hold the key. Furthermore, the claimed participant must be authenticated before the connection for data transmission is established. Authentication and encryption are two fundamental security requirements in wireless communication. The security services are provided by `openssl` in Figure 3. `Openssl` provides methods for `hostapd` to negotiate security parameters with peripheral stations when they try to make a connection.

`Libnl` is a wrapper of WLAN card driver. There are many different kinds of WLAN cards can be used when running `hostapd`. WLAN cards from different venders often have different drivers. `Libnl` is built

to encapsulate the differences of implementation of these drivers. However `libnl` only abstracts some of the drivers supported by `hostapd`, therefore `hostapd` needs to implement wrappers for the rest drivers. Both `hostapd` and `openssl` can interact with `libnl`. `Libnl` then communicate with the WLAN card driver to deliver the outgoing and incoming data.

A WLAN driver directly operates on the WLAN card. It is the proxy for all upper layer operations and provides atomic operations and functionalities that can be organized into tasks. When the driver's implementation doesn't support `libnl` it communicates directly with `hostapd` and `openssl`, otherwise it interacts with `libnl`.

2.2.1. Hostapd

`Hostapd` is a user space daemon for AP and authentication servers in WLAN. It implements IEEE802.11 AP management, authentication and encryption functionalities. The security portion includes IEEE802.1X/WPA/WPA2/EAP Authenticators, RADIUS client, EAP server, and RADIUS authentication server. The current version of `hostapd` can run in the Linux and FreeBSD operating systems. Besides, `hostapd` implements a group of programming interfaces for both console and graphic applications. These interfaces can be explored by other programmers to build applications with friendly user interfaces. Details of these interfaces and explanation of Figure 4 can be found can be found at [1] [2].

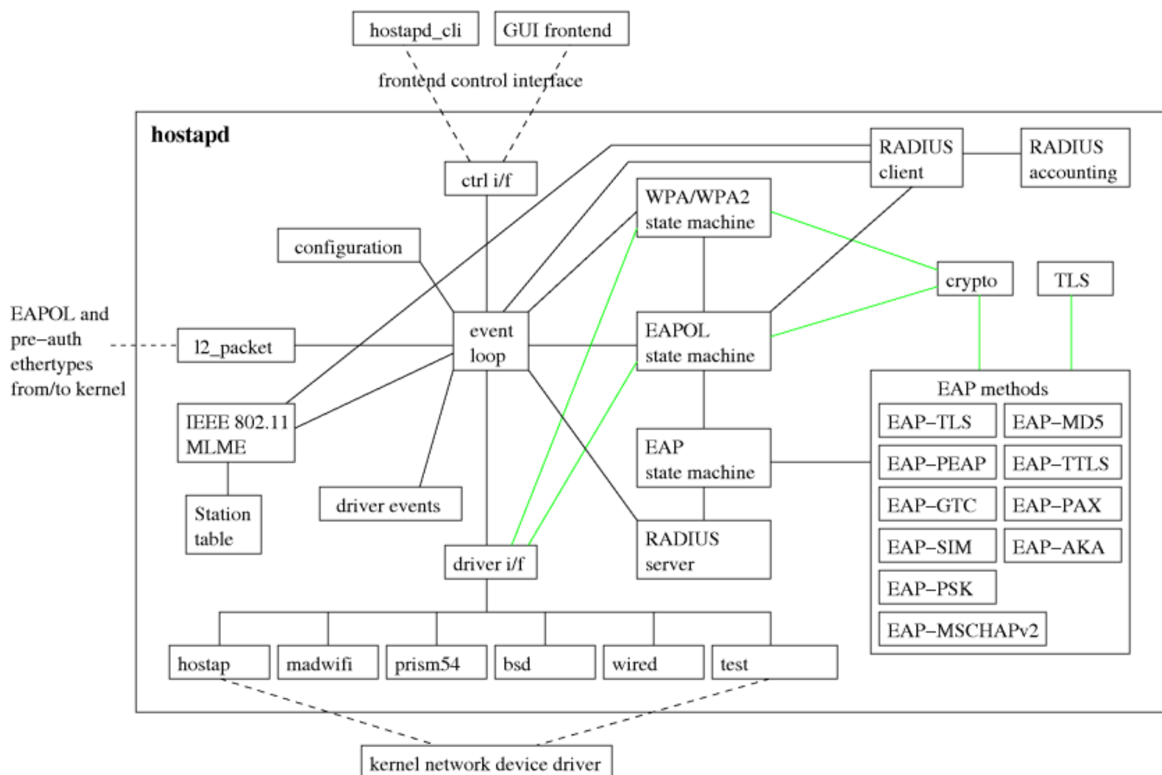

Figure 4 `Hostapd` implementation architecture

2.2.1.1. Drivers and WLAN card

`Hostapd` supports certain number of drivers and WLAN card. The list is given below

Drivers	Cards
Host AP driver (1**)	Prism2/2.5/3
Madwifi driver(2**)	Cards based on Atheros chip set
Prism54 driver (3**)	Intersil/Conexant Prism GT/Duette/Indigo
mac80211-based drivers that support AP mode (4**)	Atheros (ath9k) and Broadcom (b43) chipsets

Table 1 Wireless card supported by `hostapd`

1**http://hostap.epitest.fi/ Please note that station firmware version needs to be 1.7.0 or newer to work in WPA mode.

2**http://sourceforge.net/projects/madwifi/ Please note that you will need to add the correct path for Madwifi driver root directory in .config (see defconfig file for an example: CFLAGS += -I<path>)

3** Prism54 http://www.prism54.org/

4** with driver=nl80211 in the compiling configuration file

This list is not exhaustive. For more information please visit the sites given in reference. The driver and card must be compatible with `hostapd`. And before building `hostapd` the configuration for the driver must be set in the ".config" file.

Madwifi project is participated by a group of volunteer developers who are working on drivers for Linux system. Madwifi project drivers support devices based on Atheros chipsets. Currently three drivers are maintained by this project, namely Madwifi, ath5k and ath9k. Madwifi is stable and open source. But it depends on a proprietary Hardware Abstraction Layer (HAL) which is available only in binary format. Besides the HAL, Madwifi has support for the Wireless Extension API which allows the user to configure the driver with common tools like "`ifconfig`" distributed with the operation system. Ath5k is relatively new and does not rely on the proprietary HAL. Ath9k is the newest driver for all currently available IEEE 802.11n chipsets from Atheros.

MLME is the management entity. It is the place where Physical Layer MAC state machine resides. MLME assists in reaching services like authentication, de-authentication, association, disassociation, re-association, beacon and probing, and so on. MLME can be implemented into the wireless card itself, or it can be realized as software as part of the operating system. The former type is called fullMAC wireless card, and the latter type is called softMAC wireless card. In Linux kernel, MAC80211 provides MLME management services with which drivers can be developed to support softMAC wireless cards. MAC80211 implements the cfg80211 callbacks for SoftMAC devices, and then depends on cfg80211 for both configuration and registration to the networking subsystem. Cfg80211 is a set of interface definitions for driver development. A driver supporting cfg80211 must implement these interfaces. Kernel communicates with the driver via interfaces defined by cfg80211. With these interfaces the Linux kernel can have the driver registered and deliver a task to the driver. NL80211 is another set of interface definitions that allow user space processes to communicate with the kernel. NL80211interfaces are supported inherently in new Linux kernel and some applications are built based on such interfaces, for example `hostapd` and wireless configuration tools.

Before cfg80211 and nl80211 are defined, Wireless Extension is the standard definition of the interfaces for driver development and user space applications. Wireless extension interfaces are built on ioctl()

which is considered as a unstructured system call from user space. Gradually open source developers lost their favor in wireless extension interfaces. People started to develop new interface definitions, and these are just the cfg80211 for driver development and nl80211 for communication between user space and kernel space.

Nl80211 is implemented as `libnl` in Linux system. To use mac80211 based drivers, `libnl` must be present in the system so that the applications can have the interfaces to talk with the driver. For old version of Linux system, the library `libnl` must be installed manually as it was not shipped with the system distribution. Users can directly check its presence in the system library. Or users can discover its absence when they try to compile `hostapd`. If `libnl` is not in the system library and the user try to compile `hostapd` with the mac80211 drivers supported, the compiler will complain the absence of `libnl`. The source of `libnl` is needed for compiling of `hostapd`. And the binary library is needed for running `hostapd`. Figure 5 shows the Linux wireless stack.

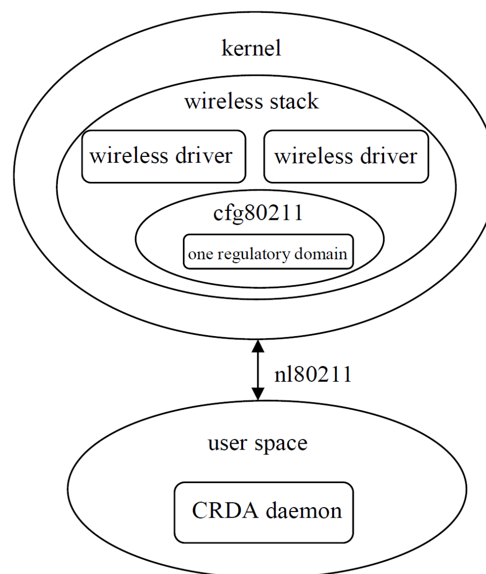

Figure 5 Linux wireless stack

2.2.1.2. Security mechanisms in hostapd

`Hostapd` implements its own security policy with existing security mechanisms. WPA and WPA2 are the protocols implemented internally. WPA2 is a later version of WPA. Improvements in WPA2 include stronger encryption algorithm and better performance in handoff.

- Authentication
  - Both WPA (WLAN Protected Access) and WPA2 can either use a pre shared key (PSK) or use an external authentication server together with EAP (Extensible Authentication Protocol).
- Encryption key generating
  - Both WPA and WPA2 use a 4-way handshake and group key handshake to generate and exchange the encryption key between authenticator and supplicant.
- Encryption algorithm

- WPA uses TKIP (temporary key integrity protocol) which uses algorithm RC4 with per-packet key.
- WPA2 uses CCMP (Counter Mode with Cipher Block Chaining Message Authentication Code Protocol) which uses algorithm AES in counter mode with CBC-MAC.

`Hostapd` can be built with configuration with particular combination of these settings in the `.config` file. However, to make `hostapd` work properly the combination must be first set properly. An example is given in the original compiling configuration file `defconfig`, and comment in this file gives explanation about how to make the settings.

2.2.1.3. Compiling configuration file

As mentioned, `hostapd` is implemented to support multiple drivers, authentication and encryption methods. With the file `.config` user can choose which drivers and security methods to be built for the specific host system. In the source package there is a file `defconfig` which is a reference configuration for specific building. Users can get detailed explanation for each entry of this file. Below is part of the `defconfig` file. The file `.config` when present at compiling is included by the Makefile.

```
# Example hostapd build time configuration
#
# This file lists the configuration options that are used when building the
# hostapd binary. All lines starting with # are ignored. Configuration option
# lines must be commented out complete, if they are not to be included, i.e.,
# just setting VARIABLE=n is not disabling that variable.
#
# This file is included in Makefile, so variables like CFLAGS and LIBS can also
# be modified from here. In most cass, these lines should use += in order not
# to override previous values of the variables.

# Driver interface for Host AP driver
CONFIG_DRIVER_HOSTAP=y

# Driver interface for wired authenticator
#CONFIG_DRIVER_WIRED=y

# Driver interface for madwifi driver
#CONFIG_DRIVER_MADWIFI=y
#CFLAGS += -I../../madwifi # change to the madwifi source directory

# Driver interface for Prism54 driver
#CONFIG_DRIVER_PRISM54=y

# Driver interface for drivers using the nl80211 kernel interface
#CONFIG_DRIVER_NL80211=y
# driver_nl80211.c requires a rather new libnl (version 1.1) which may not be
# shipped with your distribution yet. If that is the case, you need to build
# newer libnl version and point the hostapd build to use it.
#LIBNL=/usr/src/libnl
#CFLAGS += -I$(LIBNL)/include
#LIBS += -L$(LIBNL)/lib

# Driver interface for FreeBSD net80211 layer (e.g., Atheros driver)
#CONFIG_DRIVER_BSD=y
#CFLAGS += -I/usr/local/include
#LIBS += -L/usr/local/lib

# Driver interface for no driver (e.g., RADIUS server only)

#CONFIG_DRIVER_NONE=y

..........
```

Before compiling, a user first copies the reference file `defconfig` and renames it as `.config`. In the file `.config` a user comments and uncomments each entry. A line started with # is ignored while building. Some entries can be easily made mistakes, such as the choosing of driver `NL80211`. The path of source code of `libnl` must be correctly set and the version of `libnl` must be 1.1 or later. Other settings include security protocols and keys.

2.2.1.4. Running configuration file

The file ".config" is used for building the `hostapd` for particular host environment, while the file `hostapd.conf` is used for running settings. After building the capability of `hostapd` on the particular host is fixed. If the user wants service and functionalities that the current building can't provide, the user must rebuild `hostapd` with new settings. In case that the user download the binary `hostapd` from distributors like Ubuntu, the user can only change the behavior of `hostapd` with the running configuration file `hostapd.conf`. The name of the running configuration file can be a name other than `hostapd.conf` and can be located in any place in the file system. But `hostapd` must be started with an existing running configuration file.

A minimal configuration of this file should include the following entries

```
# AP netdevice name (without 'ap' postfix, i.e., wlan0 uses wlan0ap for
# management frames); ath0 for madwifi
interface=wlan0

# Driver interface type (hostap/wired/madwifi/prism54/test/none/nl80211/bsd);
# default: hostap). nl80211 is used with all Linux mac80211 drivers.
# Use driver=none if building hostapd as a standalone RADIUS server
# that does not control any wireless/wired driver.
driver=hostap

# SSID to be used in IEEE 802.11 management frames
ssid=test

# Operation mode (a = IEEE 802.11a, b = IEEE 802.11b, g = IEEE 802.11g,
# Default: IEEE 802.11b
hw_mode=a
# Channel number (IEEE 802.11) (default: 0, i.e., not set)
# Please note that some drivers (e.g., madwifi) do not use this value
# from hostapd and the channel will need to be configuration separately
# with iwconfig.
channel=60
```

Settings in this example may not fit some particular case. One can follow the comment to modify the specifics. The empty line and the line beginning with comment sign is ignored in running.

2.3. An example for hostapd building in Ubuntu 9.10 system

The example in this section is working properly under Linux circumstance. Simplified configuration files are first given and some orderly compiling errors are also listed for the sake of troubleshooting if any readers are considering having their own experiment. In the end of this section a testing scenario is established to prove architecture which includes `hostapd`, `openssl`, `libnl`, Atheros wireless network, `ath5k` drivers, `brctl`.

2.3.1. Compiling hostapd with a minimal configuration

The default compiling configuration of `hostapd` in Linux system is given in file named `defconfig` in the source code package of `hostapd`. As `hostapd` in this project is intended to use mac80211 based drivers, the entry in the compiling configuration file is enabled as

```
# Driver interface for drivers using the nl80211 kernel interface
CONFIG_DRIVER_NL80211=y
# driver_nl80211.c requires a rather new libnl (version 1.1) which may not be
# shipped with your distribution yet. If that is the case, you need to build
# newer libnl version and point the hostapd build to use it.
LIBNL=/usr/src/libnl
CFLAGS += -I$(LIBNL)/include
LIBS += -L$(LIBNL)/lib
```

The comment before the config entry should be removed. Importantly, `LIBNL` should point to the source code directory of `libnl`, `CFLAGS` points to `include/` subdirectory and `LIBS` points to the `lib/` subdirectory. The source code of `libnl` is required by the file `driver_nl80211.c` which is one of the driver wrappers in `hostapd`. This will allow the use driver `ath5k` or `ath9k`. For more configuration of compiling see the default compiling configuration file `defconfig`. We keep other entries the same as the default settings.

2.3.2. Bridge utility

Bridges is a device that connects two or more separate networks together. It can be physical device or a logical device as software. Computers in the same group forming a network usually share security polices and therefore simplify the network in connection and other protection facilities. Computers outside the network can access such security domain via a bridge. This allows the bridge function as a firewall in many cases. Besides security benefit, bridge also functions as a switch or router to deliver packets over the edge of the network. In the case of `hostapd`, the bridge is primarily running as a packet switch, transferring packets from a wired network to a wireless network.

In Linux system bridges is implemented as the wireless tool `brctl` which is included in many Linux distributions, otherwise the users can install it from a binary package or build and install it from source code. Common `brctl` commands are listed below

Run `brctl` without any parameter, it gives these commands

```
    addbr           <bridge>                  add bridge
    addif           <bridge> <device>         add interface to bridge
    delbr           <bridge>                  delete bridge
    delif           <bridge> <device>         delete interface from bridge
    show                                      show a list of bridges
    showbr          <bridge>                  show bridge info
    showmacs        <bridge>                  show a list of MAC addrs
    setageing       <bridge> <time>           set ageing time
    setbridgeprio   <bridge> <prio>           set bridge priority
    setfd           <bridge> <time>           set bridge forward delay
    setgcint        <bridge> <time>           set garbage collection interval
    sethello        <bridge> <time>           set hello time
    setmaxage       <bridge> <time>           set max message age
    setpathcost     <bridge> <port> <cost>    set path cost
    setportprio     <bridge> <port> <prio>    set port priority
    stp             <bridge> <state>          {dis,en}able stp
```

Commands used in the experiment are

```
brctl addbr <bridge name>
```

— create a bridge, one can create multiple bridges in the same system

```
brctl addif  <interface name>
```

— a interface can be a wireless interface like wlan0 or Ethernet interface `eth0`

```
brctl show
```

— show all the bridges that are currently running bridges

```
brctl delbr <bridge name>
```

— delete a bridge

```
brctl delif <bridge name> <interface name>
```

— delete an interface from a bridge

2.3.3. Bridge setup

It takes a few steps to build up the bridge and add interfaces into the bridge. In the context of this project, there are two interfaces added into this bridge, a wired Ethernet interface and a wireless interface (IEEE 80211). The wired interface is the entrance to outside network, for example Internet. The wireless interface is port where other wireless stations try to connect and data transmission happen within the wireless network. In a Linux station, mostly the first wired interface is called `eth0` and the first wireless interface is called `wlan0`. A bridge is created and named `mybr0`. And then `eth0` and `wlan0` are added into this bridge.

— Step 1 create a bridge

```
brctl addbr mybr0
```

— Step 2 add interfaces

```
brctl addif mybr0 eth0
brctl addif mybr0 wlan0
```

Use `dhcpclient` command to get the IP addresses (which one is already exists, which one needs this command, three IP addresses are needed, two interfaces and the bridge itself).

2.3.4. Wireless utilities

ifconfig

`Ifconfig` is used to configure the kernel-resident network interfaces. It is used at boot time to set up interfaces as necessary. After that, it is usually only needed when debugging or when system tuning is needed.

If no arguments are given, `ifconfig` displays the status of the currently active interfaces. If a single interface argument is given, it displays the status of the given interface only; if a single -a argument is given, it displays the status of all interfaces, even those that are down. Otherwise, it configures an interface.

iwconfig

`Iwconfig` like `ifconfig` is a wireless interface configuration utility in Linux. It is used to set the parameters of the network interface which are specific to the wireless operation. `Iwconfig` may also be used to display the parameters and the wireless statistics. `Iwconfig` extracts the information from `/proc/net/`wireless file.

2.3.5. Running hostapd in Ubuntu

In this chapter, we consider an example which illustrates how the network components fit together and the configuration in each of the components.

2.3.5.1. Testing environment

Below we give some of our testing configuration in both hardware and software.

Hardware

CPU – Pentium III (Coppermine) 647.154 MHz   cache size 256KB
Wireless card D-Link DWA-556 PCI
Ethernet card Intel 82557 Ethernet Pro 100

Software

Linux ubuntu 2.6.31-17-generic
driver for Wireless card D-Link DWA-556 PCI – ath5k
driver for Ethernet card Intel 82557 Ethernet Pro 100 – e100
`hostapd` v0.6.9
`libnl` v1.1

2.3.5.2. Configuration file

Before `hostapd` can be launched and set up a local wireless network, there must be a properly configured running config file for `hostapd` to read. The settings in the running configuration file must be within the capability of the compilation. With regard to the compiling configuration used in this project, the entries below are chosen as a simple setting of the running configuration.

```
interface=wlan0
bridge=mybr0
driver=nl80211
logger_syslog=-1
logger_syslog_level=2
logger_stdout=-1
logger_stdout_level=2
debug=4
dump_file=/tmp/hostapd.dump
ctrl_interface=/var/run/hostapd
ctrl_interface_group=0
hw_mode=g
ssid=mytest
auth_algs=3
eapol_key_index_workaround=0
eap_server=0
channel=6
```

The entry interface is set as the first wireless interface wlan0. Bridge is set as `mybr0` which is the one created while bridge setting up in last subsection. Since the compilation can support mac80211 based drivers, it chooses nl80211 as the wireless driver when `hostapd` initiated. The default running configuration file in the source package gives the full version of running configuration of `hostapd` with detailed explanation about how to make settings. It shows all possible settings in the running configuration file.

2.3.5.3. Testing

The testing is taken step by step, and in each step we will give configuration information about each host. Before diving into the procedure, we emphasize that the order is important, even though not necessarily exactly the same.

Before run these commands first turn to root user privilege.

Step 1 launch the computer, and run command `ifconfig` to check the networking interfaces in the system.

```
root@ubuntu:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:50:8b:94:49:80
          inet addr:192.168.1.159  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::250:8bff:fe94:4980/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:938 errors:0 dropped:0 overruns:0 frame:0
          TX packets:224 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:215174 (215.1 KB)  TX bytes:29418 (29.4 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:240 (240.0 B)  TX bytes:240 (240.0 B)

wlan0     Link encap:Ethernet  HWaddr 00:1e:58:4a:c9:c1
          inet addr:172.25.4.192  Bcast:172.25.4.255  Mask:255.255.255.0
          inet6 addr: fe80::21e:58ff:fe4a:c9c1/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:87 errors:0 dropped:0 overruns:0 frame:0
          TX packets:32 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:8424 (8.4 KB)  TX bytes:5201 (5.2 KB)

wmaster0  Link encap:UNSPEC  HWaddr 00-1E-58-4A-C9-C1-34-61-00-00-00-00-00-00-00-00
          UP RUNNING  MTU:0  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Figure 6 Interfaces in up state

There are four interfaces, all in active state, after boot up. The Ethernet interface `eth0` and wireless interface `wlan0` are the two used in this experiment. The interface `lo` is the loopback device. And `wmaster0` is brought up together with wlan automatically.

Step 2 Run the command to configure and bring up `eth0` interface

```
ifconfig eth0 0.0.0.0 up
```

Configure clear IP configuration at interface `eth0`. Doing this is necessary before associate it with a bridge.

Step 3 Run the command to configure and bring up `wlan0` interface

```
ifconfig eth0 0.0.0.0 up
```

Configure clear IP configuration at interface `wlan0`. Doing this is necessary before associate it with a bridge.

Step 4 Run the command to add a bridge in the system

```
brctl addbr mybr0
```

This is to create a bridge in the system. The name (`mybr0`) of bridge created should be the same as the bridge name used in the configuration file of `hostapd` running configuration file (default `hostapd.conf`, `my_hostapd.conf` in this example).

Step 5 Associate the interfaces with the bridge.

```
brctl addif mybr0 eth0
brctl addif mybr0 wlan0
```

This is to associate these two interfaces with the bridge, so that the bridge can deliver packets between them.

Step 6 Bring up the bridge

```
ifconfig mybr0 up
```

The bridge should be brought up before running `dhclient` to configure these interfaces.

Step 7 Run the command to configure these interfaces, including `mybr0`

```
dhclient
ifconfig
```

This command will use DHCP protocol to configure the interfaces. See Figure 7 below

```
root@ubuntu:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:50:8b:94:49:80
          inet addr:192.168.1.159  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::250:8bff:fe94:4980/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2032 errors:0 dropped:0 overruns:0 frame:0
          TX packets:412 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:420923 (420.9 KB)  TX bytes:59715 (59.7 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:14 errors:0 dropped:0 overruns:0 frame:0
          TX packets:14 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:3580 (3.5 KB)  TX bytes:3580 (3.5 KB)

mybr0     Link encap:Ethernet  HWaddr 00:1e:58:4a:c9:c1
          inet addr:192.168.1.162  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::21e:58ff:fe4a:c9c1/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:276 errors:0 dropped:0 overruns:0 frame:0
          TX packets:173 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:128954 (128.9 KB)  TX bytes:26451 (26.4 KB)

wlan0     Link encap:Ethernet  HWaddr 00:1e:58:4a:c9:c1
          inet addr:172.25.4.192  Bcast:172.25.4.255  Mask:255.255.255.0
          inet6 addr: fe80::21e:58ff:fe4a:c9c1/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:192 errors:0 dropped:0 overruns:0 frame:0
          TX packets:204 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:21776 (21.7 KB)  TX bytes:26371 (26.3 KB)

wmaster0  Link encap:UNSPEC  HWaddr 00-1E-58-4A-C9-C1-34-61-00-00-00-00-00-00-00-00
          UP RUNNING  MTU:0  Metric:1
```

Figure 7 Addresses configuration of interfaces

After run `dhclient`, interface eth0 gets an IP address 192.168.1.159, broadcast address 192.168.1.255, and netmask 255.255.255.0. And interface `mybr0` (the bridge) gets an IP address 192.168.1.162, broadcast address 192.168.1.255, and netmask 255.255.255.0. However, the interface `wlan0` gets an IP address 172.25.4.192, broadcast address 172.25.4.255, and `netmask` 255.255.255.0. Interfaces `eth0` and `wlan0` are configured with IP addresses in different network, because physical these two interfaces are located in two different network, separating by the bridge `mybr0`.

Step 8 Launch `hostapd` with the configuration file

```
hostapd /etc/hostapd/my_hostapd.conf
```

```
root@ubuntu:~# hostapd /etc/hostapd/my_hostapd.conf
Configuration file: /etc/hostapd/my_hostapd.conf
Using interface wlan0 with hwaddr 00:1e:58:4a:c9:c1 and ssid 'LinuxHostapd'
```

Figure 8 `Hostapd` running with its configuration file

Launch `hostapd` with its configuration file. The `ssid` of the network is `LinuxHostapd` which is set in the configuration file. `Hostapd` is running over the interface `wlan0` whose hardware address is 00:1e:58:4a:c9:c1.

Step 9 Connect to the AP

```
root@ubuntu:~# hostapd /etc/hostapd/my_hostapd.conf
Configuration file: /etc/hostapd/my_hostapd.conf
Using interface wlan0 with hwaddr 00:1e:58:4a:c9:c1 and ssid 'LinuxHostapd'
wlan0: STA 00:22:43:2f:71:bc IEEE 802.11: authenticated
wlan0: STA 00:22:43:2f:71:bc IEEE 802.11: associated (aid 1)
wlan0: STA 00:22:43:2f:71:bc RADIUS: starting accounting session 4BE3F1ED-00000000
```

Figure 9 A station connect to `hostapd`

This is the response given by `hostapd` after a connection made from a peripheral station. When a station is trying to connect to the assess point `hostapd` does the procedure. It first authenticates the station, next accepts the connection, and then starts a session with a number, `4BE3F1ED-00000000`.

Then the figure below is from station side. The Vista machine is connecting to the AP named "`LinuxHostapd`" which is just the one launched in Ubuntu machine. The signal strength is excellent since these two machines are just beside each other. To show the Vista machine is able to reach the Internet, we give two more figures cut from the command line window. The IP address 192.168.1.219 and `netmast` 255.255.255.0 belong to the same network as the Ethernet interface `eth0` and the bridge `mybr0`. We do ping command as `ping www.google.com` and we get the response from Google server.

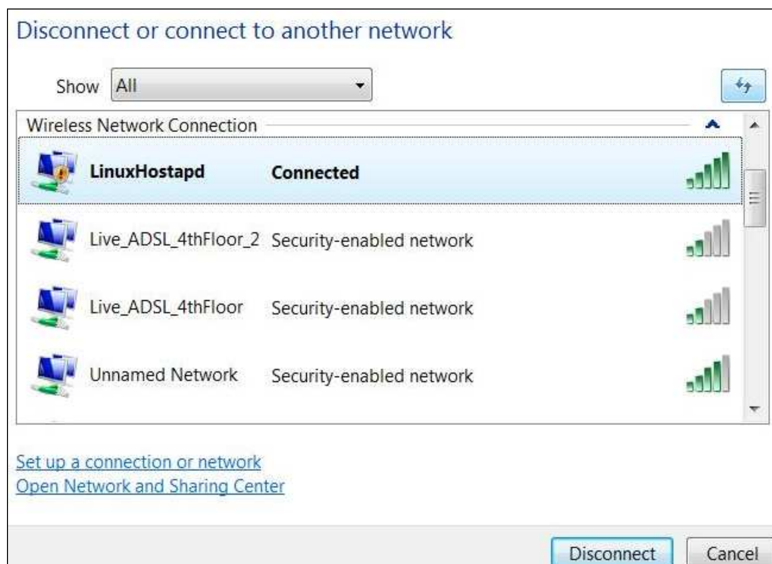

Figure 10 `Hostapd` discovered at vista host machine

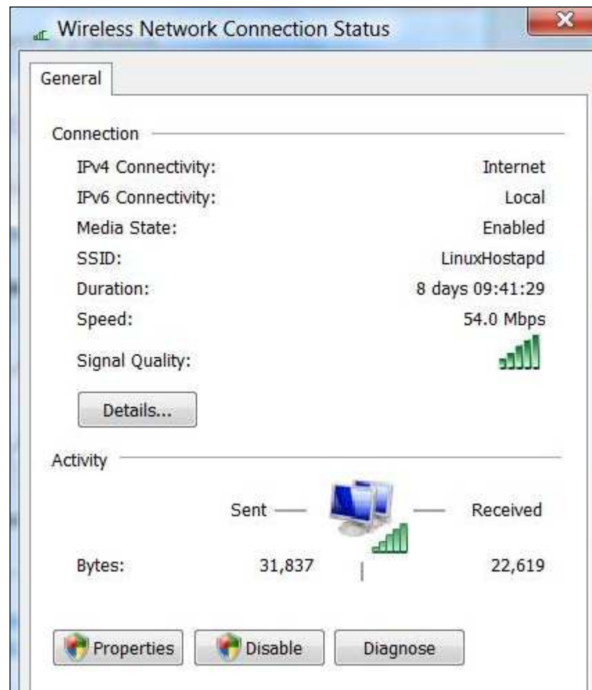

Figure 11 Connection state of vista host machine

```
C:\Users\bmgw>ipconfig

Windows IP Configuration


Ethernet adapter Bluetooth Network Connection:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Wireless LAN adapter Wireless Network Connection:

   Connection-specific DNS Suffix  . : SSG5-Serial
   Link-local IPv6 Address . . . . . : fe80::289a:85b1:c471:d296%11
   IPv4 Address. . . . . . . . . . . : 192.168.1.219
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.1.1
```

Figure 12 Address configuration of vista host machine

```
C:\Users\bmgw>ping www.google.com

Pinging www.l.google.com [66.102.13.99] with 32 bytes of data:
Reply from 66.102.13.99: bytes=32 time=48ms TTL=52
Reply from 66.102.13.99: bytes=32 time=48ms TTL=52
Reply from 66.102.13.99: bytes=32 time=47ms TTL=52
Reply from 66.102.13.99: bytes=32 time=49ms TTL=52

Ping statistics for 66.102.13.99:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 47ms, Maximum = 49ms, Average = 48ms
```

Figure 13 ping `google.com` from vista host machine

3. Android building system

This chapter introduces the Android building system. First we explain how Android source is organized. Next we describe the building environment in Ubuntu Linux, and the concept of cross compiling. Then we explore the directories in the hierarchy where some of the Android system source code and the Android building system reside in, including build/, vendor/, out/, prebuilt/ and external/. We give directions about where building changes are made in our project and can be made in other projects. Similar statement about the Android building system can be found on various websites. However we still have such a chapter because understanding the building system is fundamental to made progress in the project, and along with the exploration inside the source code we gain some experience that may help colleges who are working on the same issues.

3.1. Android source code

We'll start from pulling the Android source code down to local machine, followed by overview on the source code, and then more detailed explanation about important directories for particular building configuration.

3.1.1. Tools working on source code

To better manage the source code for downloading and uploading, the Android development group organizes the source code in `git` repository which was initially for Linux kernel development. To make it even easier, Google has done some work to wrapper `repo` above `git`. Therefore mostly when we do uploading or downloading we use the `repo` tool.

3.1.2. Source code branches

Over the past years a lot of branches have been make for various reasons, like porting to different target machines. Before down the source code one must be clear about which branch is needed. Generally, most people will follow the procedure at [3]. But this is just for building some emulators for ARM architecture. It is not proper for porting code to real x86 machines. Even for projects working on porting to x86 machines there are different working groups called `Android-x86` and `Androidx86`. `Android-x86` is the one mainly focus on Eee PC which is the machine used in this project. But again, different branches are made in this project. Branches `master`, `eclair-x86`, `donut-x86` are the three commonly seen. These branches have their own progress and maintained during different period in the community. And one more to mention is that there are two sites to get source code for the same branch

Main site `git://git.Android-x86.org/platform/manifest.git`

Mirror site `git://Android-x86.git.sf.net/gitroot/Android-x86/manifest.git`

Source code at these two sites can be asynchronous. We recommend downloading the latest source code at the main site.

3.1.3. Downloading the source code

In this project we work on the source porting to Eee PC which is x86 architecture based machine. And we choose to use the dnout-x86 branch source code from the main site.

When run the command `repo init`,

```
$ mkdir mydroid
$ cd mydroid
$ repo init -u git://git.Android-x86.org/platform/manifest.git -b donut-x86
```

```
$ repo sync
```

This will pull down the source code. The process may take one hour and more, depending on the network situation.

After it finished, the following fold structure can be seen in the root

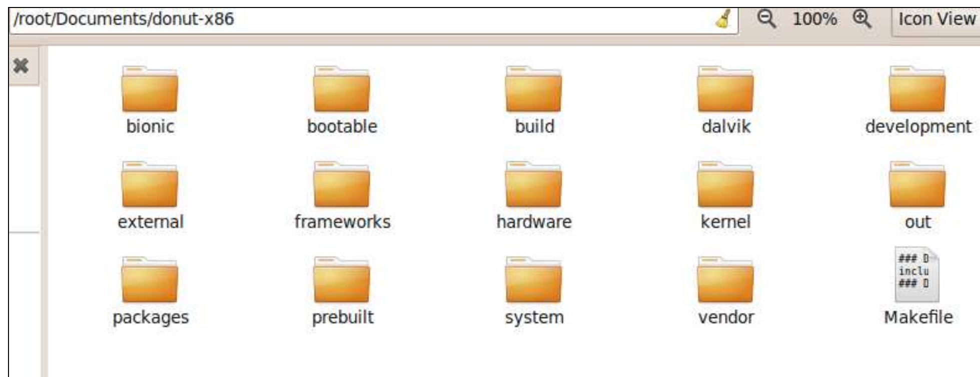

Figure 14 donut-x86 branch source code overview at the top

From now on all explanation about the source code is on the Android-x86. Here is a simple description about each directory in Figure x, more detailed discussion on some of them are given in following sections of this chapter.

bionic

— The source code of C libraries, including `libc` (c library), `libm` (math library), `libstdc++` (c++ library), `libthread` (thread library), `libdl` (dynamic linking interface library), and linker (the program linker).

bootable

— Contains the source code for building a boot loader and disk installer.

build

— The Android building system. It contains definitions of the building system, like macros and functions, as well the crossing compiling compilers, linkers, assemblers. Besides, it detects the building system environment, set up the output file directories, and make up the images after a successful compilation. More details will be given in the following section.

dalvik

— The source code for building a virtual machine in the Android platform. It's called dalvik virtual machine optimized by Google cooperation for mobile device for interpret Java byte code.

development

— Source code for building development tools for the Android platform. These tools include SDK (only the property files are here, the source of SDK is in a separate directory called sdk), NDK, and emulator. User help documentation for development is also included in this directory.

external

— This directory includes third part packages which can be system tools (for example networking tools) or applications.

frameworks

— Contains the source code for building the core libraries for applications, as well as policy definition for system configuration.

hardware

— Libraries for hardware abstraction.

kernel

— The source code of Linux kernel.

out

— The directory for containing all output files, including compiling intermediate files and the final produced image files and binary Linux kernel.

packages

— Source code for various applications, for example music, browser and so on.

prebuilt

— Contains binaries for supporting Windows and Mac OS building. The most important things are these ABIs (Application Binary Interface) corresponding to various architectures.

sdk

— The source code for build the SDK.

system

— Contains system components like system booting environment setting, Bluetooth tools, and WLAN drivers and tools. These tools can be binary or source, depending on the vendors' policy to their source code. This is actually part of Linux system. However, in the Android source code architecture there is only Linux kernel directory. Any other system level code touching hardware or system environment setting can be placed in this directory.

vendor

— This is the directory for customizing building process. New product profile can be defined to build feature specified Android platform. The user can either choose to build everything from source or make some parts to be prebuilt and tell the building system to incorporate these prebuilt packages into the final built images.

At the top level of the directory hierarchy, there is one Makefile which is the only makefile named Makefile in the whole building system. It is the entry of the whole building process.

3.2. Build environment on Ubuntu platform

It is recommended that the Android source code building process taking place in Ubuntu platform. In this subsection we show the additional elements needed to make the platform sufficient to build the Android source code.

3.2.1. Ubuntu Linux (32-bit x86)

To set up your Linux development environment, make sure you have the following:

Required Packages:

— `Git 1.5.4` or newer and the GNU Privacy Guard. `Git` is a version control tool. It can be used to pull down source from or upload source to remote `git` repository. It also helps to manage local source project, with a full track on the modification and the ability to get back to previous version after any committed changes.

— `JDK 5.0`. `JDK` is the development toolkit for Java programming language. Applications on Android need these for compiling.

— `Flex`, a text scanner used to tokenize the input to recognized descriptions. A script language interpreter need some a scanner for analyzing user input either from the standard input or from text file.

— `Bison`, the actual interpreter transforming the known tokens to be a piece of c program. Bison does its interpretation after `flex` generates the tokens. And after `bison` translates the tokens to be pieces of c program the user input can be executed to get the expected results. `Flex` and `bison` are both part of a script like language interpreter.

— `Gperf`, a perfect hash function which can produce a unique output for each given unique input. `Gperf` is used by C and C++ compilers to recognize the language reserved words.

— `Libsdl-dev`, the library for Simple DirectMedia Layer development. It is a library which allows programs portable low level access to a video framebuffer, audio output, mouse, and keyboard. The library is needed by a compiler and a link when the program is using SDL.

— `Libesd0-dev`, the library for Enlightened Sound Daemon development. It can be used to mix several audio streams to be played in a single sound device. When a program uses this library the compiler and linker will need it for compiling and running that program.

— `Libwxgtk2.6-dev` (optional), wxWidgets Cross-platform C++ GUI toolkit (GTK+ development). It is a C++ class library for developing wxWidget programs. A bunch of GUI components are provided in this library. And it's a cross platform library which can be used most popular platforms as well as some unpopular platforms.

— `Build-essential`, a package for building Debian packages. This is needed during compiling for building some packages in Debian compatible format.

— `Zip`, a utility for building .zip archive files. This is needed for building the final archives during building the Android platform.

— `Curl`, a utility for getting files from a HTTP, HTTPS, or FTP server as well as some other application level protocol based servers. This is needed for pulling down Android source from the Git repository.

All libraries and tool mentioned above can be downloaded and installed with the following command

```
$sudo apt-get install git-core gnupg sun-Java5-jdk flex bison gperf
libsdl-dev libesd0-dev libwxgtk2.6-dev build-essential zip curl
libncurses5-dev zlib1g-dev
```

One more tools can be installed to help working with memory leakage examination, `valgrind` can be installed with

```
$sudo apt-get install valgrind
```

Ubuntu Intrepid (8.10) users may need a newer version of `libreadline`, which is an assistant for working with command line so that various program can share a uniformed command line interface. It can be installed with

```
$sudo apt-get install lib32readline5-dev
```

3.3. Cross compiling

The compiling process is to build the executable for a host system on a building system. Here the host system is the platform on which the executable is going to run. Building system is the platform from which the executable is produced. When the executable is a compiler, there is one more system involved which is named target system. The target system is the platform for which the compiler is going to make executables.

Cross compiling is a type of compiling which has host system different from building system. For example, in this project the building system is Linux while the host system is Android. This is because the building process is taking place on Linux and the executables are going to run on Android.

To achieve cross compiling there must be cross compiling tool chain which contains a cross compiler, a cross assembler, and a cross linker. The process in this project is more complicated for the reason that it is going to build up both the Android platform and various applications. The Android platform is a combination of Linux kernel and dalvik virtual machine for running Java code. The applications are built with Java programming language. This means that there must be multiple cross compilers, cross assemblers and cross linkers for building the Android source.

Going further in the compiling, all libraries in both C and Java must be available for building the platform and applications. Particularly, for C program compiling the headers and C libraries must exist, and for Java program compiling the Java libraries must exist. For Android the C libraries are built from C source in the directory “bionic”, and the necessary header files can be in “bionic”, “kernel” or other C code directories. For code which is not open source, the libraries are provided by vendors in binary format. Similarly, these Java libraries can be either built from Java source code or provided in `jar` packages.

3.4. Cross compiling tool chains for Android in Ubuntu system

Android itself is an operation system. However, before it can be properly functioning as a platform for various applications it must be built from source code. The building process can be taken place in a few existing platforms including Linux (Ubuntu is recommended), Mac OS, and virtual machine. In last section it gives the needed configuration in Ubuntu 32 bit platform. For other platforms detail information can be found at [4].

3.4.1. Cross compiling ABI

Such a configuration is the general environment of building for all supported architectures. When it comes to configuration for specific architecture, the ABI (Application Binary Interface) must be considered. ABI is a concept different from API. ABI is specific to the underlying processor, while API is specific to the operating system. Programs written with regard to particular ABIs can only run on platforms that are sitting on top of the corresponding processor. Currently the Android platform can be built to be running on ARM or X-86 architecture. For a particular building, the ABI must be set to be the proper one. The setting of ABI is controlled by a building environment variable called

TARGET_CPU_ABI. If the building target is a simulator, it doesn't need a CPU ABI. So the TARGET_CPU_ABI is set as none.

```
TARGET_CPU_ABI := x86       #if you build for x86 architecture
TARGET_CPU_ABI := armeabi    #if you build for arm architecture
TARGET_CPU_ABI := none      #if you build a simulator
```

TARGET_CPU_ABI is set in the file of Boardconfig.mk which is one of essential configuration files for a particular building. The actual files of various ABI is located at the directory prebuilt/ as following

```
$(combo_target)TOOLS_PREFIX :=prebuilt/$(HOST_PREBUILT_TAG)/toolchain/i686
-unknown-linux-gnu-4.2.1/bin/
```

HOST_PREBUILT_TAG is the build TAG which can be linux-86, or linux-x86_64 and a lot more. In this project it is set as linux-x86 for the reason that the built Android system is going to be running on a 32-bit x-86 platform. $(combo_target)TOOLS_PREFIX as a whole determines the name of the path of the ABI in the file system. This setting will ultimately determine the actual compilers for compiling C and C++ source files.

3.4.2. Cross linker

The Android platform uses its own program linker which is different from the one in a regular Linux system, therefore the linker issue must be considered when building Android applications and tools. The programs must be linked with the linker that is built from source in Android. The source code of Android linker is under the directory of bionic/linker/. And again the linker is specific to a particular architecture on which the building target is going to run. The supported architectures are ARM and x86.

3.5. Build variables and build process

3.5.1. Build for a new product

In Android building system there are lots of environment variables which control the build properties. The building system defines most of these variables used internally, as well as some variables to be interfaces for users to set some of the building process properties. However, as long as the users understand these build variables they can change these variables to be the value they want. And for the sake of convenience to modify the interface variables, the building system creates several separate files to hold them under the directory of vendor/company_name/. company_name is chosen by users. And then under this directory user creates the required files suffixed with .mk. Specifically, to build for a new product the user needs to have a product tree as following:

```
• <company_name>
    o <board_name>
        ▪ Android.mk
        ▪ product_config.mk
        ▪ system.prop
    o products
        ▪ AndroidProducts.mk
        ▪ <first_product_name>.mk
        ▪ <second_product_name>.mk
```

Under company_name/ there are two subdirectories, board_name/ and products/. Product_config.mk and Android.mk and are necessary in subdirectory board_name/, while system.prop is optional.

`Product_config.mk` is used to define Product specific compile time definitions which are going to override the default values in the files `BoardConfig.mk` and `config.mk` that are both in the directory `build/`. `System.prop` is needed in the `board_name` subdirectory only when the user wants to override default settings in `build/` directory. In the product subdirectory there must be an AndroidProducts.mk file which points to the actual product definition files. In this example, there are two product definition files first_product_name.mk and second_product_name.mk. The product definition file override default settings in generic.mk which is under one of subdirectories of `build/`. In the product definition file user can redefine `PRODUCT_PACKAGES` to include the packages in the final product. Others like `PRODUCT_BRAND`, `PRODUCT_DEVICE`, and `PRODUCT_NAME` can be override to be the specific names for the particular product.

3.5.2. Default settings and build process

As mentioned, there are always default settings of all build variables in the `build/` directory. Even if without any user defined files, the build process will still precede successfully according to the default settings. Before run the make command, the user should first run the file `build/envsetup.sh` which is a script file defining some useful functions that can ease the building system. After receiving the make command from the user, the building system first looks at the file named Makefile at the top of the source code hierarchy. There is only one simple entry in the top Makefile. It tells the building system to look for more information at the file `build/core/main.mk`. `Main.mk` is the actual entrance of the Android building system. It calls other files named with `.mk` suffix to set up the default values for all building variables. The most notable settings include the build type (`eng`, `user`, `userdebug`, or `development`), final included packets, host architecture, host platform, target architecture, target platform, and output directory.

3.5.3. Adding a new packages for building

Expect for change settings for specific device build, the user can just add external packages to be compiled together with the original source code. This allows users to extend the capability of the Android platform with their own intention. Such an external package can be C/C++ source or Java source coded.

3.5.3.1. Template makefiles in the building system

There is a template file for building each kind of these packages. The template is available for building applications, shared libraries and static libraries. The list below gives these template files with a simple description.

— **binary.mk**

Define PRIVATE_ variables used by multiple module types, like the file suffixes, compiling flags, install header files, add default shared libraries (`libc`, `libm`, and etc.), collects all the built libraries. It also defines compiling flags and library dependencies.

— **host_executable.mk**

Define common rules for building executables. It calls the `binary.mk` file. The content of this file is quite simple, as following

```
LOCAL_IS_HOST_MODULE := true
ifeq ($(strip $(LOCAL_MODULE_CLASS)),)
LOCAL_MODULE_CLASS := EXECUTABLES
endif
ifeq ($(strip $(LOCAL_MODULE_SUFFIX)),)
LOCAL_MODULE_SUFFIX := $(HOST_EXECUTABLE_SUFFIX)
```

```
endif

include $(BUILD_SYSTEM)/binary.mk
$(LOCAL_BUILT_MODULE): $(all_objects) $(all_libraries)
  $(transform-host-o-to-executable)
  $(PRIVATE_POST_PROCESS_COMMAND)
```

— **host_Java_library.mk**

This is the template for building a Java library. It doesn't call `binary.mk`. It sets the value for these local variables needed to build a Java library.

```
LOCAL_MODULE_CLASS := JAVA_LIBRARIES
LOCAL_MODULE_SUFFIX := $(COMMON_JAVA_PACKAGE_SUFFIX)
LOCAL_IS_HOST_MODULE := true
LOCAL_BUILT_MODULE_STEM := Javalib.jar
```

— **host_shared_library.mk**

This is the template for building a shared library. It calls binary.mk because a shared library is just one kind of binaries built from C/C++ code.

```
include $(BUILD_SYSTEM)/binary.mk
$(LOCAL_BUILT_MODULE): $(all_objects) $(all_libraries)
$(LOCAL_ADDITIONAL_DEPENDENCIES)
  $(transform-host-o-to-shared-lib)
```

— **host_static_library.mk**

This is the template for building a static library. It calls binary.mk because a static library is just one kind of binaries built from C/C++ code.

```
include $(BUILD_SYSTEM)/binary.mk
$(LOCAL_BUILT_MODULE): $(all_objects)
$(transform-host-o-to-static-lib)
```

— **Java.mk**

This makefile defines common variables and set up the environment for building Java code. It is called by other makefiles when they need to build Java sources.

— **Java_library.mk**

This is the template for building a Java library which can be either a shared library or a static library. It is called by the makefile `static_Java_library.mk`.

— **raw_executable.mk**

This is the template for building an executable. It calls the makefile binary.mk for it is just one type of binary. This specific type of executable depends on the source code and the target platform. `Raw_executable.mk` calls the linker to link the objective files to be an executable. Again the specific tools used depend on the source code and the target platform.

```
$(LOCAL_BUILT_MODULE): $(all_objects) $(all_libraries)
  @$(mkdir -p $(dir $@)
  @echo "target Linking: $(PRIVATE_MODULE)"
  $(hide) $(TARGET_LD) \
      $(addprefix --script ,$(PRIVATE_LINK_SCRIPT)) \
      $(PRIVATE_RAW_EXECUTABLE_LDFLAGS) \
      -o $(PRIVATE_ELF_FILE) \
```

```
        $(PRIVATE_ALL_OBJECTS) \
        --start-group $(PRIVATE_ALL_STATIC_LIBRARIES) --end-group \
        $(PRIVATE_LIBS)
    $(hide) $(TARGET_OBJCOPY) -O binary $(PRIVATE_ELF_FILE) $@
```

— **raw_static_library.mk**

This is just a wrapper for building static library. It does nothing but `include $(BUILD_STATIC_LIBRARY)`.

3.5.3.2. Adding a new module and writing the Android.mk

For each application there must be a makefile named Android.mk for the building system to discover it and building the source according to the user's indication. A makefile typically include all the following elements:

— `LOCAL_MODULE` defines the build name for the particular package.

```
LOCAL_MODULE := <build_name>
```

— `CLEAR_VARS` clears all local variables. The Android platform defines a group of local variables for controlling the building of each particular package. They need to be cleared each time when the building system enters a new package to cancel the influence of settings of last package

```
include $(CLEAR_VARS)
```

— `LOCAL_SRC_FILES` indicates all source files that eventually compose the final target.

```
LOCAL_SRC_FILES := <all sources>
```

— `LOCAL_MODULE_TAGS` indicates the build tags for this package. Build tag tells the building system when to build the package with regard to each build flavor (can be `eng`, `user`, `userdebug`, or `development`).

```
LOCAL_MODULE_TAGS := user development
```

— `LOCAL_SHARED_LIBRARY` tells the building system which shared libraries this package depends on. Packages generally always rely on external libraries to achieve a successful building. C/C++ packages can utilize either shared libraries or static libraries. Java source is built with some .jar class libraries.

```
LOCAL_SHARED_LIBRARY := libc libcutils
```

— `BUILD_EXECUTABLE` tell the building system to build this package to an executable. Both C/C++ and Java sources can be built to be executables, shared libraries, or static library. There is a template file in the build/ directory for building a target in each of these types respectively. Refer to the template files mentioned in this subsection earlier for details.

```
include $(BUILD_EXECUTABLE)
```

Here is a simple example of Android.mk

```
LOCAL_PATH := $(my-dir)
```

```
include $(CLEAR_VARS)
LOCAL_MODULE := hostapd
LOCAL_SRC_FILES := hostapd.c
LOCAL_MODULE_TAGS := eng development
LOCAL_SHARED_LIBRARIES := libc libm libcutils
include $(BUILD_EXECUTABLE)
```

3.6. Out directory

The `out/` directory is initially empty. It is the directory where the all output files are located during the build process, including intermediate object files, final executables, and the images. The building system creates a subdirectory named after the value of `TARGET_PRODUCT`. The user can either define the variable in the `ProductConfig.mk` file or at the typing of make command. In this project, the intention is on the Linux kernel and three image files, `initrd.img`, `ramdisk.img`, and `system.img`. Linux kernel provides all actual system resource management. All C/C++ coded programs communicate directly with Linux kernel. `Initrd.img` contains the init script which sets up the system running environment and launches daemons and kernel modules. `Ramdisk.img` is a temporary root file system which provides all the tools for setting up system environment and mounting the actual root file system. `System.img` holds the file system. Therefore all application packages are housed in this image. There might be other image files created. The user can fetch the files to compose a properly functioning the Android platform.

The user should be careful about the disk space holding the `out/` directory. The reason is that there are many intermediate files are going to be generated in the building process and huge space is needed. Many intermediate files have been removed before the process terminates. This might lead the user to think of having less disk space for the process. A minimal of 6G free space is necessary.

3.7. Details about other directories like system and prebuilt

These directories can be important in some cases according to specific build requirement. Below is some description about them.

3.7.1. external/ directory

If the user thinks of adding new packages into the hierarchy the directory `external/` is generally the place. `External/` holds all third party packages. The packages can be Java program or C/C++ applications that are built as background services. There are four packages directly evolved in this project, `hostapd`, `libnl`, `opeenssl`, and `bridge-utils`. They are all placed in `external/` directory. There is no requirement of the building system for users where to place their packages, however for the sake of clearance of source code management `external/` is the proper directory.

3.7.2. prebuilt/ directory

Android building system needs a lot of tools for it to be functioning in variable building settings. These tools are mostly located in `prebuilt/` directory. When do crossing compiling the available tools in this directory are fundamental. There are tools for building the Android platform running on ARM and x86 architecture. Android currently only support two types of building. The user can create the tools for Android building system individually so that more architecture can be supported in particular user's need. Except for these tools, some libraries are also placed here. These libraries are then used by the building system to create applications that are specific to the Android platform.

3.7.3. bootable/ directory

This directory includes sources for building the bootloader and disk installer as well as the system initialization image. After the kernel is loaded it first do kernel initialization like setting the segmentation and paging table, configuration environment for process management, and enabling interrupt in both hardware and software form. And then it turns to do initialization for running user space services, like loading some modules to support particular operation, launching and configuration some programs running as services depending on the user configuration in the init scripts. The last type of initialization is performed by program built from source in this `bootable/` directory.

4. Hostapd in Android

The Android platform comes with its own building system. Any new external package must rewrite its makefile to be recognized and compiled in the building process. This chapter gives detailed explanation about the makefiles of these components in the Android building system.

4.1. Configuration for building hostapd

`Hostapd` is the starting point for the need to introduce all other components. `Hostapd` in Android, just like in Linux, requires a driver to do the physical signal processing. We grab Linux kernel 2.6.29 [15] as the kernel of our Android platform. This kernel ships with the ath9k driver for Atheros chipset cards. Tracing ath9k to the ongoing Madwifi driver project, it gives us the information that ath9k is a mac80211 based driver. Mac80211 is a new wireless stack in Linux to replace the old wireless extension standard. As mentioned in chapter 2 (where mac80211 is mentioned), all daemons and applications running in user space talk with Linux wireless driver via the nl80211 interfaces which are implemented as the `libnl` in source code. One can download and compile `libnl` to have Linux support the mac80211 based drivers. Therefore as we decide to choose ath9k driver, we should set "`CONFIG_DRIVER_NL80211`" with yes, and tell the directory of `libnl` to the compiler by setting `CFLAGS` and `LOCAL_SHARED_LIBRARIES` compiling flags.

The driver option is the only special setting in the compiling configuration file of `hostapd`. We keep all other security settings unchanged as they are in the default configuration file. Android building system has many local variables which tightly control the building in each part of the whole source package. It can be very tricky sometimes when the source code organizing changes. To be conservative we don't use subdirectory makefiles for `hostapd` in the Android source code hierarchy, and wherever can be tricky we use absolute directory instead of relative directory. We recommend this because the default settings of these source directories result in many errors in the compiling. The other reason for this is that the building in this project is a cross compiling, therefore many header files and libraries are located in the indicated directories of the source code hierarchy instead of the standard directories as native compiling. However the compiler will go to these default places to find the headers and libraries if there's no explicit indication telling the compiler where to look at these files. The absolute directories will avoid errors resulting from these settings which sometimes are unclear to third part engineers. When the engineer doesn't know the profound details of the building system, it's better to use absolute directories in both the configuration files and the header files in the source file. This will get rid of most of the compiling errors. Since we have only one makefile for one package we put the makefile at the top of each package.

The latest version of `hostapd` is labeled as 0.7.2 as this project is finished. The latest version can be unstable in some important aspects. We choose to use version 0.6.9. There is a reported bug of version 0.6.10 when the underlying wireless driver is `ath9k`. One should be careful about the version issues so that unnecessary work can be avoided. We place the source code of `hostapd` under `external/` directory and rewrite the makefile with respect to the Android building system. The new makefile is always named as `Android.mk`. It then is put on the top directory of `hostapd`. As recommended we change the directories of header files to be absolute directories. The structure of source code keeps the same as downloaded.

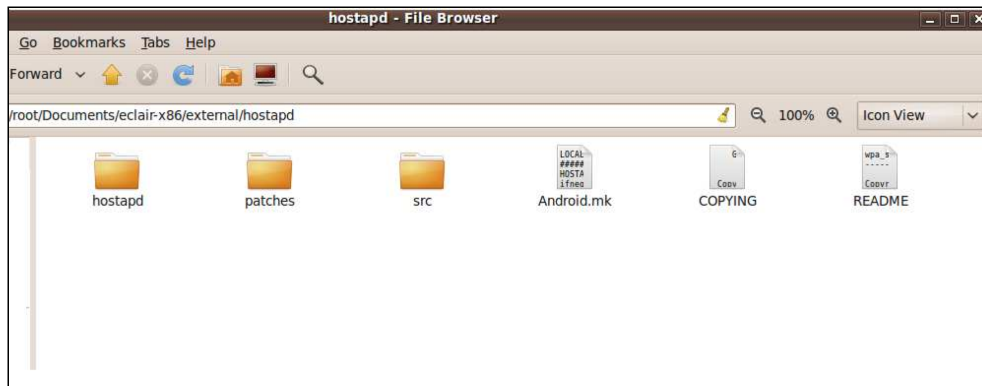

Figure 15 `Hostapd` source code overview at the top

`Hostapd` source package is relatively large compared with `libnl` and `bridge-utils`. We'll give the `Android.mk` files for these two packages in the following sections and leave `Android.mk` file in Appendix B .

```
# hostapd
hostapd v0.6.9
User space daemon for IEEE 802.11 AP management,
IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator
Copyright (c) 2002-2009, Jouni Malinen <j@w1.fi> and contributors

usage: hostapd [-hdBKtv] [-P <PID file>] <configuration file(s)>

options:
   -h   show this usage
   -d   show more debug messages (-dd for even more)
   -B   run daemon in the background
   -P   PID file
   -K   include key data in debug messages
   -t   include timestamps in some debug messages
   -v   show hostapd version
```

Figure 16 successful running of `hostapd` in Android

4.2. Openssl in Android

`Openssl` implements both `SSL` and `TLS` which are both basis for secure web communication. It has been already placed in the Android source code tree under the `external/` directory. `Hostapd` needs `openssl` to work with protected wireless access. The only thing needed to do is to tell the directory of the `openssl` source to the building system when it comes into the `hostapd` subdirectory. With this setting of `CFLAGS` in `Android.mk` file of `hostapd` the building system will be able to find the headers needed by security implementation in `hostapd`. `Openssl` will be compiled with the Android source code. Therefore telling the name of libraries resulting from `openssl` is sufficient for the linker to find them. There are two libraries built from `openssl` source code, `libssl` and `libcrypto`. They both should be added in the local variables `LCOAL_SHARED_LIBRARIES` in the `Android.mk` of `hostapd`.

4.3. Configuration for building libnl

`Libnl` is needed for `hostapd` to communicate with wireless driver. There is no source code or binary library in the Android source code `git` repository. When it is necessary in any particular building, the source code of `libnl` must be downloaded and placed in the some directory like `external/` in Android. And again there must be an Android style makefile for the building system to find such a third party package. `Hostapd` v0.6.9 requires a newer version of `libnl`, at least v1.0. We choose v1.1 in this project.

The downloaded source code package includes the code for building the library as well as code for other use. Since we only need the library we just write Android.mk for the library code which is together located in the subdirectory `lib/`. The compiler also needs the header files in the subdirectory `include/`. There is only one `Android.mk` file for the `libnl` source code package. Therefore, the easiest way is to put the Android.mk file at the top of `lib/` subdirectory. And when the `.c` file and its included header files are not in the same directory we use absolute directory to tell the compiler where the header file is. The structure of source code is given as Figure 17 and the shadow text shows the content of our `Android.mk` file for `libnl`.

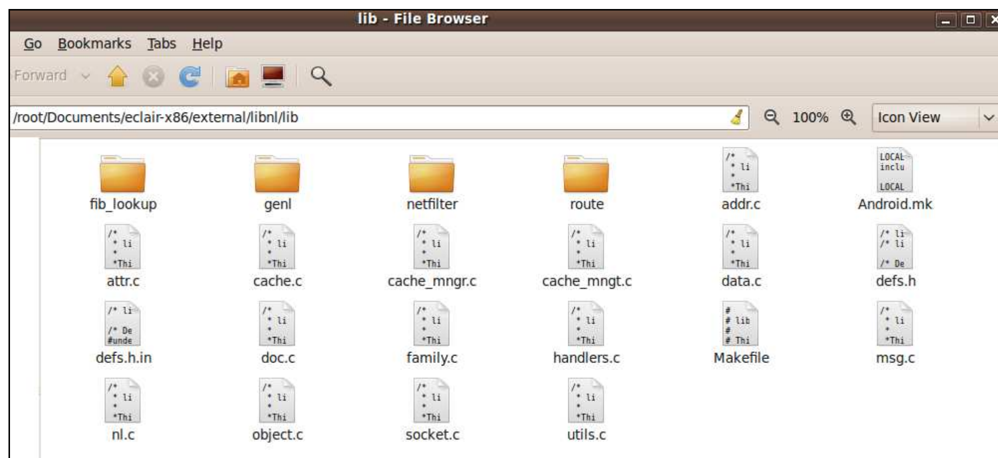

Figure 17 Source code overview of `libnl` in the lib subdir

```
###################################################################Android
.mk file for libnl
##################################################################
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SRC_FILES:= addr.c attr.c cache.c cache_mngr.c cache_mngt.c data.c
doc.c family.c handlers.c msg.c nl.c object.c socket.c utils.c genl/genl.c
genl/ctrl.c genl/mngt.c genl/family.c

LOCAL_C_INCLUDES += \
 bionic/libc/include \
 bionic/libc/arch-x86/include \
 bionic/libc/kernel/common \
 bionic/libc/kernel/arch-x86 \
 bionic/libm/include \
 external/openssl \
 external/openssl/include \
 external/openssl/crypto \
 external/libnl/include
```

```
LOCAL_SHARED_LIBRARIES := libc libcutils libcrypto libssl

LOCAL_MODULE:= libnl

include $(BUILD_SHARED_LIBRARY)
```

In the `Android.mk` file, the line starting with # mark is ignored by the compiler. The first thing `Android.mk` does is to tell the building system that now it steps into a new package which is the `lib/` subdirectory of `libnl` in the `external/` directory. It does this by `LOCAL_PATH:= $(call my-dir)`. Next it clears the local variables that the building system used for the previous package with `include $(CLEAR_VARS)`. The order here is strict for the first two callings in the Android.mk file. After these two steps, the other five callings can be made in any order. `LOCAL_SRC_FILES` indicates the source code used to build the ultimate libraries. There is no need to tell the dependencies between these `.c` files, instead just a whole source package is sufficient. The Android building system will figure out these dependencies and make the right call at the right time. The source code included in this variable is just these under the `lib/` subdirectory. `LOCAL_C_INCLUDES` is actually not needed here because all header files out of the `lib/` subdirectory are indicated with their absolute directories. The compiler can find them with the absolute directory. This is not quite elegant since it can be vary tedious when the source package is large and lots of header files are included or when lots of external packages are needed to be added into the building system. We leave this variable here to tell the reader that the better way to indicate the location of header files is to tell the path of them in this variable. `LOCAL_SHARED_LIBRARIES` then holds on all necessary libraries for building the `libnl` library. We successfully build the `libnl` library with these libraries. But we have not yet test which libraries are necessary. It can be that some of them have no influence for building `libnl`. After indicating the necessary components for building the library we give the library a name `LOCAL_MODULE:= libnl`. In the end it's time to call the compiler and linker to do their job by calling `include $(BUILD_SHARED_LIBRARY)`. This is to build a shared library.

4.4. Configuration for building bridge-utils

`Bridge-utils` is not the necessary part for building `hostapd`. However to let `hostapd` really function as an AP to provide Internet service we first need to make the Internet available to AP. This is achieved by adding a bridge between the AP and another interface which is connected to the Internet. In real situation of mobile case, it's most likely that the mobile can access the Internet via the telephone network. And then a bridge is running to move data from telephone radio interface to the WLAN interface which `hostapd` is taking care of. Bridge forwards data from a network to the other. The bridge functionality added to the Android platform for now is to test our Android AP design, and for the long run is implemented as a solution to connect the WLAN and telephone network. In this project, we are working on a demo system which is running on an x86 laptop instead of a mobile phone. The bridge is to transfer data between the Ethernet interface and the WLAN interface back and forth.

Bridge itself is part of the Linux kernel. When it is necessary in a system it must be enabled in the kernel compiling configuration file and built into the operating system. This is done by `set networking -> 802.1d Ethernet Bridging` to be `y` or `m`. After enabling the bridge support there must be also some utilities in the system for controlling and configuring the bridge such as creating or removing a bridge, adding or removing interfaces into a bridge, and so on. The bridge utilities are not part of the Linux kernel. The source code of such utilities is in a package called `bridge-utils` which can be downloaded from [5]. It's in a `git` repository. One should first install `git` tools in its system and then run the `git` command with its URL parameters given in reference in the console.

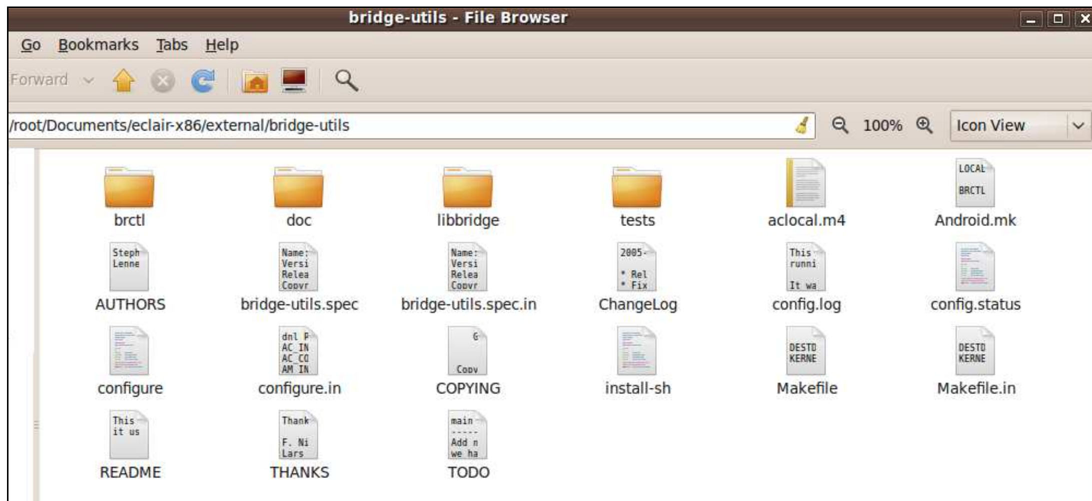

Figure 18 Source code overview of bridge utils at the top

Download the source code of bridge-utils and extract it into a directory `bridge-tuils`. The source code structure of `bridge-utils` looks like Figure 18. In the source code package no `Makfile` can be found directly. Files named `aclocal.m4`, `configure.in`, `configure` are present as well as some other files which together are necessary for a GNU compiling process. The Makefile is the reference we use to write Android style makefile. So the first step to work on the source code is to generate the Makefile. It is as simple as running the configure script in the `bridge-utils/` directory. After running configure script a file named Makefile shows up. With it we write our Android.mk file. Again the Android.mk file is the only makefile for this package in Android source hierarchy. Therefore the Android.mk file is placed at the top of `bridge-utils/` directory and the package `bridge-utils` is placed under the directory `external/`.

`Bridge-utils` is a very small package. The main source code is under subdirectories `brctl` and `libbridge`. The code under `libbridge/` subdirectory is used to build a bridge utility library. And the code in `brctl/` subdirectory is for building the utilities. Therefore in the `Androis.mk` file there are two building tasks, first to build the library and second to use the library together with the source in `brctl/` to build the utilities. Some private defined variables are used in this `Android.mk` file. They are helpful in assisting management of the source code. `BRCTL_BUILD` is used to control when compiling this package. We choose to build a static library instead of a shared library because this allows us to use the bridge utilities independently with the libraries. To build a static library we need to do the routine `include $(BUILD_STATIC_LIBRARY)` and call `include $(BUILD_EXECUTABLE)` to build an executable which is our bridge utilities.

The Android.mk for `bridge-utils` is given as below

```
LOCAL_PATH := $(call my-dir)

BRCTL_BUILD := true

ifndef L_CFLAGS
L_CFLAGS := -O2 -Wall -g
endif

INCLUDES_BRCTL :=
-I/root/Documents/eclair-x86/external/bridge-utils/libbridge
INCLUDES_LIBBRIDGE :=
-I/root/Documents/eclair-x86/external/bridge-utils/libbridge
```

```
COMMON_SOURCES := brctl/brctl_cmd.c brctl/brctl_disp.c
SOURCE_BRCTL := brctl/brctl.c $(COMMON_SOURCES)

LIBBRIDGE_SOURCES := \
    libbridge/libbridge_devif.c \
    libbridge/libbridge_if.c \
    libbridge/libbridge_init.c \
    libbridge/libbridge_misc.c

ifeq ($(BRCTL_BUILD),true)

include $(CLEAR_VARS)
LOCAL_MODULE := libbridge
LOCAL_SRC_FILES := $(LIBBRIDGE_SOURCES)
LOCAL_CFLAGS := $(L_CFLAGS)
LOCAL_C_INCLUDES := $(INCLUDES_LIBBRIDGE)
LOCAL_MODULE_TAGS := user eng development
LOCAL_SHARED_LIBRARIES := libc libcutils
include $(BUILD_STATIC_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MODULE := brctl
LOCAL_SHARED_LIBRARIES := libc libcutils
LOCAL_STATIC_LIBRARIES := libbridge
LOCAL_CFLAGS := $(L_CFLAGS)
LOCAL_SRC_FILES := $(SOURCE_BRCTL)
LOCAL_C_INCLUDES := $(INCLUDES_LIBBRIDGE)
LOCAL_MODULE_TAGS := user eng development
include $(BUILD_EXECUTABLE)

endif
```

```
# brctl
Usage: brctl [commands]
commands:
        addbr           <bridge>                add bridge
        delbr           <bridge>                delete bridge
        addif           <bridge> <device>       add interface to bridge
        delif           <bridge> <device>       delete interface from bridge
        setageing       <bridge> <time>         set ageing time
        setbridgeprio   <bridge> <prio>         set bridge priority
        setfd           <bridge> <time>         set bridge forward delay
        sethello        <bridge> <time>         set hello time
        setmaxage       <bridge> <time>         set max message age
        setpathcost     <bridge> <port> <cost>  set path cost
        setportprio     <bridge> <port> <prio>  set port priority
        show                                    show a list of bridges
        showmacs        <bridge>                show a list of mac addrs
        showstp         <bridge>                show bridge stp info
        stp             <bridge> {on|off}       turn stp on/off
```

Figure 19 successful running of bridge utils in Android

4.5. Wireless card and the driver

Wireless card and its driver take care of the actual signal processing. Currently not many wireless drivers support AP functionalities in their implementation. Even some drivers with clear declaration for supporting AP may still have some problems when working in different combinations of `hostapd` and wireless driver. These issues limit our choices for this project. As the demonstration proceeding we will give the story in which we encounter some exotic bug.

`Hostapd` implements its own driver wrapper for collaborating with different drivers that are claimed to support AP operation mode, including Madwifi driver, `hostapd` driver, Prism54 driver, mac80211 based drivers (like `ath5k` and `ath9k`), and FreeBSD net80211 based driver. Theoretically, all these drivers can work properly with `hostapd`. As we did in section 2.3 we add `libnl` in Android building system. This allows us to use the mac80211 based driver `ath5k` or `ath9k`. In this project we use `ath9k` driver. It comes together with Linux kernel since v2.6.27. This driver works for all Atheros IEEE 802.11n PCI/PCI-Express and AHB WLAN based chipsets. The downloaded Android-x86 source code is packaged with Linux kernel v2.6.29.

To compile the driver `ath9k`, we set the configuration in the kernel compiling configuration to support the driver.

```
Device Drivers  --->
  [*] Network device support  --->
        Wireless LAN  --->
          <M>   Atheros 802.11n wireless cards support
```

As `ath9k` is a mac80211 based drivers, we also need to enable the support for mac80211 in the kernel.

```
Networking  --->
  Wireless  --->
    <M> Improved wireless configuration API
    <M> Generic IEEE 802.11 Networking Stack (mac80211)
```

The configuration here is exactly the same what previously was done in Linux system. The reason for this is that the operating system kernel of Android is just a Linux kernel.

After enabling these configurations the ath9k will then be compiled as a module in the final built Linux kernel. Anytime when the kernel is being built these two modules are being built. The build time for the Linux kernel may vary from several minutes to hours depending on the configuration of the building and the machine that the building is taking place. When Android boots up, it detects the hardware configuration in the system. If it detects a wireless device that uses ath9k module as its driver, the kernel will automatically load this module into the running kernel memory. However there are some cases in which the user needs to add the driver manually with command `insmod` or `modprobe`. Android mostly is built with only `insmod` available. `Insmod` needs the specific path where the module is located. The Android platform implements a terminal simulator in which users can type any command that is currently available in the system to interact with the system. One can use command `lsmod` to check all currently running modules in the memory. If the module `ath9k` is successfully loaded one should be able to see it in the list produced by `lsmod`. `Mac80211` should also be present in the list. Figure 20 shows all current running modules in our experiment.

```
# lsmod
uvcvideo 48684 0 - Live 0xf8221000
videodev 34092 1 uvcvideo, Live 0xf80f2000
snd_hda_intel 22736 3 - Live 0xf806d000
ath9k 263708 0 - Live 0xf88c2000
atl1c 25804 0 - Live 0xf8064000
i915 129412 4 - Live 0xf83c1000
drm 132940 4 i915, Live 0xf8370000
i2c_algo_bit 5012 1 i915, Live 0xf8340000
btusb 10316 0 - Live 0xf8334000
sco 8928 0 - Live 0xf8325000
rfcomm 29848 0 - Live 0xf8311000
bnep 10968 0 - Live 0xf82fb000
l2cap 19400 4 rfcomm,bnep, Live 0xf82ea000
bluetooth 47776 5 btusb,sco,rfcomm,bnep,l2cap, Live 0xf82cf000
bsd_comp 4616 0 - Live 0xf82b5000
ppp_deflate 4040 0 - Live 0xf82ab000
zlib_deflate 17748 1 ppp_deflate, Live 0xf82a0000
ppp_async 7324 0 - Live 0xf8292000
crc_ccitt 1824 1 ppp_async, Live 0xf8289000
pppoe 8988 0 - Live 0xf827e000
pppox 2996 1 pppoe, Live 0xf8271000
ppp_generic 19952 5 bsd_comp,ppp_deflate,ppp_async,pppoe,pppox, Live 0xf8263000
slhc 5084 1 ppp_generic, Live 0xf8255000
option 18824 0 - Live 0xf8247000
usbserial 26204 1 option, Live 0xf8230000
cdc_acm 13460 0 - Live 0xf821b000
eeepc_laptop 13868 0 - Live 0xf820a000
rfkill 9772 5 ath9k,bluetooth,eeepc_laptop, Live 0xf81fd000
snd_hda_codec_realtek 174804 1 - Live 0xf81d0000
snd_hda_codec 57412 2 snd_hda_intel,snd_hda_codec_realtek, Live 0xf8182000
snd_hwdep 6212 1 snd_hda_codec, Live 0xf80d8000
snd_pcm 61420 3 snd_hda_intel,snd_hda_codec, Live 0xf80c2000
snd_timer 17624 1 snd_pcm, Live 0xf805d000
snd_page_alloc 8012 2 snd_hda_intel,snd_pcm, Live 0xf804d000
```

Figure 20 Modules currently running in Android

4.6. Launching hostapd with configuration file

To check whether any component is missing one can simply run `hostapd` from the console. If all components mentioned above have been successfully feed into the Android platform, `hostapd` should be able to be launched with its running configuration file.

This is a simple running configuration file of `hostapd`, named `Android-hostapad.conf`

```
interface=wlan0
driver=nl80211
logger_syslog=-1
logger_syslog_level=2
logger_stdout=-1
logger_stdout_level=2
debug=4
ctrl_interface_group=0
hw_mode=g
ssid=ESSID_Genar
auth_algs=3
eapol_key_index_workaround=0
eap_server=0
channel=6
```

No bridge settings in this configuration file, therefore after running `hostapd` the attached stations cannot get access to the Internet via this AP in the Android platform. This is going to be accomplished in next section about testing. The important entries include `interface` and `driver`. The first entry `interface=wlan0` tells `hostapd` to run on the wireless interface labeled `wlan0`. The second entry `driver=nl80211` tells `hostapd` to use the mac801211 based driver which is `ath9k` in case of this project. There is no authentication needed in this configuration. Any user device with WLAN support can connect to the AP. Figure 21 shows `hostapd` is running with its configuration file.

```
# hostapd /etc/hostapd/hostapdAndroid.conf
Configuration file: /etc/hostapd/hostapdAndroid.conf
Using interface wlan0 with hwaddr 00:25:d3:e3:64:ff and ssid 'MyAndroid'
wlan0: STA 00:22:43:2f:71:bc IEEE 802.11: authenticated
wlan0: STA 00:22:43:2f:71:bc IEEE 802.11: associated (aid 1)
wlan0: STA 00:22:43:2f:71:bc RADIUS: starting accounting session 4BED5546-00000000
wlan0: STA 00:24:ef:2e:36:cc IEEE 802.11: authenticated
wlan0: STA 00:24:ef:2e:36:cc IEEE 802.11: associated (aid 2)
wlan0: STA 00:24:ef:2e:36:cc RADIUS: starting accounting session 4BED5546-00000001
```

Figure 21 `Hostapd` launched with its configuration file in Android

5. Testing

In this chapter we run the test case to show the AP is functioning in the Android platform. To demonstrate the process we begin with the expected results of the testing. Next we give the testing configuration in both hardware and software. With that we set up the testing network. And then we demonstrate the details of the testing process and outcomes from each station.

5.1. Expected results

After the local network is set up, it will do simple packet transmission to show the local network is properly functioning. The Android platform running the AP currently cannot support Ethernet and WLAN at the same time. So to test the AP functionality in the Android platform, we just test the packet transmission within the local network, rather than using a bridge to forward the packet to the Internet.

When the AP (`hostapd`) is running at the Android platform, any device with WLAN support should be able to connect to it and all hosts connected should be able to `ping` each other. In the result part of the testing procedure we will prove the success of `ping` with pictures.

5.2. Testing environment

There are three machines involved in this testing, one for running Android and `hostapd`, and the other two for running as WLAN peripheral stations. We choose an ASUS machine with Windows vista system and a Sony Ericsson phone with proprietary system as the peripheral stations. Such a choice is to show that there is no restrict on the category of host machine provided it has WLAN compatible network support (IEEE802.11b/g). This is important as the goal of this project is to make the AP capable for providing WLAN service without device bias.

Below is the configuration of each of these host machines

— **Eee PC**

Operating system	Android with Linux kernel v2.6.29
Processor	Intel Atom
Wireless card	Atheros ar8132 PCI-e fast Ethernet controller
Wireless driver	ath9k
`Hostapd`	`hostapd` v0.6.9
`Libnl`	`libnl` v1.1

— **ASUS X50Z series**

Operating system	Windows Vista
Processor	AMD Turion(tm) X2 Dual-Core Mobile RM-70
Wireless card	Atheros AR5007EG Wireless Network Adapter
Wireless driver	AR5007EG 7.4.2.57 for Windows Vista

— **Sony Ericsson W715**

Operating system	Sony Ericsson proprietary operating system
Processor	Intel StrongARM
Wireless card	Marvell 88W8686 chipset based card
Wireless driver	libertas

5.3. Network topology

The AP operates in infrastructure mode in which all peripheral stations connect to the AP. The Android platform was designed for handheld device which has the pervasive telephone network access capability. The nice feature for such device is that it can function as modem for other types of devices that have WLAN supported. Therefore we are intended to add this modem capability for the Android platform. In this testing case we let the Android platform machine run as AP while other hosts run as peripheral stations that expect to connect to the AP for service. The network topology is given as Figure 22 below.

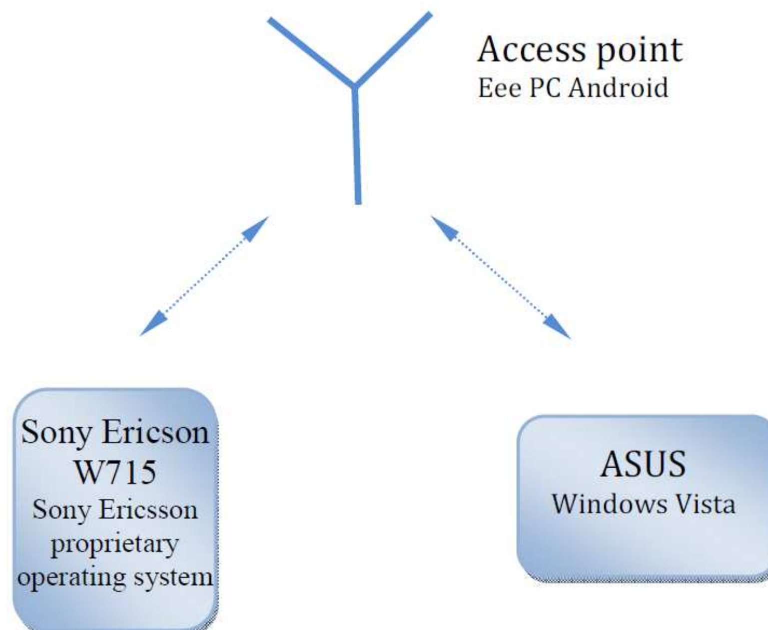

Figure 22 Network topology in the testing case of `hostapd` in Android

5.4. Setting up the network

The process of setting up the network consists of two steps. The first step is to launch `hostapd` with its running configuration file and connect the peripheral stations to the AP. The second step is to do the network configuration for all the hosts.

5.4.1. Running hostapd and connecting the hosts

The Android platform provides both desktop environment and console operation environment for interactions with the user. In this test we use the console for currently we have not implemented a graphic interface for our AP. The graphic interface of `hostapd` is designed in the next chapter. We place

the Android system in a USB stick so that we can run the test on any x86 based machine. With the USB stick we first plug it into the USB interface of the Eee PC and then choose to boot up from USB so that Android can be launched into the memory of the Eee PC machine as Figure 23.

Figure 23 Eee PC boots from the USB stick and Android is running

Next we launch `hostapd` with its configuration file. The content of the configuration file is given below in the gray frame. Figure 24 shows `hostapd` is running.

```
interface=wlan0
bridge=mybr0
driver=nl80211
logger_syslog=-1
logger_syslog_level=2
logger_stdout=-1
logger_stdout_level=2
debug=4
ctrl_interface_group=0
hw_mode=g
ssid=MyAndroid
auth_algs=3
eapol_key_index_workaround=0
eap_server=0
channel=6
```

After `hostapd` is running at the Eee PC, other host machines can connect to the AP with their WLAN interfaces. Since both the ASUS and the Sony Ericsson machines have graphic user interfaces installed, the connection can be performed from corresponding graphic network manager. Figure 24 shows `hostapd` is launched with its configuration file and `hostapd` is running with SSID of `MyAndroid`. Two hosts with MAC addresses `00:22:43:2f:71:bc` and `00:24:ef:2e:36:cc` are authenticated and connected. The accounting is started for each of the connected hosts.

```
# hostapd /etc/hostapd/hostapdAndroid.conf
Configuration file: /etc/hostapd/hostapdAndroid.conf
Using interface wlan0 with hwaddr 00:25:d3:e3:64:ff and ssid 'MyAndroid'
wlan0: STA 00:22:43:2f:71:bc IEEE 802.11: authenticated
wlan0: STA 00:22:43:2f:71:bc IEEE 802.11: associated (aid 1)
wlan0: STA 00:22:43:2f:71:bc RADIUS: starting accounting session 4BED5546-00000000
wlan0: STA 00:24:ef:2e:36:cc IEEE 802.11: authenticated
wlan0: STA 00:24:ef:2e:36:cc IEEE 802.11: associated (aid 2)
wlan0: STA 00:24:ef:2e:36:cc RADIUS: starting accounting session 4BED5546-00000001
```

Figure 24 `Hostapd` is running and stations are connected

5.4.2. Configuration of hosts

After the Android boots up, we run command `netcfg` to see the available interfaces in the system. As seen in Figure 25 there are four interfaces, two of which are up and the other two are shut down. `lo` is the loopback device. `Eth0` is the Ethernet interface. `Wlan0` is the wireless interface. `Wmaster0` is the internal master device used by mac80211. Since we need to use the wireless interface `wlan0`, we need to run the command `netcfg wlan0 up` to bring up the wireless interface.

```
# netcfg
lo       UP    127.0.0.1     255.0.0.0       0x00000049
eth0     UP    0.0.0.0       0.0.0.0         0x00001003
wmaster0 DOWN  0.0.0.0       0.0.0.0         0x00001002
wlan0    DOWN  192.168.1.2   255.255.255.0   0x00001002
```

Figure 25 Available network interfaces in the Android

To make the hosts recognize each other in the same network, we need to configure the address in the same network. We need to do the configuration manually because there is no DHCP server is listening in the local network. We choose the addresses from 192.168.1.2 to 192.168.1.4, with 255.255.255.0 as network mask.

— For the Eee PC machine run commands

  `ifconfig wlan0 192.168.1.2 netmask 255.255.255.0`

— For the ASUS PC machine run commands

  `ifconfig wlan0 192.168.1.3 netmask 255.255.255.0`

— For the Sony Ericson W715 machine do configuration as

  `wlan0 192.168.1.4 netmask 255.255.255.0`

Figure 26 is an example of Sony Ericson W715 phone networking information after configuration.

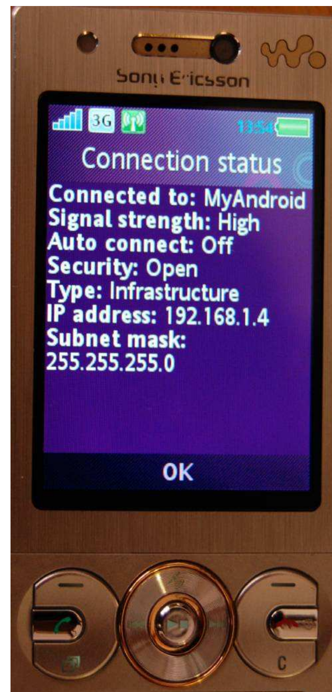

Figure 26 Network address configuration in Sony Ericson W715 mobile phone

5.5. Packets transmission

With the network setup we can do simple packet transmission within the local network. This is achieved by doing the `ping` command from the text console of these hosts. Below are two examples of `ping` other two hosts from the Windows vista host.

```
C:\Users\bmgw>ping 192.168.1.2

Pinging 192.168.1.2 with 32 bytes of data:
Reply from 192.168.1.2: bytes=32 time=7ms TTL=64
Reply from 192.168.1.2: bytes=32 time=4ms TTL=64
Reply from 192.168.1.2: bytes=32 time<1ms TTL=64
Reply from 192.168.1.2: bytes=32 time<1ms TTL=64

Ping statistics for 192.168.1.2:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 7ms, Average = 2ms
```

Figure 27 `ping` Android host from Windows vista

```
C:\Users\bmgw>ping 192.168.1.4

Pinging 192.168.1.4 with 32 bytes of data:
Reply from 192.168.1.4: bytes=32 time=475ms TTL=64
Reply from 192.168.1.4: bytes=32 time=86ms TTL=64
Reply from 192.168.1.4: bytes=32 time=109ms TTL=64
Reply from 192.168.1.4: bytes=32 time=130ms TTL=64

Ping statistics for 192.168.1.4:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss)
Approximate round trip times in milli-seconds:
    Minimum = 86ms, Maximum = 475ms, Average = 200ms
```

Figure 28 `ping` Sony Erricson W715 from Windows vista

In Figure 27, when `ping` the Android host 4 packets, 32 bytes of each, are sent. And 4 packets are sent back from the Android host, with the same size of each responding packet. The same are achieved when `ping` the Sony Ericson W715 mobile phone, which is shown in Figure 28.

5.6. Result analysis

The experiment succeeds in both steps as mentioned in subsection 5.4. First the peripheral stations can discover the AP in the Android platform and all stations are able to connect to it. There is no difference between connection to the AP in the Android platform and connection to a normal hardware AP device. Secondly, the data are successfully transmitted between stations, which means that there is no barrier in the AP in the Android platform for providing data service. From the view of networking layering, we conclude that we succeed in both link layer and application layer.

6. User interface

In this chapter, the focus of our work is transferred from hardware layer to application layer. Our goal is to create a user interface for underlying `hostapd`, which provides the possibility for user to control `hostapd`'s AP functionality from Android application layer, e.g. start/stop `hostapd`, etc. First, we present the overview of Android application framework which provides the basis for our application. Then, we propose two alternatives to implement the user interface for `hostapd`. They are discussed in subsection 6.2 and 6.3 respectively.

6.1. Android application framework

6.1.1. Four essential components

One of the main features of Android is that one application can utilize other applications' elements, if it gets the permission. This feature greatly improves the efficiency of code reuse. Instead of incorporating or linking to a piece of code of another application, the application starts up that piece of code if needed. Therefore, applications in Android differently from in other systems don't have a single entry point for everything in the application, i.e. there is no main() function in Android application. Rather, Android specially provides four essential components which can be instantiated and run by the system. There are four building blocks to an Android application: activities, services, broadcast receivers, and content providers.

- Activity provides visual user interface for Android application. An application may consist of one or more activities. Each one activity is implemented as a subclass that extends the Activity base class. And also each activity is usually a single screen in the application. When the application is launched, one of the activities is marked as the first screen that will be presented to the user. Moving from one screen to another one is accomplished by starting a new activity in the current one. This is usually done by adding a button in the activity to trigger an event. The user interface is displayed through the content of customized screen, which is provided by a hierarchy of views. They are various objects derived from the View base class such as `TextView`, `EditText`, `Button`, etc. The layout of views is written as a specific file in the format of xml.
- A service is code that runs in the background for a long period of time, and without a visual user interface. A good example to explain this is a media player. A service is required in the case where the background music can be kept playing as the user navigates to other activities. Another case is that a service can fetch data from the network and calculate and return the result to the activities that need it [6].
- Broadcast receiver is used to receive and react to an external event. Some broadcasts are initiated by system code informing that the battery is low or the picture has been deleted for instance. And also applications can send broadcast to others to announce that some data over the network is available now. Broadcast receivers do not display a user interface. However, they may display notifications to alert the user if something important has happened. Typically, there will be a special icon shown in the status bar, and users can open to get the message [6].
- Content provider makes it possible for the application to share its data with others. Applications store the data in the file system, in a SQLite database, or in any other ways. The content provider extending the `ContentProvider` base class implements a standard set of methods to let other applications retrieve and store the type of data handled by that content provider [6].

All the four components above are listed in a special xml file called AndroidManifest.xml. Not every application needs to contain all four components, but each application is a combination of those. AndroidManifest.xml declares the components needed by the application and their capabilities.

6.1.2. Intent and intent filter

Additionally, there is an activating component called intent in Android. As mentioned above, of four components only content providers are not activated by asynchronous messages, i.e. intents, rather by a request from `ContentResolver`. Intent is an object of class Intent carrying the message that is transferred among components. It tells the component what kind of action to be taken or what kind of data to be processed. Typical values for action are MAIN, i.e. the entry point of the application, VIEW, PICK, EDIT, etc. The data is defined as a Uniform Resource Indicator (URI). Each type of component is activated by different methods, which are described in [11] as follows.

— An activity is launched by passing an Intent object to the method of `Context.startActivity()`. If a result are expected when the sub-activity exits `Activity.startActivityForResult()` should be used to start a sub-activity.
— A service is started by passing an Intent object to `Context.startService()`. To establish an connection between the calling component and a target service, an Intent object is passed to `Context.bindService()`.
— A broadcast is delivered by passing an Intent object to `Context.sendBroadcast()`, `Context.sendOrderedBroadcast()`, or `Context.sendStickyBroadcast()`.

In each case above, the Android system can find and instantiate the appropriate component (among activity, service and a set of broadcast receivers) in response to the intent, according to the declarations in the `AndroidManifest.xml` file. But in the case that intents do not explicitly name a target component, intent filters mechanism is used to test the intent object in order to find the best component and activate it. A component can have one or more intent filters, which describe what kind of intents the component is capable to handle. Each filter presents a capability of the component. They are defined in `AndroidManifest.xml` file as <intent-filter> elements. When the implicit intent is tested against an intent filter, three fields of an intent filter are consulted: action, data and category. If the implicit intent mismatches any of the fields, it cannot be delivered to the component with that filter.

The most important role of intent is used to switch among applications, activities and services. It functions as an information bridge, which can start a new activity or pass data from one activity to another. Android provides the intent mechanism to navigate from screen to screen. There are two kinds of intent: explicit and implicit. Explicit intent means that the intent receiver is assigned when the Intent object is created. However, implicit intent does not care who will be the intent receiver. For example, an activity calls `startActivity(myIntent)`, and the Android system will check the intent filters for all activities and pick the activity whose intent filters best match `myIntent` to launch it.

6.1.3. Android User interface design

There are two ways in Android to build User interfaces (UI). One way is to write Java code and construct application's UI directly in source code. The other way is non-programmatic, i.e. to define XML-based code file. The second way is highly recommended due to the reason that it is easy to modify without changing the whole source code.

The basic units of user interface in Android application are `view` and `viewgroup`, which are descendants of `View` class. The `View` object serves as a base class for all widgets, which are the elements appearing on the screen (e.g. `TextView`, `Button`, etc). `Viewgroup` serves as a base class for all `laypouts`, which are the layout architecture to order the various widgets. Examples of `viewgroup` are `LinearLayout`, `TableLayout`, `RelativeLayout`, etc.

In Android application, a customized UI is built using a tree-structured of view and `viewgroup` nodes, as shown in Figure 29.

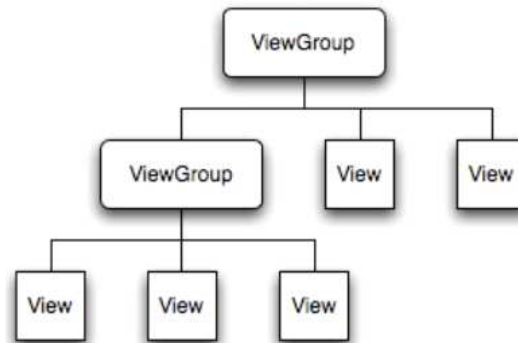

Figure 29 Android tree-structured UI [14]

In order to present the view hierarchy defined in xml file to the screen, the activity calls its `setContentView()` method and pass a reference to the root code of the tree. The Android system parses the elements from the top of the tree. Each `viewgroup` is responsible for its child views, i.e. it requests its child views to draw themselves. Hence, the whole layout is built in order.

Writing an xml file is as easy as creating a HTML file, using a series of nested elements. The advantage of declaring UI in the xml file is to separate the presentation of the application from the behavior of the code. As mentioned above, the `setContentView` method is called when compiling the application. It aims at loading the xml resources from the Java source code. Each view object is assigned an ID by the Android system, which is an attribute defined in xml file and used to uniquely identify the view in the Java code. When an xml file is created in Android project's `res/layout/` directory, the Android system automatically generates an ID in R.Java file for each view object. The method `findViewById()` is used to get the reference from the xml file and assign it to the local variable within the Java code.

6.1.4. The AndroidManifest.xml

The AndroidManifest.xml is required for every Android application and stored in Android project's root directory. If developing Android application with Eclipse (discussed in the subsection 6.1.5), the `AndroidManifest.xml` file is automatically generated instead of writing ourselves. It describes essential information about an Android application, including the four core components mentioned in subsection 6.1.1, their capabilities (e.g. what kind of intent message they are able to handle) and how they can be launched. Additionally, the manifest declares the permissions for both ways. On one hand, an application needs permission to access the protected area of other applications' APIs. On the other hand, the permissions are also declared when other applications want to communicate with the application's components. The permissions are defined in the `<permission>` tag.

The diagram Figure 30 below shows a simple `AndroidManifest.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.example.openfile"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".openfile"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-sdk android:minSdkVersion="6" />
</manifest>
```

Figure 30 A simple AndroidManifest.xml file

It includes a namespace declaration (xmlns:Android="http://schemas.Android.com/apk/res/Android"). The role of this is to include a variety of standard Android attributes and provide the data for the elements in the file [7]. Each manifest has only one `<application>` element, which contains the components used in the application and their capabilities defined in the `<intent-filter>` tag.

6.1.5. Develop with Eclipse

For development Google provides the Android Development Tools (ADT) for eclipse to develop Android applications. It adds powerful functionality to the Eclipse integrated development environment, which provides the convenience for creating and debugging the Android application. In addition, it helps to write Android manifest and layout files in the format of xml, and also it can export the Android project into an executable program in the format of `apk`.

The diagram Figure 31 below shows how to create an Android project in Eclipse. After creating a new Android project, the file directory structure is shown in Figure 32.

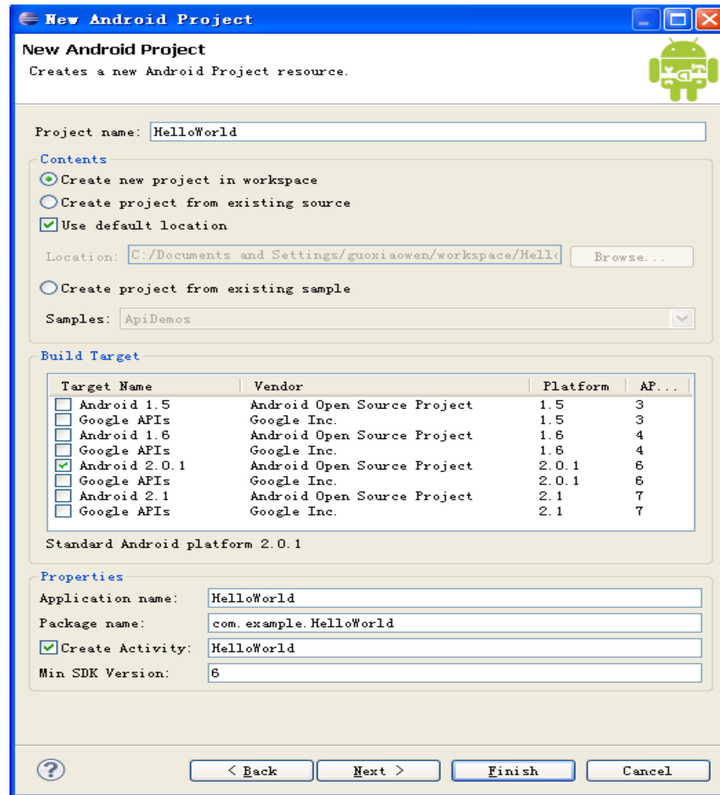

Figure 31 Create an Android project with Eclipse

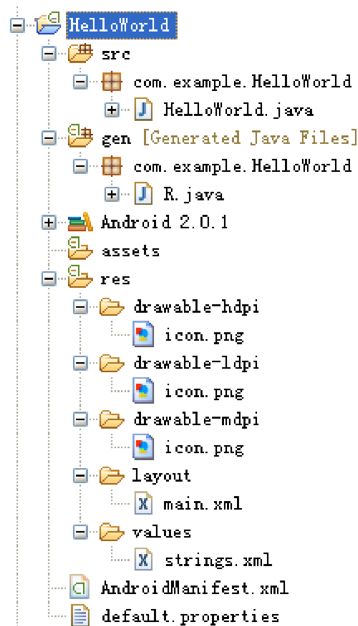

Figure 32 Android project directory structure in Eclipse

R.Java is an automatically generated class file, which contains all the UI elements. The res/drawable directory stores image resources. The res/layout directory stores customized layout files, which contains various view objects and will be present on the screens. The files describing application's variables are located in the res/values directory. All the files under res directory are called resources for Android, which are independent from code file (i.e. Java file). However, the users want to include and reference them in their code behaviors. The main role of R.Java file is to setup a connection between resources and Java codes. The Android application can access the resources using their IDs defined in the R.Java.

The general steps to develop an Android application are the following:

- Step 1: Design one or more layout files, and add UI elements in each layout files according to the requirements. The Figure 33 shows how a layout xml file looks like. This file contains two UI widgets: `TextView` and `Button`. It uses the `LinearLayout` way to vertically order the two view objects. Both UI elements have text attributes, which will be present in the screen result. Moreover, the button element is attached with id attribute, which can be used in the Java code to interact with other code behaviors. Beside the xml-file edit window is a layout window, presenting the result of xml editing, e.g. shown in Figure 34.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="You are in the first Screen"
    />
    <Button
        android:id ="@+id/btnClick"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Open New Screen"
    />
</LinearLayout>
```

Figure 33 A simple layout xml file with two elements

Figure 34 The screen result of Fig. 33

- Step 2: Write Java code for Android application to describe the application behaviors. The four

essential components can be used according to the development requirements. For example, write two class files (`MainActivity` and `SubActivity`) both extending from Activity class; and use intent message implementing the switch between two activities. The Java code can access the resources by referencing resource ID defined in the XML-layout file. For example, a Button object can be created in the Java code in response to an external event through `setOnClickListener()` method.

- Step 3: Write the `AndroidManifest.xml` file to describe all the components used in the application. The manifest has its own writing rules and it is required by the Android system to run the application.
- Step 4: Run the Android project that has been created. There will be at least one Android emulator included in the Eclipse development environment. Various versions of Android emulators can be installed by `Android SDK` and `AVD Manager`. Figure 35 is one example of Android emulator. Figure 36 shows the running result, which is the same as the one designed in the XML-layout shown in Figure 34.

Figure 35 An example of Android emulator

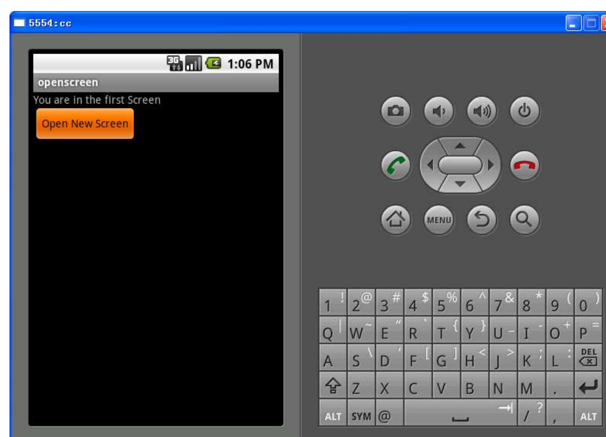

Figure 36 Running result shown in the Android emulator

6.1.6. Important APIs

Android provides a variety of APIs for developers to develop Android application. Those APIs can be used to interact with the underlying Android system. The Android API consists of a core set of packages and classes, and their main definition in [10] are as follows.

— `Android.app` package are high-level classes encapsulating the overall Android application model. The central class is Activity.

`AlertDialog` extends Dialog that is a class of Android.app package. It can be used to display a title, a text message, one or more buttons, etc. And it functions as an external window shown on the application screen. The Figure 37 displays when clicking the button `Msg`, the application pops up an external window.

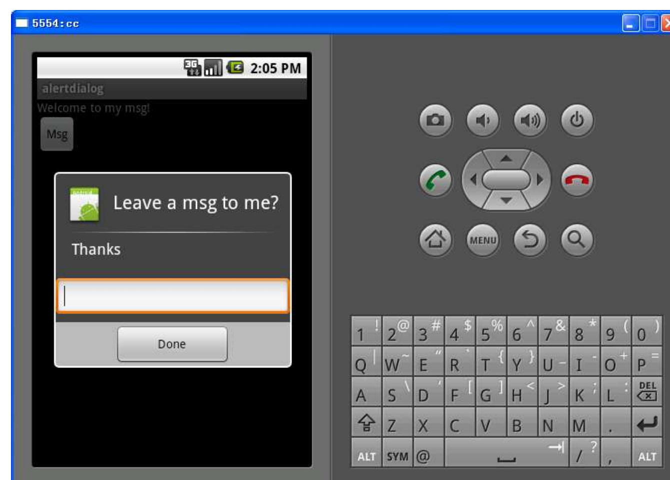

Figure 37 Popup an external window using `AlertDialog` class

— `Android.os` package provides basic operating system services, message passing, and inter-process communication on the device.

Bundle is a class implementing a mapping from String values to various `data/message` types. It is used together with Intent message when passing the data from one activity to the other. It can extract the content of message in String type using its own methods.

— `Android.view` package provides classes that expose basic user interface classes that handle screen layout and interaction with the user.

View class is the most useful in this package. It represents the basic building block for user interface components.

— `Android.content` package contains classes for accessing and publishing data on the device.

`BroadcastReceiver` is a base class for code that will receive intents sent by `sendBroadcast()`.

`ContentProvider` is one of the primary building blocks of Android applications, providing content to applications.

`Intent` is an abstract description of an operation to be performed.

`IntentFilter` class is a structured description of Intent values to be matched.

— `Android.widget` package contains mostly visual UI elements to use on the application screen.

— `Android.util` package provides common utility methods such as `data/time` manipulation, base64 encoders and decoders, string and number conversion methods, and XML utilities.

Log class is the API for sending log output.

6.2. GUI design through hostapd control interface

6.2.1. Hostapd GUI design overview

After compiling `hostapd` in the Android system, `hostapd` can be run and used through its initialized configuration file. But the disadvantage is that all the operation is done by command line interface. To increase the user-interaction and manipulate `hostapd` from upper layer, we supply a friendly Graphical user interface (GUI) design to allow users to control `hostapd` from Android application layer (e.g. `start/stop hostapd`).

GUI for `hostapd` provides an interface for sending network information in the form of broadcast, which is visible for the wireless devices (e.g. laptop, PDA, mobile phone, etc.) to connect with. It is also necessary to implement an authentication with the wireless devices that have the connection requests. It will ask them to provide additional security information (e.g. username and password) to authenticate. The most important role of an AP is to distribute the data among devices.

`Hostapd` GUI should support all of the interactive status and configuration features of the command line client. The way to develop an Android application that is capable of talking to `hostapd` is to directly talk with `hostapd` control interface. `Hostapd` provides a control interface that can be called by external programs to control the `hostapd`. In the source code structure, there includes a small library in the form of C file, i.e. `wpa_ctrl.c`, that provides the link for external programs. All the library functions are documented in `wpa_ctrl.h`, and `hostapd_cli.c` is an example program that uses the library.

There is an obstacle for designing such a `hostapd` GUI. As mentioned above, `hostapd` control interface is written in C programming language. However, an Android application is developed using Java language. To solve this problem, we use Android NDK or JNI development tools, which are able to use the helper functions in C language that `hostapd` provides.

Specifically, `hostapd` GUI have the following main features:

- It starts `hostapd` to send network information (i.e. beacon frame) to other wireless devices.
- It asks for user credentials to authenticate with the requesting device and allow it to connect
- It limits the number of connecting wireless devices
- It adds a wireless device and sets its permission
- It remembers the authenticated devices that are frequently communicated with in order to enable automatic reconnection in the future
- It shows the currently connecting devices with their IDs (e.g. MAC address)
- It shows the `hostapd` configuration file and allows users to edit it
- It displays important connection information, such as SSID, network interface, STA MAC address
- It displays an event history log of messages generated by `hostapd`
- It displays current status, such as authenticated, associated information, etc.
- It distribute/rely the data among wireless devices
- It stops `hostapd`

More details are presented in the following subsections.

6.2.2. Functionality design diagram

In this subsection, first we provide the functionality of `hostapd` GUI presenting with the `hostapd` GUI screen. Then we use the thinking of software engineering to transform the functionalities into design classes. Various UML diagrams are used to create design model and visualize the architecture of GUI development.

6.2.2.1. Hostapd GUI screen design

`Hostapd` GUI divides the functionalities into three important windows: one main window, one setting window and one adding window. Figure 38 shows our `hostapd` GUI main window that is composed of some buttons and text views. It is used for displaying key information of the last connection request, including connection status, network parameters, connecting device's parameters and security information. It can start and stop `hostapd` (i.e. the functionality of AP) through two control buttons. The other buttons are used to trigger the events, e.g. to open other windows (i.e. `edit conf.` button, `setting` button and `add` button) or show the information of currently connecting devices in list views (i.e. `connecting devices` button). The log button gives the whole review of the history log of connections.

Figure 38 `Hostapd` GUI main window

Figure 39 shows how to set the network for an AP. The user can select a network name (SSID), and setup a network interface for other wireless devices to connect through. `Hostapd` GUI also provides some network security features, e.g. using WPA-PSK (WLAN Protected Access-Pre Shared key) for authentication and WEP (Wired Equivalent Privacy)/CCMP (IEEE802.11i encryption protocol) for encryption. The WEP key is customized to authenticate with wireless devices that have connection requests. The authentication window is shown in Figure 40, asking for the username and password for the connection. There are three options for the IEEE802.11 mode (IEEE802.11b/g) setting. There is a checkbox element in the screen that is used to enable auto channel scan, or the user can select the specified channel for transmitting data (e.g. channel 11). Transmission rate can be set to best by default. The user can limit the number of wireless devices simultaneously accessing the AP network. The last

setting uses two radio-buttons to choose the AP network visible for other devices. The bottom line is a save button that can store all the above settings.

Figure 39 Network setting window

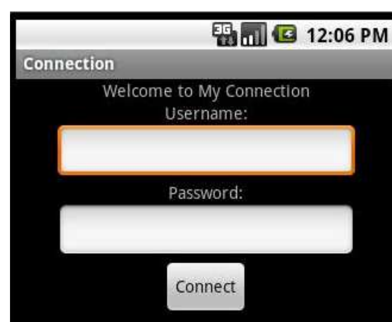

Figure 40 Authentication window

`Hostapd` GUI adding window is used to add or reserve a new wireless device to connect to the AP network, as shown in Figure 41 below. It consists of two parts, adding and reservation list. If the wireless device has the problem with searching for the AP network, the user can manually add this device by filling the computer name, IP address, and MAC address in the field and mark it enabled. The click of save button makes this newly-added device shown in the reservation list. The user can freely edit and delete the wireless devices in the list.

Figure 41 `Hostapd` GUI adding window

Figure 42 is a list view showing the number and information of wireless devices currently connecting to the AP network. The expiration column shows the remaining time for each device when it can be connected. The list provides the possibility for the user to revoke and reserve the devices easily.

Number of dynamic connecting wireless devices: 5

Hardware Address	Assigned IP	Hostname	Expires		
1c:4b:d6:3e:b2:a5	10.10.1.239		1 Days 12 Hours 32 Minutes	Revoke	Reserve
00:1f:3b:6b:01:0d	10.10.1.240	xiaowenguo	Never		
00:0e:35:5d:9e:1e	10.10.1.250	amilo	Never		
00:14:a5:0f:74:7e	10.10.1.253	Basel-PC	2 Days 18 Hours 9 Minutes	Revoke	Reserve
00:08:7b:07:48:d4	10.10.1.254		2 Days 56 Minutes	Revoke	Reserve

Figure 42 List view of currently connecting with the AP

The functionality of editing `hostapd` configuration file is present in the subsection 6.3.

6.2.2.2. Hostapd GUI use case design

The diagrams above show the functionalities of `hostapd` AP in the form of GUI. The aim of `hostapd` GUI is to increase the user experience, allowing users to control `hostapd` functionalities from application layer. From the perspective of designers and developers, the requirements of users are the starting point in software engineering. We use various UML diagrams to create our visual models. UML diagrams represent two different views of a system model: structural and behavioral views. Use case diagram is used to show the functionality provided by a system in terms of three main elements: actors, use cases, and relationships among those use cases. Figure 43 shows the primary functions of an AP [9]: (Distribution system (DS) consisting of a set of basic service sets and integrated local area networks.)

a) Provide DS access for the WDs.

– Includes WD authentication and extends to notifying the DS.

– Includes relaying data between the WDs and the DS.

- Uses a special data relay function called data filtering.

b) Configure the AP

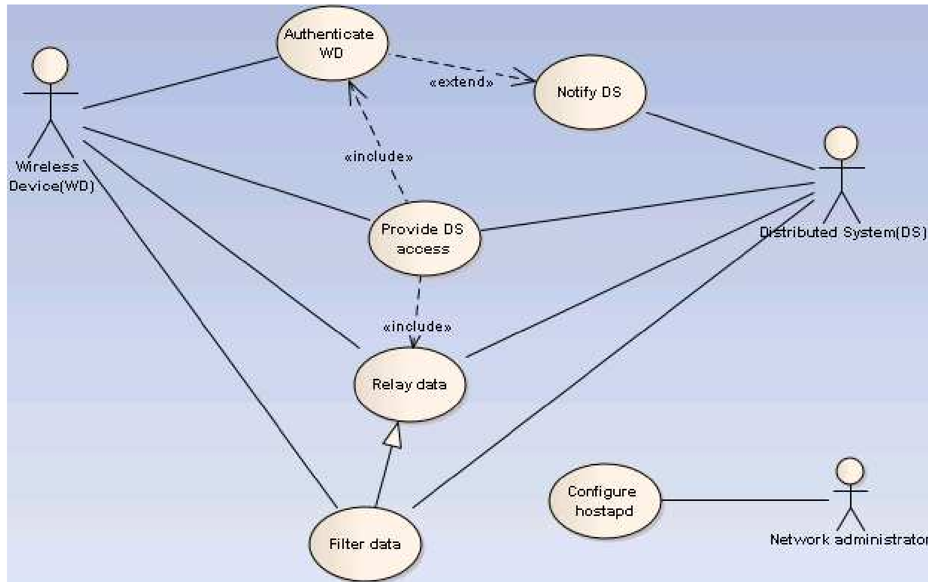

Figure 43 Use case diagram for the AP

In use case diagram (Figure 44), `hostapd` GUI instead of AP is viewed as a whole system. The diagram presents the functionalities of `hostapd` GUI in terms of actors, new use cases, and the relationship between them. The wireless network user through `hostapd` GUI system can start/stop `hostapd`, which extends to sending network information to the WDs. The wireless network user can enter adding window to do the operation for the WDs, including adding and deleting a WD. The user can open a setting window to set the information for the AP. Specifically the user can limit the number of simultaneously connecting WDs. The user can edit (view, edit and save) `hostapd` configuration file through editing window. In addition, the GUI system has three display functions (e.g. showing AP information).

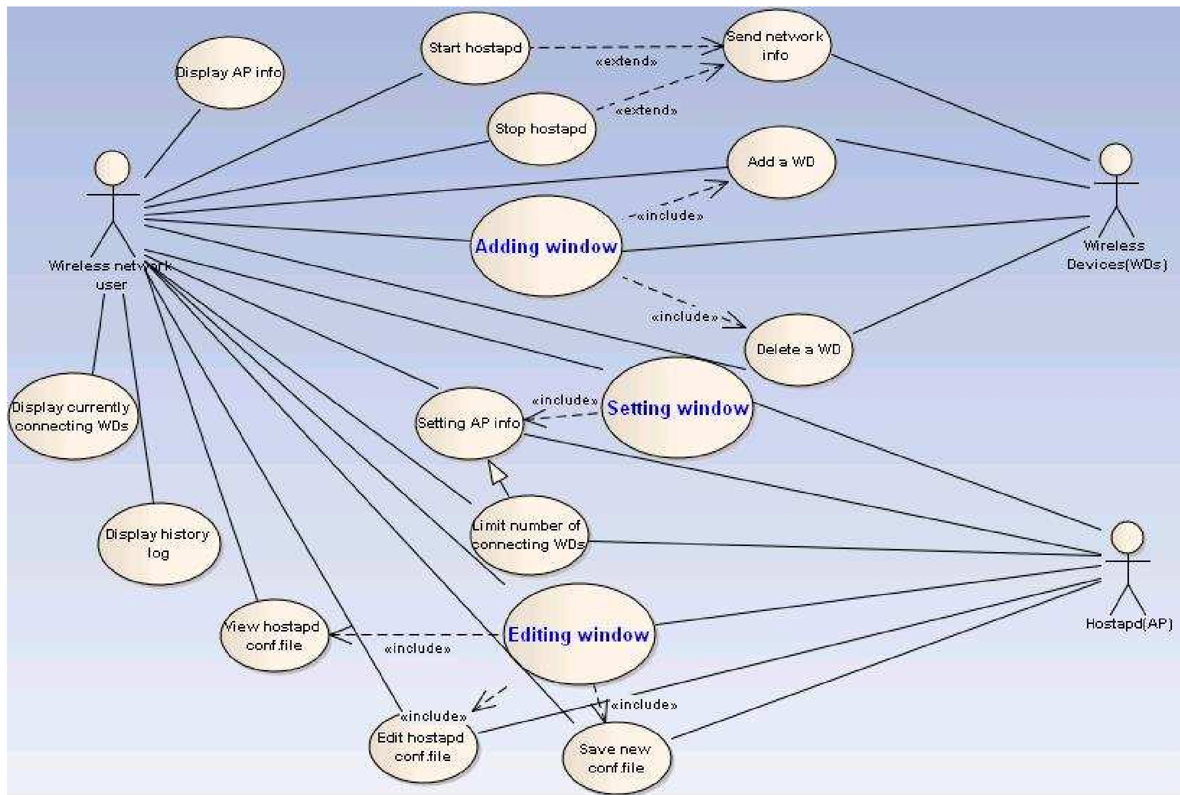

Figure 44 Use case diagram for `hostapd` GUI

To emphasize the dynamic behavior of the system, we also create a sequence diagram shown in Figure 45. The sequence diagram shows how the objects interact with each other in by transferring a sequence of messages. And also it indicates the lifespan of each object. Figure 45 describes how external events communicate with `hostapd` through `hostapd` control interface and how GUI reflects those actions to the wireless network user.

6.2.2.3. Hostapd control interface

In our project we focus on how to control `hostapd` from external program, thus the control interface of `hostapd` is our concerning part. The `hostapd` provides a control interface that external program can utilize by rewriting those functions to control the behaviors of `hostapd`. It is also used to get status information and external event notifications. `Wpa_ctrl.c` file is written in C language, which is a small library providing helper functions to facilitate the use of `hostapd` control interface. Instead of caring about the inter-process communication (IPC), the functions defined in `wpa_ctrl.h` can be used to shield the details of IPC from external programs. External program can use the library functions to communicate with `hostapd`.

`Hostapd` control interface is used to processes two kind of messages: commands and unsolicited event messages. Commands mean the request from external program and the response from `hostapd`. Unsolicited event messages are sent by `hostapd` to the control interface without knowing that which external program will receive the message. The main helper functions defined in `wpa_ctrl.c` as follows [8]:

- `wpa_ctrl_open()`: open a control interface to `hostapd`.

- `wpa_ctrl_request()`: send a command to `hostapd`.
- `wpa_ctrl_attach()`: register the control interface connection as a monitor for `hostapd` events in order to receive the unsolicited messages.
- `wpa_ctrl_pending()`: check whether there are pending event messages available to be received by `wpa_ctrl_recv()`. It is only used for event messages, i.e. `wpa_ctrl_attach()` has been used.
- `wpa_ctrl_recv()`: receive a pending event message from control interface. It is only used for event messages, i.e. `wpa_ctrl_attach()` has been used.
- `wpa_ctrl_detach()`: unregister the control interface connection as a monitor for `hostapd` events.
- `wpa_ctrl_close()`: close a control interface to `hostapd`.

Usually the external program can open two control interface connections to `hostapd` by method `wpa_ctrl_open()`. One is reserved for sending commands to `hostapd`, and the other one is for receiving unsolicited messages from control interface. In other words, `wpa_ctrl_attach()` method is only used when the control interface connection registered is used for receiving event messages.

In Figure 45, wireless network user startup the `hostapd` GUI window and click the start button in the main window to start `hostapd`. A `hostapd` control interface instance is created. Then `hostapd` sends the network information (i.e. beacon frame) to the wireless devices. If the device has a request for interacting with `hostapd`, it opens a control interface to `hostapd` using `wpa_ctrl_open()`. This message in the figure is marked with a star, which means it can be sent more than once. If the device wishes to receive the unsolicited messages, it is required to attach to the control interface by calling `wpa_ctrl_attach()`. Now it registers as an event monitor for the control interface. Then the device sends the command to `hostapd` by `wpa_ctrl_request()`. `Wpa_ctrl_recv()` is only used for receiving the event messages. After a period of transmit time, `wpa_ctrl_pending()` is used to check whether there are pending event messages to be received. By far, a completed communication between `hostapd` and external events through `hostapd` control interface is finished. The user can terminate the `hostapd` by clicking the stop button. After the external program receives the beacon information, it unregisters the control interface to `hostapd` by calling `wpa_ctrl_detach()`. Then it uses `wpa_ctrl_close()` to close the control interface to `hostapd`.

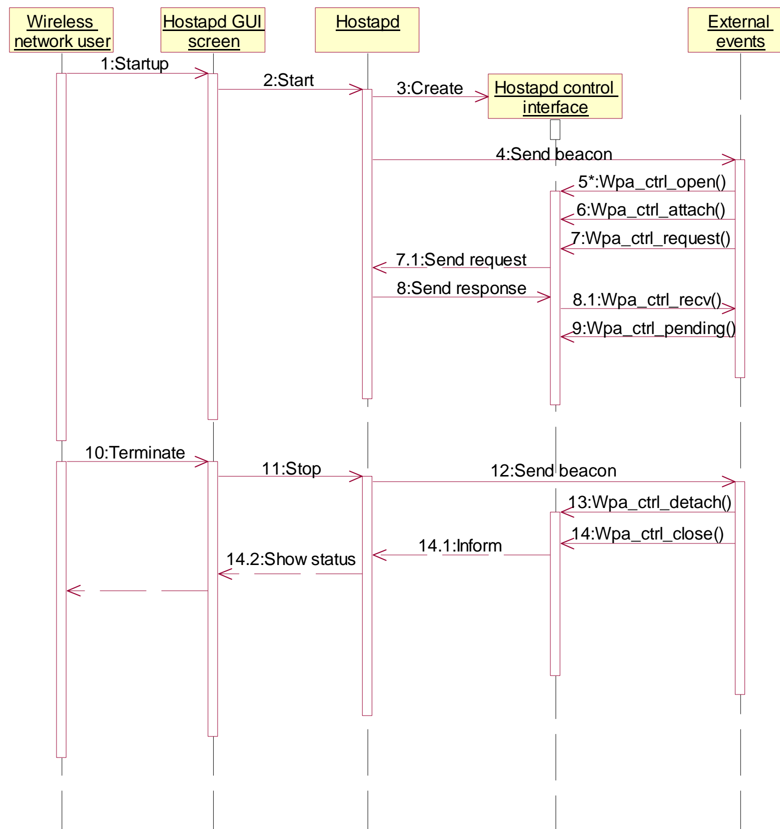

Figure 45 Sequence diagram for `hostapd` control interface

In addition, `hostapd` also defines some commands that can be used together with `wpa_ctrl_request()` function sent to `hostapd`. For example:

- `PING`: This command can be used to test whether `hostapd` is replying to the control interface commands. The expected reply is `PING` if the connection is open and `hostapd` is processing commands.
- `MIB`: Request a list of MIB variables (dot1x, dot11). The output is a text block with each line in `variable=value` format. For example

```
dot11RSNAEnabled=FALSE
dot11RSNAPreauthenticationEnabled=FALSE
dot11RSNAConfigVersion=1
```

- `STA<addr>`: get MIB variables for one station.
- `NEW_STA<addr>`: add a new station.
- `SA_QUERY`: send SA Query request.
- `WPS_PIN`: configure WPS negotiation to act as an integrated WPS Registrar and provision credentials for WPS Enrollees.

- `WPS_PBC`: configure WPS negotiation to act as an integrated WPS Registrar and provision credentials for WPS Enrollees.
- `ALL_STA`: get MIB variables for all stations.
- `HELP`: show this usage help.
- `LICENSE`: show full `hostapd_cli` license.
- `QUIT`: exit `hostapd_cli`.
- `LEVEL`: Change debug level.
- `INTERFACE`: List configured interfaces or select interface, `wlan0`, `eth0`, etc.

The `hostapd` control interface (i.e. library functions defined in `wpa_ctrl.c`) together with its commands provide the basis for implementing the functionalities in the application layer.

6.2.2.4. Hostapd GUI class design

In software engineering, class diagram plays an important role in describing the architecture of a system in terms of classes, attributes and the relationship between the classes. All the functionalities analyzed require to be implemented in the various classes. The class diagram is used both for general conceptual modeling of the functionalities, and for detailed modeling that can be later translated into programming code.

In Android, each application is composed of any combination of the four essential components: activities, services, broadcast receivers, and content providers. More specifically, each class should be implemented extending those components. However, differently each Java application is directly constructed of various functional classes. The basic unit of an Android application is an activity. In Android user interface (UI) design, an activity creates a window to place UI, which contains various widgets such as buttons, text boxes, etc. Figure 46 shows a class diagram for `hostapd` GUI from the perspective of Android UI application.

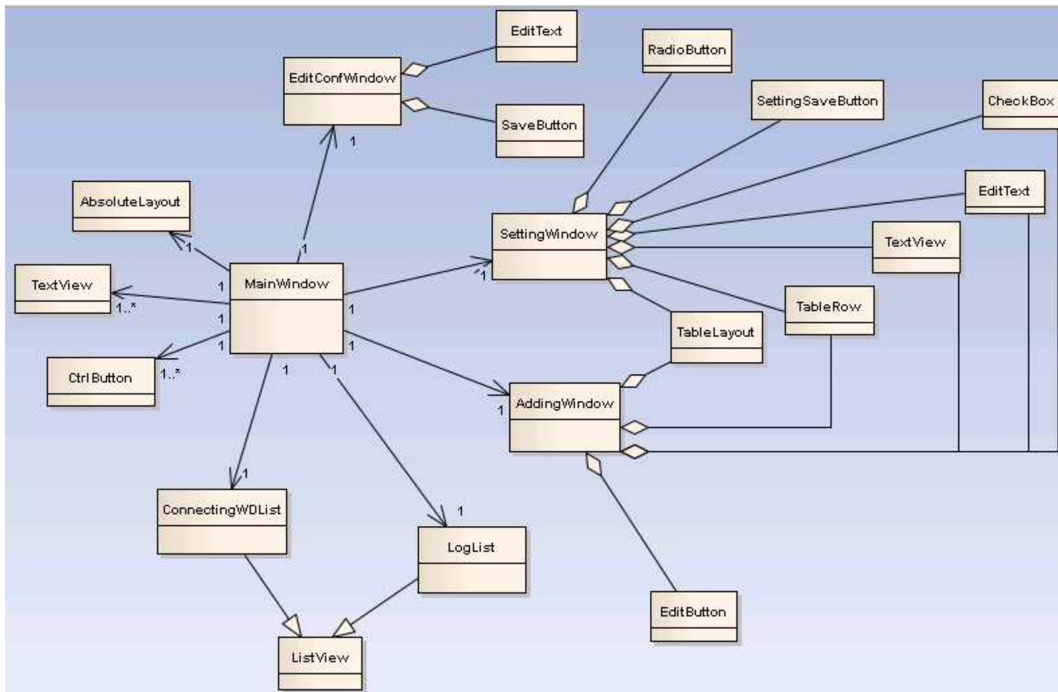

Figure 46 Class diagram for `hostapd` GUI

There are four windows: main window includes the other three ones. Each of them is an activity used to present UI elements. All the elements are extending from the base class view, which is building block for UI elements. Setting window and adding window have some common elements, e.g. `checkbox`, `textView`, and `editText`. Additionally, both windows' elements are ordered in the `tableLayout`. And different buttons are used to operate their respective windows. Main window itself is structured of `textView` and button, which are placed in `absoluteLayout`. It also incorporates two display functions, which are descending from `ListView` class. The following tables (Table 2-4) show the UI elements and their functions implemented in each window.

Setting window as Table 2

Element	Function
TableLayout	Show the content of setting window in the format of table
TableRow	Define a row, used together with TableLayout
TextView	Show the title and required setting parameters of the window
EditText	Show the text box in which the user can edit those parameters
CheckBox	Enable or disable the auto channel scan
RadioButton	Set the AP network visible or invisible
SettingSaveButton	Save all the settings

Table 2 Elements and functions in setting window

Adding window as Table 3

Element	Function
TableLayout	Show the content of adding window in the format of table
TableRow	Define a row, used together with TableLayout
TextView	Show the title and the parameters required to add a new device
EditText	Show the text box in which the user can edit those parameters
CheckBox	Enable or disable the newly added device
EditButton	Save or clear the adding action of a new device

Table 3 Elements and functions in adding window

Main window as Table 4

Element	Function
AbsoluteLayout	Show the elements of main window in any position
TextView	Show the title and display info about the last association
CtrlButton	Start or stop `hostapd`
ConnectingWDList	Show the list of WDs currently connecting with AP network
LogList	Show the history log list of connections
EditConfWindow	View, edit and save the configuration file of `hostapd`
AddingWindow	Add or delete a new wireless device associated with `hostapd` and add it into
SettingWindow	Set the parameters for the connection, both AP network and WD

Table 4 Elements and functions in main window

6.2.3. Implementation approach

To implement the Android application (i.e. `hostapd` GUI), the underlying functions of `hostapd` control interface written in C language are required. The user takes advantage of GUI screen to talk to `hostapd` located in the Android system through `hostapd` control interface. In other words, the implementation requires a bridge providing a transformation between Java language and C language. According to the knowledge so far, Android NDK and JNI are feasible solutions for this.

Android NDK

Google releases Android Native Development Kit (NDK) after Android SDK. Developers rely on Android SDK that is based on Java language to develop their applications. However, Android NDK allows them to build their applications in native code language, such as C/C++. This makes it possible to reuse some existing code without rewriting all the code in Java during application development. Android NDK provides a set of tools, by which C/C++ native libraries are generated. Besides, it allows embedding the generated C/C++ libraries into Android application package files (`.apk`) that can be

deployed on Android devices. And also there integrates a cross compiler in Android NDK, which eliminates the dependency among various platforms. The developer can achieve the cross compiling by modifying the build files included in the Android system. However, there also exists the disadvantage of Android NDK. First, Android NDK is designed for use only in conjunction with Android SDK, because Android application runs in the dalvik virtual machine [12]. Second, it can increase the complexity of an application. Third, it relies on the release version of the platform, that is, the native system libraries that are to be used are not stable and they may change in the future platform version [12].

JNI

The Java Native Interface (JNI) is a Sun-developed interactive technology that is designed to integrate Java code with the code written in other languages, e.g. C/C++. It allows Java program running in a JVM to call the native applications and libraries that are specific to an operating system platform, in order to utilize the system functionality. In the same way, JNI also implements the Java objects to be called by the native method, so that the advantage of Java platform can be maintained.

JNI is a benefit to handle the situation when the standard Java class library does not support the platform-dependent features required by the user-written application. And also it is adapted to the case when there is an existing application or library written in other languages, and the user wish to make it accessible to Java applications.

However, the drawback of using JNI should not be ignored. When an application uses the JNI, it also risks losing two benefits of the Java platform. First, the Java application depending on JNI can no longer be easily deployed to multiple host environments. Even through the part of the application written in Java language is portable to multiple host environments. The reason for that is the part of the application written in native language requires being recompiled [10]. Second, the Java language is type-safe and secure, while native languages such as C/C++ are not. Consequently, extra care must be paid when using JNI to develop an application. Therefore, the developer should construct the application in order that native methods are called in classes as few as possible [10].

JNI developers have provided a set of specific rules to establish the connection between Java code and native methods. From the perspective of developers, a JNI program written in C/C++ is needed to directly manipulate the target native method, and Java application through JVM loading and calling JNI program indirectly calls that native method. Figure 47 shows the call process. When the JVM invokes the native function, it passes a JNI interface pointer (JNIEnv). JNIEnv is passed as argument for each native function mapped to Java method, which allows Java to interact with the native function through JNI environment.

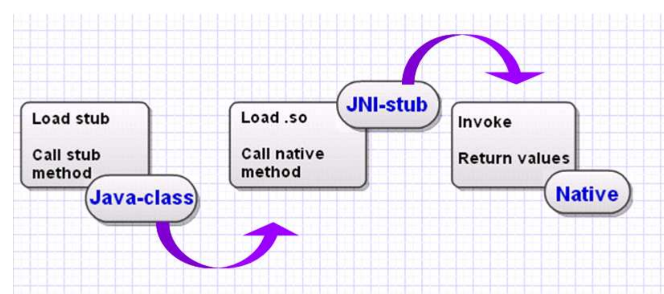

Figure 47 JNI call process

The writing steps of using JNI shown in Figure 48 are as follows (take `HelloWorld` for an example):

1) Create a Java class that declares the native method (`HelloWorld.Java`)

```
public class HelloWorld {
    private native void print();
    public static void main(String[] args) {
        new HelloWorld().print();
    }
    static {
        System.loadLibrary("HelloWorld");
    }
}
```

2) Use `Javac` to compile the source file, resulting in the class file (`HelloWorld.class`)
3) Use `Javah` to generate a C header file (`HelloWorld.h`)
4) Write the C implementation of the native method (`HelloWorld.c`)

```
#include <jni.h>
#include <stdio.h>
#include "HelloWorld.h"
JNIEXPORT void JNICALL Java_HelloWorld_print(JNIEnv *env, jobject obj)
{
    printf("Hello World!\n");
    return;
}
```

5) Compile C code and generate native library (`HelloWorld.dll` or `libHelloWorld.so`)
6) Run the `HelloWorld` program using the Java runtime interpreter. Both the class files (`HelloWorld.class`) and the native library (`HelloWorld.dll` or `libHelloWorld.so`) are loaded at runtime.

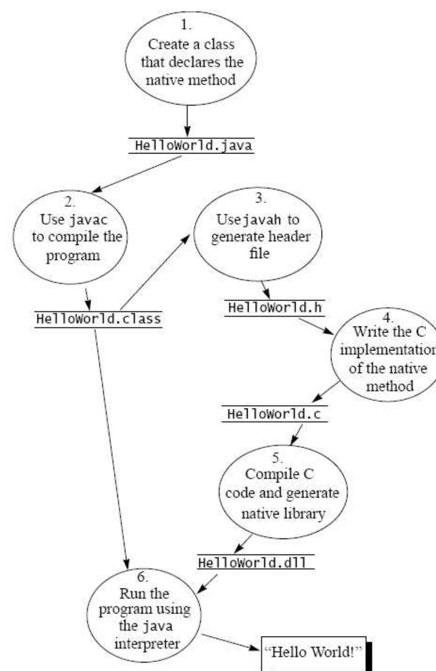

Figure 48 Steps of writing and running `HelloWorld` program [10]

The Android implementation follows the general rules specified in JNI, but there exists some difference: the C implementation of the native method requires an Android makefile (i.e. `Android.mk`) to be built as a shared library (.so file). This file will be called by a Java class with a declaration of the native method through loading the library. And also the Java class file needs another Android.mk to build as an Android package (i.e. .apk) file to be installed in the Android emulator. (.apk file is Android executable program)

As mentioned above, there are two options to implement such an Android application, which requires calling the native methods written in C language. But both of them have the disadvantages which increase the complexity of application development and decrease the benefits of Java platform. Hence, the developer should architect the application before utilizing Android NDK or JNI in order to keep the maximum purity of Java/Android application.

6.3. UI design through hostapd configuration file

This part presents the second alternative to implement user interface for `hostapd`. We first state the concept of design, and then provide the process of implementation in subsection 6.3.2 and 6.3.3. At last, we compare the advantages and disadvantages between two alternatives.

6.3.1. The concept of design

Subsection 6.2 provides an alternative to implement user interface for `hostapd`. We design an Android application called `hostapd` GUI, which encapsulates the functions provided in `hostpad` control interface into various UI elements (`text box, button, label`, etc) and present the friendly UI to the user. In this subsection, we propose a second approach to achieve the task of user interface for `hostapd`.

For the sake of simplicity, we consider to run `hostapd` as a whole instead of directly talking to `hostapd` control interface. In our previous work, we compile and run `hostapd` as an executable application in the Android system. Hence it is possible to develop an Android application with the functionality that it can run another application program. Due to that the Android application program can be run using the `hostapd` command. Therefore, our efforts focus on writing an Android application with the capability of running a command.

In order to increase the communication between user and `hostapd`, i.e. allowing the user to select various functionalities of `hostapd`, we consider that it is a feasible strategy to let the user edit `hostapd` configuration file. Meanwhile, this configuration file is required and will be read when starting the `hostapd`. This file describes the functionalities that need to be loaded together with `hostapd` and it is written in the form of text. Hence the basic idea of our approach is to write an application through which the user can customize the configuration file and save it for running `hostapd`.

In summary, the Android application to be created should achieve two goals:

- The capability to run a command in order to run an application program (i.e. `hostapd`).
- The capabilities to edit a text file so as to customize the configuration file, including view, edit and save basic functions.

We assume that every time the user changes the content of configuration file, which results in restarting the `hostapd` to implement the new functions. The path of configuration file stored will be included in the part that runs `hostapd`.

The basis of our implementation is the Application Program Interfaces (APIs) provided in Android. More specifically, two classes (`Java.lang.Runtime` and `.Process`) can be used for running `hostapd` command and communicating with it. In addition, the operations of reading and writing files in Android will be addressed in details in subsection 6.3.3.

6.3.2. Write an application to run hostapd

According to the APIs provided by the Android system, packet `Java.lang` provides core classes of the Android environment. It includes two classes that can be used to realize the function of running application: Process and Runtime classes. It has an `InterruptedException` which is thrown when a waiting thread is activated before the condition it was waiting for has been satisfied [13].

Class Process represents an external process. It enables writing to, reading from, destroying, and waiting for the external process, as well as querying its exit value [13].

It provides some public methods that are included in our implementation shown in Table 5. (Note: Table 5-11 are listed in Appendix A, which are cited from Android APIs)

Class Runtime allows Java applications to interface with the environment in which they are running. Applications cannot create an instance of this class, but they can get a singleton instance by invoking `getRuntime()` [13].

It provides some public methods that are included in our implementation shown in Table 6.

In addition, we also utilize some classes defined in package Java.io which provide direct access to the file system. They are `InputStream`, `InputStreamReader` and `BufferedReader`. The definitions of them in [13] are the following:

- Class `InputStream` is the base class for all the input streams. An input stream is a way of reading data from a source in a byte-wise manner.
- Class `InputStreamReader` is a class for converting a byte stream into a character stream.
- Class `BufferedReader` wraps an class Reader and increase the function of buffering the input.

The following presents the critical part of our application of running a command:

```
public void execCommand(String command) throws IOException {
    Runtime runtime = Runtime.getRuntime();
    Process proc = runtime.exec(command);

    InputStream inputstream = proc.getInputStream();
    InputStreamReader inputstreamreader = new InputStreamReader(inputstream);
    BufferedReader bufferedreader = new BufferedReader(inputstreamreader);
    …
}
```

The class `Java.lang.Runtime` holds a static method `getRuntime()`, which is the only way to retrieve a reference to the object Runtime. Then the external program can be run by invoking the `exec()` method of Runtime class. By far, we have already started the command we pass as the argument of our self-defined method `execCommand()`. Actually, we start a separate child process (i.e. external program) without the control of parent process. As a result, the output of this child process is not visible. Hence, an input stream (the output of external program is viewed as the input for Java application) is required to obtain the output of the child process after it is run by the command.

Another part of our application shows below the importance of using method `waitFor()`. To see the value that the external process returns, we use the `exitValue()` method of the Process class. But the problem arises when the external process has not yet completed. The `exitValue()` method will throw an `IllegalThreadStateException`. The reason is that the `exitValue()` method is non-blocking.

Hence, the `waitFor()` method is required, which will block waiting until the external process completes. To avoid only using `waitFor()` method to wait an external process that may never complete, we use a if-condition to determine whether the current process should wait or not.

```
try {
 if (proc.waitFor() != 0) {
          System.err.println("exit value = " + proc.exitValue());
 }
} catch (InterruptedException e) {
 System.err.println(e);
}
```

In our application, we use `execCommand("/system/bin/ps");` to test the running result. Figure 49 shows the result for running a command in the emulator. Therefore, this can be applied to the general scenario that we can use this application to execute the `hostapd` command to run `hostapd` application.

Figure 49 Result of running a command

6.3.3. Write an application to edit hostapd configuration file

Due to the security-model of Android, we are not able to use the standard Java file-access method. For example, `FileWriter f = new FileWriter("impossible.txt");` Each *.apk file installed on the emulator or device gets its own user-id from the Android system, which is the key to the sandbox of the application. This sandbox is used to protect the application and its files from other applications that may manipulate the files in a bad manner. Hence, every file that is created is also signed with the user-id of the application. In addition, the developer can set flags (`MODE_WORLD_READABLE` or `MODE_WORLD_WRITEABLE`) to make the file accessible (read or write) for other applications with other user-ids. And the file is written to the following folder on the emulator: `data/data/project_package_name/files/*.txt`. However, the standard Java file-access method can still be applied to the case when a file is written to the SD-card. For example, `FileWriter f = new FileWriter("/sdcard/download/possible.txt");` Android APIs provide various file-access methods that can be applied to our application. The details of feasible methods are presented in the following.

The design steps of our application are as follows:

1. Read the entire asset (i.e. where the file is included) into a local byte buffer.

2. Convert the byte buffer into a string type.
3. Load the string into the `EditText` view.
4. Set a button to capture the event.
5. Edit the file in the `EditText` view.
6. After modification get text to string.
7. Write a file to the disk.
8. Write the string to the file.
9. Ensure that everything is really written out and close.

Referring to the Android APIs, we use `getAssets().open(file.txt)` to retrieve the file that is put or stored in `assets/` directory. With respect to this method, see the description in Table 6.

Class `InputStream` provides some public methods to accomplish step one shown in Table 8.

When loading the string into the `EditText` view, we add a scroller function to view the whole content of the file. Method `setHorizontallyScrolling()`serves as a solution. See the description in Table 9.

We add a button to trigger the event when it is clicked. See the description in Table 10.

At step 7 with respect to the way of opening a file, we use the `openFileOutput()` method that the class Context provides, to protect the file from other applications due to the security reason. See the description in Table 11. We `choose MODE_WORLD_READABLE` flag allowing others to read the file, i.e.`FileOutputStream fOut = openFileOutput("hostapd_conf.txt", MODE_WORLD_READABLE)`. Additionally, we use another two classes provided in the package Java.io `FileOutputStream` and `OutputStreamWriter`.

- `FileOutputStream`, A specialized `OutputStream` that writes to a file in the file system. All write requests made by calling methods in this class are directly forwarded to the equivalent function of the underlying operating system [13].
- `OutputStreamWriter`, A class for turning a character stream into a byte stream. Data written to the target input stream is converted into bytes by either a default or a provided character converter [13]. Some methods are available for us, such as `write()`, `close()`, and `flush()`(i.e. used to ensure everything is really written out).

In our application, the user is able to view, edit and save the changed `hostapd` configuration file. Figure 50 shows the running result on the emulator. The user can easily select the customized functions provided in configuration file by commenting or uncommenting operations.

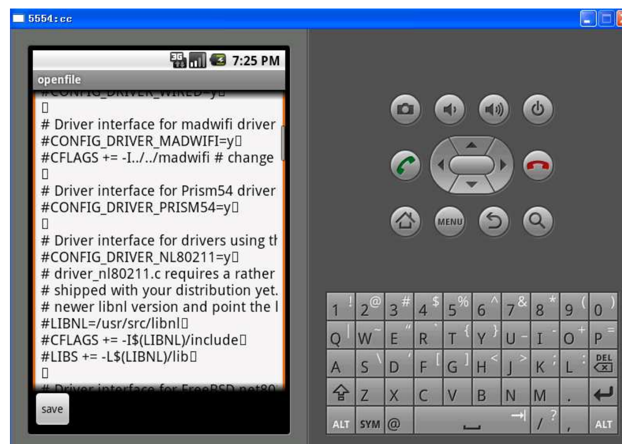

Figure 50 Running result for editing conf file

We use the pull and push functions embedded in the DDMS-Perspective in Eclipse to verify the feasibility of our application. After modification of the configuration file, we pull the file from the device and get the customized file.

6.3.4. Comparison between two designs

In chapter 6, we introduce two different designs of user interface for `hostapd`. One is to encapsulate the functions provided in `hostapd` control interface into functional UI elements, and use Android UI design methods to design a `hostapd` GUI for the user. The other way is to view the `hostapd` as an executable application, and write an Android application to run it. The user experience is accomplished by allowing them to edit the `hostapd` configuration file. Both solutions have advantages and disadvantages. Compared with the second option, the first one has a better expansibility. Due to its realizing some `hostapd` control interface commands in the method `wpa_ctrl_request()`, it can be extended to provide more functionalities of controlling `hostapd`. However, it is not possible in the latter solution. Generally, there is a balance between complexity and functionality. The first alternative supplies more functionality, which increases the complexity of implementation. On the contrary, the second one is easy to actualize but with less functionality.

7. Contributions

The main contributions of this thesis work are three-fold:

Firstly, we have implemented the AP functionality in the Android platform. This function greatly increases the flexibility of network connectivity in real life specifically in the following two aspects: providing access to the Internet and facilitating local WLAN connection. In the first case, an AP mobile phone can act as the Internet gateway for other devices, as long as it is within the coverage of a cellular network. For example in areas with poor wireless network availability or where lack of wired network, the wireless devices connecting with the AP mobile phone can access to the Internet through cellular network, e.g. GPRS, 3G, LTE etc. In the second case, the AP can interconnect with other WLAN supported devices within the same local area, so that they can exchange data between each other and enjoy local services. A good example can be that a laptop uses a printer nearby through the mobile phone AP.

Secondly, our thesis provides a systematic user manual on the Android development documentation, which is valuable for both individual developers and also the Android community as a whole. In this thesis, we well document the process of the whole development, which can give others good guidelines and advice. At the same time, our work will be freely available to Ericsson, which we hope will benefit them as well.

Thirdly, the results achieved from this thesis work may trigger potential business opportunities, since they provide the benefit of reducing the cost on purchasing and deploying extra APs and may generate additional revenues for operators when more devices are connected to the Internet.

In addition to the above major contributions, there are also a few minor contributions. Through this work, we are willing to share our experience on how to develop an Android application program, which is different from java applications, with other developers. Furthermore, we discovered a bug in `hostapd` v0.6.10, which is the incompatibility between `hostapd` v0610 and the `ath9k` driver. We suggest using the `hostapd` version 0.6.9 as we have succeeded in having it work together with the `ath9k` driver.

8. Discussions

In this chapter, we provide discussions on lessons learnt through this thesis work, corresponding to the four particular problems mentioned in subsection 1.2. The purpose of including this chapter is to share our experience with other developers who may work in this direction.

8.1. How to run hostapd in the Linux system

This part provides the basis and functions as a starting point for our following work on Android. The crucial work in this part is to make all the elements match with each other, including the WLAN chip/driver, `hostapd` and its software dependencies. There is some relevant information of this part available on the Internet. However, due to the variety of drivers and the requirements of software versions, we have struggled many times to get the finally satisfied results.

Initially we selected the Madwifi driver to work together with `hostapd` in AP mode. But when we tried to compile `hostapd`, we discovered the absence of `libnl` that is one of the necessary software utilities. Later on, we found the other software element required by `hostapd` is `openssl`. Other than building and installing the `openssl`, we need to add its dynamic library files into the `hostapd` Makefile, otherwise errors occur with the `hostapd` link process.

In order to test `hostapd`'s AP function, we used `wlanconfig` wireless tool to configure the network interfaces (by default the wired and wireless interfaces are `ath0` and `wifi0`). We met the problem that those two interfaces disappear in the interfaces list, instead of two new interfaces `eth0` and `wlan0` show up. Thus we suspect there might exist another driver in the system that is also running. To verify our suspicion, we uninstalled the Madwifi driver, while the wireless network was still available. As a result, we need to solve the conflict problem resulting from two running drivers, in order to control the driver we need. We found the other driver `ath5k` that affects our Madwifi, and attempted to disable it. Owing to the fact that the driver is embedded in the Linux system kernel, it is not easy to remove it without changing kernel configuration. We have to change the solution using `ath5k` instead because it also supports AP mode.

The limited time does not allow us to redo everything from the scratch, including compiling `libnl`, `openssl` and `hostapd`. Thus we use the compiled packages of them and run `hostapd` working in our expected way, i.e. the PC with `hostapd` installed can provide AP network. This part of work implicates the possible phenomena later occurring in the Android system. The dependencies that `hostapd` requires are portable to the Android platform. It also provides other developers who are aiming at the same goal a good reference.

8.2. How to understand the Android building system

The main outcome of this part is to know the build discrepancy between Linux and Android. The Android building system is a customized system. Every function such as `hostapd`, `libnl`, etc can be viewed as an external package to the Android platform. Developers can assign which packages are to be built and which are not. The goal of building is to generate the executable files of a program from the program's source files, in order to later install the program. In addition, each of external packages needs a specific makefile to tell the building system how to build a particular application. We have to write runnable makefiles ourselves for `hostapd` and `libnl` compiling. Recompiling can be carried out in two ways: one is to stand at the top of Android directory and specify the package name that requires compiling; the other is to compile in the directory where the package is located. There is no difference in efficiency between them.

8.3. How to port hostapd from Linux to the Android system

At the porting phase, according to the experiment in Linux we need to recompile `hostapd` and `libnl`. `Openssl` is included in Android source code structure, so what we need is to inform the path of `openssl` to `hostapd`. We choose `ath5k` driver that is provided in the Android system kernel. Thus we downloaded the Android kernel and compiled it to incorporate `ath5k` driver.

It turned out to be a tricky part when we compiled `hostapd`. It showed strange errors that we referred to some undefined variables. We have not found any better solution but to include the absolute path of some header files in the error files. We ended up with compiling the Android system into an image file suffix with .img. Additionally, we design to store this image Android system into a pluggable flash card, so as to facilitate various tests on the system that is independent from the machine.

Initially, the WLAN card we use is a PCI card that is limited for desktop computer. To deploy the card to laptop and carry out tests without place restriction, we have to change to a PCMCIA card. We were stuck in the process of running `hostapd`, because we got errors showing that the `hostapd` could not find the `ath5k` driver. We used `hostapd` command with option to pull out the error log, but without any finding. There is almost no relevant information that can be referred by far.

The solution we retrieved is to consult to the previous experience in Linux circumstance. The reason might be the presence of another driver affecting our driver's behavior. We checked the installed module lists and found the absence of `ath5k` driver, meanwhile we discovered a driver called `iwlagn` is running. That is where the problem is located. We incorrectly assume the existence of `ath5k` because it is included in Android kernel. To fix this two-driver problem, we need to disable one of them leaving the other that is AP compatible. To avoid the complexity, instead of using the extra WLAN card, our final solution is to use the card coming with the laptop itself and the `ath9k` driver that supports for the card. `Ath9k` is newer than `ath5k`, but the driver requirements of `hostapd` should be enough provided in `ath5k`. The reason for replacing `ath5k` with `ath9k` is not clear yet, which could be an open issue for the further work.

Our results confirm that an ordinary client installed with the Android system is able to provide AP function. We do not deploy `hostapd` on the real Android phone in our project. However, the deployment on the platform does not affect the result of AP function we have achieved. There is possibility to adjust the configuration for hardware to satisfy the mobile phone based on our experience. It is important to note also that there can be a different solution to implement such a function. The function is tested by using two different clients, a laptop and a WLAN supported mobile phone to associate with the AP network.

8.4. How to control hostapd from the application layer

To enhance the user experience, we also include the user interface part in our thesis. There are two options in front of us when interacting with `hostapd`. One alternative is to make use of the control interface implemented in `hostapd` so as to control it. In this case, we consult to `wpa_supplicant` which has the common control interface library as `hostapd`, but lacks support of working in a master mode. We encapsulate the basic library functions to better provide the functionalities we design. We have designed several use cases. They are designed based on what functionality an AP could support to do currently, and combined with what special command the `hostapd` control interface could provide.

The other alternative is to view `hostapd` as an application program to run it instead of dealing with its control interface to talk to it. Meanwhile, the implementation of functionality is done through selecting different function entries in the `hostapd` configuration file. In this case, we have two `hostapd` configuration files. One is for initially running `hostapd`, in which we set only the necessary information. The other one is for building `hostapd` later, in which user can edit, choose the required functionality and

generate his/her customized build file. This solution is smart, which breaks the routine of user interface design. And it is also flexible, because user can decide the minimum or maximum functionality.

Those two options above are both the representative of user interface implementation. As common, they are both Android applications and provide a kind of graphic user interface to some degree. Besides, they are both based on the concept of Android view design and utilize the methods resided. Compared with the latter one, the former has some advantages. First, the user interface has more interactive elements, such as button, checkbox, etc. and various layouts with colorful text views. Second, the functionalities are divided into several windows to present. It seems that it is possible to extend the functionalities that are provided currently. However, it is limited by the commands that can be provided by the control interface. This could be another open issue for the future.

As we considered the implementation complexity and the current need for controlling `hostapd` from the graphic user interface, we chose to implement the user interface in the simpler way which was the second solution. But we also gave the detailed design of the first solution, including the use case diagram, class diagram and class description. With our implementation we provide all possibilities for the user to control the AP (`hostapd`), just like what they can achieve on a normal Windows machine.

9. Conclusions and future work

This thesis project is initiated based on the idea of whether an ordinary mobile phone can function as AP, as an enhancement to the function of a station's role in current WLANs. Our goal is to implement that an ordinary station can function as an AP as a replacement of a separate AP device in a wireless network. Meanwhile, it can provide networking services for other devices within the same group such as desktop, laptop, mobile phone, PDA, or other facilities that support WLAN network. This objective should be achieved under the Android system environment.

Given the fact that there are few relevant activities within this topic, the significance of our result is that it provides an innovative contribution to the development of the Android system. Our work covers from the underlying operating system up to the upper layer application, and the development task is performed from the scratch. It requires the knowledge from both hardware design and software design.

We start from Linux circumstance to make `hostapd` run with a WLAN card that supports AP function. This is a good starting point since Android is based on Linux, and combined Linux system kernel with Android unique dalvik virtual machine. We stress the importance of deep understanding to the Android building system, due to its vital role in porting problem. In our work, we disclose the difference between Android and Linux building system in terms of the build process, how to add an external package, etc. We succeed in porting `hostapd` to the Android system and implement the AP functionality in an ordinary station role. In addition, we provide user interface for `hostapd` application so that user can control `hostapd` service through the interface. All in all, the goal of this Master thesis project has been fulfilled with the result that our Android laptop can be used as an AP.

Our results confirm that it is feasible to make an ordinary wireless-support device provide AP functionality. It is important to note that this work can be extended to the deployment in Android phone. Based on our experience, due to the difference in hardware requirements between phone and computer, different configuration of processor and WLAN card for example are necessary. Furthermore, our result constitutes a positive and encouraging step as part of the efforts in this direction for the Android development team at Ericsson.

As possible future work, firstly we can implement what we have designed for the other alternative of user interface in coding. Secondly, our AP functionality could be provided more tests through the bridge between the wired and wireless network. Specifically, the user experience of other devices that connect with the AP should be included. As we have mentioned earlier, it is possible to expand the functionality of user interface of `hostapd`; besides, how to migrate this AP functionality to Android phone will also a further step towards commercial success. Furthermore, other than an AP `hostapd` is also an authenticator, whose security functions included can be implemented as well. Last but not least, the WLAN direct standard currently is becoming a hotspot, which generates a keen interest of WLAN developers. It aims at providing wireless devices peer-to-peer connectivity. Therefore, future work in the direction of WLAN P2P is worthwhile.

References

[1] Hostapd reference, http://hostap.epitest.fi/wpa_supplicant/devel/

[2] Hostapd reference, http://hostap.epitest.fi/wpa_supplicant/devel/hostapd_ctrl_iface_page.html

[3] Android git repository, git://Android.git.kernel.org/platform/manifest.git

[4] Android source code, http://source.Android.com/download

[5] bridge-utils source code, git://git.kernel.org/pub/scm/linux/kernel/git/shemminger/bridge-utils

[6] Developer reference, http://developer.Android.com/intl/zh-CN/guide/topics/fundamentals.html

[7] Android book, available from http://andbook.anddev.org/

[8] Developers' documentation, http://hostap.epitest.fi/wpa_supplicant/devel/wpa__ctrl_8h.html

[9] IEEE 802.11 WP, "Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications", IEEE Standard for Information technology-Telecommunications and information exchange between systems-Local and metropolitan area networks-Specific requirements, Jun 2007

[10] The Java Native Interface-Programmer's Guide and Specification, release 1999, http://java.sun.com/docs/books/jni/

[11] Android intents, http://developer.Android.com/intl/zh-CN/guide/topics/intents/intents-filters.html

[12] Android NDK, http://developer.Android.com/intl/zh-CN/sdk/ndk/1.5_r1/index.html

[13] Android APIs, http://developer.Android.com/intl/zh-CN/reference/packages.html

[14] Android user interface, http://developer.Android.com/intl/zh-CN/guide/topics/ui/index.html

[15] Linux kernel, git://git.android-x86.org/kernel/common.git

Appendix A

Method type	Method name	Method description
abstract InputStream	getInputStream()	Returns an input stream that is connected to the standard output stream (`stdout`) of the native process represented by this object.
abstract int	exitValue ()	Returns the exit value of the native process represented by this object. It is available only when the native process has terminated.
abstract int	waitFor()	Causes the calling thread to wait for the native process associated with this object to finish executing.

Table 5 Methods and method description in class `Process`

Method type	Method name	Method description
Process	Exec(String prog)	Executes the specified program in a separate native process. The new process inherits the environment of the caller. prog: the name of the program to execute
static Runtime	getRuntime()	Returns the single Runtime instance.

Table 6 Methods and method description in class `Runtime`

	Package	Class	Method
Name	Android.content.res	resources	final AssetManager getAssets()
Description		Class for accessing an application's resources. This sits on top of the asset manager of the application (accessible through getAssets()) and provides a higher-level API for getting typed data from the assets.	Retrieve underlying AssetManager storage for these resources.

Table 7 Description of method `getAssets()`

Method type	Method name	Method description
int	available()	Returns the number of bytes that are available before this stream will block. This implementation always returns 0. Subclasses should override and indicate the correct number of bytes available.
void	close()	Closes this stream. Concrete implementations of this class should free any resources during close.
int	read(byte[]b)	Reads bytes from this stream and stores them in the byte array b. b: the byte array in which to store the bytes read.

Table 8 Methods and method description in class `InputStream`

	Package	Class	Method	Related XML attributes
Name	Android.widget	textview	Void setHorizontallyScrolling(boolean whether)	Android:scrollHorizontally
Description			Sets whether the text should be allowed to be wider than the View is. If false, it will be wrapped to the width of the View.	Whether the text is allowed to be wider than the view (and therefore can be scrolled horizontally). Must be a boolean value, either "true" or "false".

Table 9 Description of method `setHorizontallyScrolling()`

	Package	Class	Method
Name	Android.view	view	void setOnClickListener (View.OnClickListener)
Description			Register a callback to be invoked when this view is clicked. If this view is not clickable, it becomes clickable.

Table 10Description of method `setOnClickListener()`

Name	Description
Android.content (package)	
Context (class)	Parent class for Activity. It allows access to application-specific resources and classes, as well as up-calls for application-level operations such as launching activities, broadcasting and receiving intents, etc.
abstract FileOutputStream openFileOutput(String name, int mode) (methods)	Open a private file associated with this Context's application package for writing (openFileInput for reading). Creates the file if it doesn't already exist
notes	mode: Operating mode. Use 0 or MODE_PRIVATE for the default operation, MODE_APPEND to append to an existing file, MODE_WORLD_READABLE and MODE_WORLD_WRITEABLE to control permissions.

Table 11 Description of method `openFileOutput()`

Appendix B Android.mk of hostapd

```
LOCAL_PATH := $(call my-dir)
######## defind BOARD_HOSTAPD_DRIVER is defined
in ../eclair-x86/vendor/htc/dream-open/BoardConfig.mk   do change this
HOSTAPD_BUILD := true
ifneq ($(TARGET_SIMULATOR),true)
  ifneq ($(BOARD_HOSTAPD_DRIVER),)
    HOSTAPD_BUILD := true
    CONFIG_DRIVER_$(BOARD_HOSTAPD_DRIVER) = y
  endif
endif


ifndef L_CFLAGS
L_CFLAGS = -MMD -O2 -Wall -g
endif

# define HOSTAPD_DUMP_STATE to include SIGUSR1 handler for dumping state to
# a file (undefine it, if you want to save in binary size)
L_CFLAGS += -DHOSTAPD_DUMP_STATE

L_CFLAGS += -Isrc
L_CFLAGS += -Isrc/crypto
L_CFLAGS += -Isrc/utils
L_CFLAGS += -Isrc/common

# Uncomment following line and set the path to your kernel tree include
# directory if your C library does not include all header files.
# CFLAGS += -DUSE_KERNEL_HEADERS -I/usr/src/linux/include

include /root/Documents/eclair-x86/external/hostapd/hostapd/.config


# To force sizeof(enum) = 4
ifeq ($(TARGET_ARCH),arm)
L_CFLAGS += -mabi=aapcs-linux
endif


INCLUDES = /root/Documents/eclair-x86/external/hostapd/src/utils
/root/Documents/eclair-x86/external/hostapd/src/common
/root/Documents/eclair-x86/external/hostapd/src/drivers
/root/Documents/eclair-x86/external/hostapd/src/crypto
/root/Documents/eclair-x86/external/hostapd/src/eap_common
/root/Documents/eclair-x86/external/hostapd/src/eapol_supp
/root/Documents/eclair-x86/external/hostapd/src/eap_peer
/root/Documents/eclair-x86/external/hostapd/src/eap_server
/root/Documents/eclair-x86/external/hostapd/src/hlr_auc_gw
/root/Documents/eclair-x86/external/openssl/include
/root/Documents/eclair-x86/frameworks/base/cmds/keystore

ifdef CONFIG_DRIVER_NL80211
INCLUDES += /root/Documents/eclair-x86/external/libnl/include
INCLUDES += /root/Documents/eclair-x86/external/libnl/include/linux
INCLUDES += /root/Documents/eclair-x86/external/libnl/include/linux/netfilter
INCLUDES += /root/Documents/eclair-x86/external/libnl/include/netlink
INCLUDES += /root/Documents/eclair-x86/external/libnl/include/netlink/fib_lookup
INCLUDES += /root/Documents/eclair-x86/external/libnl/include/netlink/genl
INCLUDES += /root/Documents/eclair-x86/external/libnl/include/netlink/netfilter
INCLUDES += /root/Documents/eclair-x86/external/libnl/include/netlink/route
INCLUDES += /root/Documents/eclair-x86/external/libnl/include/netlink/route/cls
INCLUDES += /root/Documents/eclair-x86/external/libnl/include/netlink/route/link
INCLUDES += /root/Documents/eclair-x86/external/libnl/include/netlink/route/sch
```

```
LIBS += -L/root/Documents/eclair-x86/external/libnl/lib
endif


ifndef CONFIG_OS
ifdef CONFIG_NATIVE_WINDOWS
CONFIG_OS=win32
else
CONFIG_OS=unix
endif
endif

ifeq ($(CONFIG_OS), internal)
L_CFLAGS += -DOS_NO_C_LIB_DEFINES
endif

ifdef CONFIG_NATIVE_WINDOWS
L_CFLAGS += -DCONFIG_NATIVE_WINDOWS
LIBS += -lws2_32
endif

OBJS =  hostapd/hostapd.c hostapd/ieee802_1x.c hostapd/eapol_sm.c \
    hostapd/ieee802_11.c hostapd/config.c hostapd/ieee802_11_auth.c
hostapd/accounting.c \
    hostapd/sta_info.c hostapd/wpa.c hostapd/ctrl_iface.c \
    hostapd/drivers.c hostapd/preauth.c hostapd/pmksa_cache.c hostapd/beacon.c \
    hostapd/hw_features.c hostapd/wme.c hostapd/ap_list.c \
    hostapd/mlme.c hostapd/vlan_init.c hostapd/wpa_auth_ie.c

OBJS += src/utils/eloop.c
OBJS += src/utils/common.c
OBJS += src/utils/wpa_debug.c
OBJS += src/utils/wpabuf.c
OBJS += src/utils/os_$(CONFIG_OS).c
OBJS += src/utils/ip_addr.c

OBJS += src/common/ieee802_11_common.c
OBJS += src/common/wpa_common.c

OBJS += src/radius/radius.c
OBJS += src/radius/radius_client.c

OBJS += src/crypto/md5.c
OBJS += src/crypto/rc4.c
OBJS += src/crypto/md4.c
OBJS += src/crypto/sha1.c
OBJS += src/crypto/des.c
OBJS += src/crypto/aes_wrap.c
OBJS += src/crypto/aes.c

HOBJS=src/hlr_auc_gw/hlr_auc_gw.c src/utils/common.c src/utils/wpa_debug.c
src/utils/os_$(CONFIG_OS).c src/hlr_auc_gw/milenage.c src/crypto/aes_wrap.c
src/crypto/aes.c

L_CFLAGS += -DCONFIG_CTRL_IFACE -DCONFIG_CTRL_IFACE_UNIX

ifdef CONFIG_IAPP
L_CFLAGS += -DCONFIG_IAPP
OBJS += hostapd/iapp.c
endif

ifdef CONFIG_RSN_PREAUTH
L_CFLAGS += -DCONFIG_RSN_PREAUTH
CONFIG_L2_PACKET=y
endif

ifdef CONFIG_PEERKEY
L_CFLAGS += -DCONFIG_PEERKEY
OBJS += hostapd/peerkey.c
endif
```

```
ifdef CONFIG_IEEE 80211W
L_CFLAGS += -DCONFIG_IEEE 80211W
NEED_SHA256=y
endif

ifdef CONFIG_IEEE 80211R
L_CFLAGS += -DCONFIG_IEEE 80211R
OBJS += hostapd/wpa_ft.c
NEED_SHA256=y
endif

ifdef CONFIG_IEEE 80211N
L_CFLAGS += -DCONFIG_IEEE 80211N
endif

ifdef CONFIG_DRIVER_HOSTAP
L_CFLAGS += -DCONFIG_DRIVER_HOSTAP
OBJS += hostapd/driver_hostap.c
endif

ifdef CONFIG_DRIVER_WIRED
L_CFLAGS += -DCONFIG_DRIVER_WIRED
OBJS += hostapd/driver_wired.c
endif

ifdef CONFIG_DRIVER_MADWIFI
L_CFLAGS += -DCONFIG_DRIVER_MADWIFI
OBJS += hostapd/driver_madwifi.c
CONFIG_L2_PACKET=y
endif

ifdef CONFIG_DRIVER_ATHEROS
L_CFLAGS += -DCONFIG_DRIVER_ATHEROS
OBJS += hostapd/driver_atheros.c
CONFIG_L2_PACKET=y
endif

ifdef CONFIG_DRIVER_PRISM54
L_CFLAGS += -DCONFIG_DRIVER_PRISM54
OBJS += hostapd/driver_prism54.c
endif

ifdef CONFIG_DRIVER_NL80211
L_CFLAGS += -DCONFIG_DRIVER_NL80211
OBJS += hostapd/driver_nl80211.c hostapd/radiotap.c
LIBS += -llibnl
ifdef CONFIG_LIBNL20
LIBS += -llibnl-genl
L_CFLAGS += -DCONFIG_LIBNL20
endif
endif

ifdef CONFIG_DRIVER_BSD
L_CFLAGS += -DCONFIG_DRIVER_BSD
OBJS += hostapd/driver_bsd.c
CONFIG_L2_PACKET=y
CONFIG_DNET_PCAP=y
CONFIG_L2_FREEBSD=y
endif

ifdef CONFIG_DRIVER_TEST
L_CFLAGS += -DCONFIG_DRIVER_TEST
OBJS += hostapd/driver_test.c
endif

ifdef CONFIG_DRIVER_NONE
L_CFLAGS += -DCONFIG_DRIVER_NONE
OBJS += hostapd/driver_none.c
endif
```

```
ifdef CONFIG_L2_PACKET
ifdef CONFIG_DNET_PCAP
ifdef CONFIG_L2_FREEBSD
LIBS += -lpcap
OBJS += src/l2_packet/l2_packet_freebsd.c
else
LIBS += -ldnet -lpcap
OBJS += src/l2_packet/l2_packet_pcap.c
endif
else
OBJS += src/l2_packet/l2_packet_linux.c
endif
else
OBJS += src/l2_packet/l2_packet_none.c
endif


ifdef CONFIG_EAP_MD5
L_CFLAGS += -DEAP_MD5
OBJS += src/eap_server/eap_md5.c
CHAP=y
endif

ifdef CONFIG_EAP_TLS
L_CFLAGS += -DEAP_TLS
OBJS += src/eap_server/eap_tls.c
TLS_FUNCS=y
endif

ifdef CONFIG_EAP_PEAP
L_CFLAGS += -DEAP_PEAP
OBJS += src/eap_server/eap_peap.c
OBJS += src/eap_common/eap_peap_common.c
TLS_FUNCS=y
CONFIG_EAP_MSCHAPV2=y
endif

ifdef CONFIG_EAP_TTLS
L_CFLAGS += -DEAP_TTLS
OBJS += src/eap_server/eap_ttls.c
TLS_FUNCS=y
CHAP=y
endif

ifdef CONFIG_EAP_MSCHAPV2
L_CFLAGS += -DEAP_MSCHAPv2
OBJS += src/eap_server/eap_mschapv2.c
MS_FUNCS=y
endif

ifdef CONFIG_EAP_GTC
L_CFLAGS += -DEAP_GTC
OBJS += src/eap_server/eap_gtc.c
endif

ifdef CONFIG_EAP_SIM
L_CFLAGS += -DEAP_SIM
OBJS += src/eap_server/eap_sim.c
CONFIG_EAP_SIM_COMMON=y
endif

ifdef CONFIG_EAP_AKA
L_CFLAGS += -DEAP_AKA
OBJS += src/eap_server/eap_aka.c
CONFIG_EAP_SIM_COMMON=y
endif

ifdef CONFIG_EAP_AKA_PRIME
L_CFLAGS += -DEAP_AKA_PRIME
endif
```

```
ifdef CONFIG_EAP_SIM_COMMON
OBJS += src/eap_common/eap_sim_common.c
# Example EAP-SIM/AKA interface for GSM/UMTS authentication. This can be
# replaced with another file implementating the interface specified in
# eap_sim_db.h.
OBJS += src/eap_server/eap_sim_db.c
NEED_FIPS186_2_PRF=y
endif

ifdef CONFIG_EAP_PAX
L_CFLAGS += -DEAP_PAX
OBJS += src/eap_server/eap_pax.c src/eap_common/eap_pax_common.c
endif

ifdef CONFIG_EAP_PSK
L_CFLAGS += -DEAP_PSK
OBJS += src/eap_server/eap_psk.c src/eap_common/eap_psk_common.c
endif

ifdef CONFIG_EAP_SAKE
L_CFLAGS += -DEAP_SAKE
OBJS += src/eap_server/eap_sake.c src/eap_common/eap_sake_common.c
endif

ifdef CONFIG_EAP_GPSK
L_CFLAGS += -DEAP_GPSK
OBJS += src/eap_server/eap_gpsk.c src/eap_common/eap_gpsk_common.c
ifdef CONFIG_EAP_GPSK_SHA256
L_CFLAGS += -DEAP_GPSK_SHA256
endif
NEED_SHA256=y
endif

ifdef CONFIG_EAP_VENDOR_TEST
L_CFLAGS += -DEAP_VENDOR_TEST
OBJS += src/eap_server/eap_vendor_test.c
endif

ifdef CONFIG_EAP_FAST
L_CFLAGS += -DEAP_FAST
OBJS += src/eap_server/eap_fast.c
OBJS += src/eap_common/eap_fast_common.c
TLS_FUNCS=y
NEED_T_PRF=y
endif

ifdef CONFIG_WPS
L_CFLAGS += -DCONFIG_WPS -DEAP_WSC
OBJS += src/utils/uuid.c
OBJS += wps_hostapd.c
OBJS += src/eap_server/eap_wsc.c src/eap_common/eap_wsc_common.c
OBJS += src/wps/wps.c
OBJS += src/wps/wps_common.c
OBJS += src/wps/wps_attr_parse.c
OBJS += src/wps/wps_attr_build.c
OBJS += src/wps/wps_attr_process.c
OBJS += src/wps/wps_dev_attr.c
OBJS += src/wps/wps_enrollee.c
OBJS += src/wps/wps_registrar.c
NEED_DH_GROUPS=y
NEED_SHA256=y
NEED_CRYPTO=y
NEED_BASE64=y

ifdef CONFIG_WPS_UPNP
L_CFLAGS += -DCONFIG_WPS_UPNP
OBJS += src/wps/wps_upnp.c
OBJS += src/wps/wps_upnp_ssdp.c
OBJS += src/wps/wps_upnp_web.c
OBJS += src/wps/wps_upnp_event.c
```

```
OBJS += src/wps/httpread.c
endif

endif

ifdef CONFIG_EAP_IKEV2
L_CFLAGS += -DEAP_IKEV2
OBJS += src/eap_server/eap_ikev2.c src/eap_server/ikev2.c
OBJS += src/eap_common/eap_ikev2_common.c src/eap_common/ikev2_common.c
NEED_DH_GROUPS=y
endif

ifdef CONFIG_EAP_TNC
L_CFLAGS += -DEAP_TNC
OBJS += src/eap_server/eap_tnc.c
OBJS += src/eap_server/tncs.c
NEED_BASE64=y
ifndef CONFIG_DRIVER_BSD
LIBS += -ldl
endif
endif

# Basic EAP functionality is needed for EAPOL
OBJS += src/eap_server/eap.c
OBJS += src/eap_common/eap_common.c
OBJS += src/eap_server/eap_methods.c
OBJS += src/eap_server/eap_identity.c

ifdef CONFIG_EAP
L_CFLAGS += -DEAP_SERVER
endif

ifndef CONFIG_TLS
CONFIG_TLS=openssl
endif

ifeq ($(CONFIG_TLS), internal)
ifndef CONFIG_CRYPTO
CONFIG_CRYPTO=internal
endif
endif
ifeq ($(CONFIG_CRYPTO), libtomcrypt)
L_CFLAGS += -DCONFIG_INTERNAL_X509
endif
ifeq ($(CONFIG_CRYPTO), internal)
L_CFLAGS += -DCONFIG_INTERNAL_X509
endif


ifdef TLS_FUNCS
# Shared TLS functions (needed for EAP_TLS, EAP_PEAP, and EAP_TTLS)
L_CFLAGS += -DEAP_TLS_FUNCS
OBJS += src/eap_server/eap_tls_common.c
NEED_TLS_PRF=y
ifeq ($(CONFIG_TLS), openssl)
OBJS += src/crypto/tls_openssl.c
LIBS += -lssl -lcrypto
LIBS_p += -lcrypto
LIBS_h += -lcrypto
endif
ifeq ($(CONFIG_TLS), gnutls)
OBJS += src/crypto/tls_gnutls.c
LIBS += -lgnutls -lgcrypt -lgpg-error
LIBS_p += -lgcrypt
LIBS_h += -lgcrypt
endif
ifdef CONFIG_GNUTLS_EXTRA
L_CFLAGS += -DCONFIG_GNUTLS_EXTRA
LIBS += -lgnutls-extra
endif
ifeq ($(CONFIG_TLS), internal)
```

```
OBJS += src/crypto/tls_internal.c
OBJS += src/tls/tlsv1_common.c src/tls/tlsv1_record.c
OBJS += src/tls/tlsv1_cred.c src/tls/tlsv1_server.c
OBJS += src/tls/tlsv1_server_write.c src/tls/tlsv1_server_read.c
OBJS += src/tls/asn1.c  src/tls/x509v3.c
OBJS_p += src/tls/asn1.c
OBJS_p += src/crypto/rc4.c src/crypto/aes_wrap.c src/crypto/aes.c
NEED_BASE64=y
L_CFLAGS += -DCONFIG_TLS_INTERNAL
L_CFLAGS += -DCONFIG_TLS_INTERNAL_SERVER
ifeq ($(CONFIG_CRYPTO), internal)
ifdef CONFIG_INTERNAL_LIBTOMMATH
L_CFLAGS += -DCONFIG_INTERNAL_LIBTOMMATH
else
LIBS += -ltommath
LIBS_p += -ltommath
endif
endif
ifeq ($(CONFIG_CRYPTO), libtomcrypt)
LIBS += -ltomcrypt -ltfm
LIBS_p += -ltomcrypt -ltfm
endif
endif
NEED_CRYPTO=y
else
OBJS += src/crypto/tls_none.c
endif

ifdef CONFIG_PKCS12
L_CFLAGS += -DPKCS12_FUNCS
endif

ifdef MS_FUNCS
OBJS += src/crypto/ms_funcs.c
NEED_CRYPTO=y
endif

ifdef CHAP
OBJS += src/eap_common/chap.c
endif

ifdef NEED_CRYPTO
ifndef TLS_FUNCS
ifeq ($(CONFIG_TLS), openssl)
LIBS += -lcrypto
LIBS_p += -lcrypto
LIBS_h += -lcrypto
endif
ifeq ($(CONFIG_TLS), gnutls)
LIBS += -lgcrypt
LIBS_p += -lgcrypt
LIBS_h += -lgcrypt
endif
ifeq ($(CONFIG_TLS), internal)
ifeq ($(CONFIG_CRYPTO), libtomcrypt)
LIBS += -ltomcrypt -ltfm
LIBS_p += -ltomcrypt -ltfm
endif
endif
endif
ifeq ($(CONFIG_TLS), openssl)
OBJS += src/crypto/crypto_openssl.c
OBJS_p += src/crypto/crypto_openssl.c
HOBJS += src/crypto/crypto_openssl.c
CONFIG_INTERNAL_SHA256=y
endif
ifeq ($(CONFIG_TLS), gnutls)
OBJS += src/crypto/crypto_gnutls.c
OBJS_p += src/crypto/crypto_gnutls.c
HOBJS += src/crypto/crypto_gnutls.c
CONFIG_INTERNAL_SHA256=y
```

```
endif
ifeq ($(CONFIG_TLS), internal)
ifeq ($(CONFIG_CRYPTO), libtomcrypt)
OBJS += src/crypto/crypto_libtomcrypt.c
OBJS_p += src/crypto/crypto_libtomcrypt.c
CONFIG_INTERNAL_SHA256=y
endif
ifeq ($(CONFIG_CRYPTO), internal)
OBJS += src/crypto/crypto_internal.c src/tls/rsa.c src/tls/bignum.c
OBJS_p += src/crypto/crypto_internal.c src/tls/rsa.c src/tls/bignum.c
L_CFLAGS += -DCONFIG_CRYPTO_INTERNAL
CONFIG_INTERNAL_AES=y
CONFIG_INTERNAL_DES=y
CONFIG_INTERNAL_SHA1=y
CONFIG_INTERNAL_MD4=y
CONFIG_INTERNAL_MD5=y
CONFIG_INTERNAL_SHA256=y
endif
endif
else
CONFIG_INTERNAL_AES=y
CONFIG_INTERNAL_SHA1=y
CONFIG_INTERNAL_MD5=y
CONFIG_INTERNAL_SHA256=y
endif

ifdef CONFIG_INTERNAL_AES
L_CFLAGS += -DINTERNAL_AES
endif
ifdef CONFIG_INTERNAL_SHA1
L_CFLAGS += -DINTERNAL_SHA1
endif
ifdef CONFIG_INTERNAL_SHA256
L_CFLAGS += -DINTERNAL_SHA256
endif
ifdef CONFIG_INTERNAL_MD5
L_CFLAGS += -DINTERNAL_MD5
endif
ifdef CONFIG_INTERNAL_MD4
L_CFLAGS += -DINTERNAL_MD4
endif
ifdef CONFIG_INTERNAL_DES
L_CFLAGS += -DINTERNAL_DES
endif

ifdef NEED_SHA256
OBJS += src/crypto/sha256.c
endif

ifdef NEED_DH_GROUPS
OBJS += src/crypto/dh_groups.c
endif

ifndef NEED_FIPS186_2_PRF
L_CFLAGS += -DCONFIG_NO_FIPS186_2_PRF
endif

ifndef NEED_T_PRF
L_CFLAGS += -DCONFIG_NO_T_PRF
endif

ifndef NEED_TLS_PRF
L_CFLAGS += -DCONFIG_NO_TLS_PRF
endif

ifdef CONFIG_RADIUS_SERVER
L_CFLAGS += -DRADIUS_SERVER
OBJS += src/radius/radius_server.c
endif

ifdef CONFIG_IPV6
```

```
L_CFLAGS += -DCONFIG_IPV6
endif

ifdef CONFIG_DRIVER_RADIUS_ACL
L_CFLAGS += -DCONFIG_DRIVER_RADIUS_ACL
endif

ifdef CONFIG_FULL_DYNAMIC_VLAN
# define CONFIG_FULL_DYNAMIC_VLAN to have hostapd manipulate bridges
# and vlan interfaces for the vlan feature.
L_CFLAGS += -DCONFIG_FULL_DYNAMIC_VLAN
endif

ifdef NEED_BASE64
OBJS += src/utils/base64.c
endif

ifdef CONFIG_NO_STDOUT_DEBUG
L_CFLAGS += -DCONFIG_NO_STDOUT_DEBUG
endif

ifdef CONFIG_NO_AES_EXTRAS
L_CFLAGS += -DCONFIG_NO_AES_UNWRAP
L_CFLAGS += -DCONFIG_NO_AES_CTR -DCONFIG_NO_AES_OMAC1
L_CFLAGS += -DCONFIG_NO_AES_EAX -DCONFIG_NO_AES_CBC
L_CFLAGS += -DCONFIG_NO_AES_DECRYPT
L_CFLAGS += -DCONFIG_NO_AES_ENCRYPT_BLOCK
endif


ifeq ($(HOSTAPD_BUILD),true)

include $(CLEAR_VARS)
LOCAL_MODULE := hostapd.conf
LOCAL_MODULE_TAGS := user eng development
LOCAL_SRC_FILES := hostapd/hostapd.conf
LOCAL_MODULE_CLASS := ETC
LOCAL_MODULE_PATH := $(TARGET_OUT)/etc/hostapd
include $(BUILD_PREBUILT)

include $(CLEAR_VARS)
LOCAL_MODULE := hostapd
LOCAL_SHARED_LIBRARIES := libc libcutils libcrypto libssl libnl
LOCAL_CFLAGS := $(L_CFLAGS)
LOCAL_SRC_FILES := $(OBJS)
LOCAL_C_INCLUDES := $(INCLUDES)
LOCAL_MODULE_TAGS := user eng development
LOCAL_MODULE_PATH := /root/Documents/eclair-x86/out/target/product/eeepc/system/bin
include $(BUILD_EXECUTABLE)

endif
```