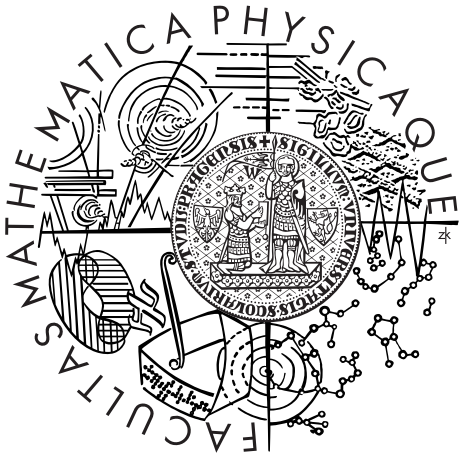
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jan Kolárik

IEEE 802.11 wireless networking for HelenOS

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Martin Děcký
Study programme: Software Systems
Specialization: Computer Science

Prague 2015

I would like to thank my supervisor, Martin Děcký, for his guidance and valuable advices during working on this thesis.

I would also like to thank my friend, David Brázdil, for the grammatical corrections of the text.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague on 30.07.2015 Jan Kolárik

Název práce: IEEE 802.11 wireless networking for HelenOS

Autor: Jan Kolárik

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: Mgr. Martin Děcký, Katedra distribuovaných a spolehlivých systémů

Abstrakt: IEEE 802.11 (alias WiFi) je populární standard pro bezdrátové LAN sítě. Tato práce dokumentuje jeho implementaci v rámci výzkumného operačního systému HelenOS, který se od tradičních systémů liší tím, že je postavený na mikrojádrové architektuře. Nejprve jsou popsány obecné koncepty WiFi technologie. Následuje popis použitých nástrojů při vývoji práce. Společně s ním je znázorněna zvolená metoda integrace bezdrátového frameworku do současného systému. V další kapitole jsou rozebrány implementační detaily práce. Zde je důkladně vysvětlena struktura ovladače pro vybrané WiFi zařízení a popsána funkcionalita IEEE 802.11 knihovny. V závěru práce je pak zhodnocení výsledných vlastností a porovnání dodané implementace s již existujícími řešeními.

Klíčová slova: HelenOS, IEEE 802.11, WiFi, síťové architektury, ovladač

Title: IEEE 802.11 wireless networking for HelenOS

Author: Jan Kolárik

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Martin Děcký, Department of Distributed and Dependable Systems

Abstract: IEEE 802.11 (a.k.a. WiFi) is a popular wireless LAN specification. This thesis documents implementation of this standard within the experimental operating system HelenOS which differs from conventional operating systems with its microkernel-based design. First, the basic concepts of WiFi technology are described. Description of external tools used during development is following. Together with that, the chosen method of integrating the wireless framework into existing system is depicted. In the next chapter we analyse the implementation details of the work. There is thoroughly explained the structure of the driver for selected WiFi device and also the functionality of resulting IEEE 802.11 library. In the end of the thesis there is an evaluation of the features of final work and comparison of enclosed implementation with existing solutions.

Keywords: HelenOS, IEEE 802.11, WiFi, networking, driver

Contents

Introduction

WiFi networks have become an inseparable part of a common computer user's everyday life and a wireless Internet connection is nowadays available literally on every street corner. WiFi provides a convenient way for connecting a user to a computer network without the need for dealing with stringing cords plugged into a wall.

In the beginning, the only possibility for securing a WiFi connection was to use the WEP method introduced as a part of the IEEE 802.11 standard. However, it was soon discovered to have major security flaws and using wireless networks became unsuitable for transferring sensitive data. A few years later, this problem was solved by releasing a security amendment which included protocols providing much better safety.

WiFi technology is used today either for interconnection of a user device with an ISP's end instrument or for bridging between a service provider and an end user location. The upside for the user is mobility, but comes at the cost of worse connection stability and lower data transfer rates compared to wired connections.

Motivation and goals

This thesis has two main goals. The first is to design and implement an IEEE 802.11 framework providing generic wireless functionality to device drivers and user applications. The second is to implement a wireless device driver for a specific IEEE 802.11-compliant device which will also serves as an exemplary implementation.

Before writing this thesis, HelenOS had only support for wired network devices based on Ethernet technology. Our task is also to integrate the resulting framework prototype implementation and driver into the existing HelenOS environment and providing the same functionality as wired devices. Given the microkernel nature of HelenOS, the process is rather different from any conventional operating system.

To provide a secure network access for users, we also have to implement a library with cryptographic functions used for data encryption and decryption. The library will contain many generic functions that are not IEEE 802.11-specific and can be reused for other purposes.

The implementation should also be accompanied by a simple front-end tool for scanning networks and managing wireless connections.

Contents

The thesis is divided into several sections. Chapter *Context* describes basic concepts of the IEEE 802.11 technology and also includes an introduction into the HelenOS system. Chapter *Analysis* comprises an evaluation of potential suitable devices for driver development, an analysis of the selected device's

existing driver in Linux, a discussion about implementation decisions, description of development tools used during implementation and integration of the device driver and wireless framework into HelenOS. Chapter *Implementation* documents implementation details during source code development and enumerates a list of changes to existing HelenOS source code. Chapter *Evaluation* resumes the implemented functionality and compares performance of the wireless framework to the wired one. Chapter *Related work* covers existing implementations of IEEE 802.11 networking stack in other operating systems. Finally chapter *Conclusion* sums up the results of the efforts.

1. Context

This section begins with a short introduction into IEEE 802.11 fundamentals, followed by a brief familiarization with the HelenOS operating system.

1.1 IEEE 802.11 concepts

The family of IEEE 802.11 standards (also referred to as WiFi) is a set of techniques for implementing wireless local area network communication between computers. Like other IEEE 802 standards, they focus on the two lowest levels of the ISO/OSI model - physical and link layer (the media access control part).

1.1.1 Different variants of standard

With the innovations in transmission infrastructure and modulation techniques, new amendments to original IEEE 802.11 standard (dating back to 1997) were released. The first two, IEEE 802.11a and IEEE 802.11b, were ratified in 1999.

The widely accepted 802.11b operates in 2.4GHz band with maximum data rate of 11Mbit/s. Its adjacent channels overlap, which means that it needs to leave several channels clear between the used channels to avoid interference. It is capable of adaptive rate selection based on signal quality in order to decrease the number of retransmitted packets.

IEEE 802.11a variant operates on 5GHz frequencies at maximum speed of 54Mbit/s, originally with non-overlapping channels. The idea of migrating to different spectrum of frequencies is based on the observation that 2.4GHz band is overcrowded. Hence, moving to the 5Ghz band gives the advantage of less interference, but at the same time introduces transmission problems which lower the range compared to the IEEE 802.11b variant. The 802.11a is therefore not compatible with 802.11b/g networks.

In 2003, a third variant was announced under the 802.11g amendment. It uses the 2.4GHz band, same as 802.11b, but allows for maximum data rate of 54Mbit/s at the cost of instability in the presence of interference. The standard is backward compatible with 802.11b.

The IEEE 802.11n standard was ratified in October 2009. It uses the so-called multiple-input multiple-output (MIMO) technique to increase data rates up to 150Mbit/s, and a frame aggregation feature for packing multiple data units together for reducing overheads associated with decoding MAC frame fields..

Apart from these basic variants of IEEE 802.11, several other variants exist as extensions or corrections to previous specifications.

Implementation in this thesis supports communication with devices which implement the 802.11b and 802.11g variants.

1.1.2 Operating modes

There are two basic modes of operation defined in IEEE 802.11: infrastructure mode and ad-hoc mode.

In the infrastructure mode, nodes are organized in star topology, i.e. one node operates as the *access point* and other nodes, called *stations*, communicate solely through it. Conversely, an ad-hoc network consists of stations which communicate with one another directly.

The smallest subset of a wireless network architecture is referred to as the basic service set (BSS). It consists of two or more nodes that communicate directly. A distribution system (DS) is used for BSS interconnection on link layer. The resulting network is called the extended services set (ESS) and it corresponds to wireless network (WLAN). Access points are parts of the DS and they are assumed static (immobile) nodes. All access points in the same ESS have identical identifier (SSID).

Station transitions between networks are possible. Migration between two BSSs inside one ESS is done through a DS service called *reassociation*. Roaming (transition between two ESSs) is not supported and it should be performed by the station itself..

1.1.3 Access methods

Because in practice there are typically more candidates to transmit over a transmission medium, the communication protocol must coordinate simultaneous attempts to access to the shared medium. The IEEE 802.11 standard defines three basic ways of dealing with this issue.

The radio interface of communicating nodes is usually half-duplex, which prevents the sender from recognizing communication interference. The solution is to send packet acknowledgements to notify sender about successful packet reception.

First method is Distributed coordination function employing Carrier sense multiple access with collision avoidance (DCF CSMA/CA). Candidates monitor the shared medium for ongoing communication and wait for random time if somebody else is currently transmitting before trying again. Variable waiting time is used to eliminate collisions that occur when multiple stations sense the channel as busy and then simultaneously find that the channel has been released.

DCF suffers from two problems called hidden and exposed terminal. The hidden terminal problem occurs when a node is visible from an access point, but not from other communicating nodes which then do not detect an ongoing transmission. Exposed terminal is a similar problem when the medium is unused but a candidate detects it as occupied because of signal interference from a neighbouring node. These can be eliminated by using DCF together with the Request-to-send and Clear-to-send (RTS+CTS) mechanism. A sender first sends an RTS frame to inform the recipient's neighbours about an upcoming transmission. The recipient then replies with a CTS frame to the sender's neighbours.

Besides the mentioned distributed methods, the standard also specifies the Point coordination function (PCF) centralized method. It requires existence

of an access point which coordinates traffic in the network (not applicable in ad-hoc mode).

1.2 IEEE 802.11 authentication

This section delves into the details of the process of how a station, a wireless client device, connects to an access point.

1.2.1 Finding the network

The first thing a station needs to do before using a wireless network is to find it. There are two ways how it can accomplish that.

First method is called passive scanning. The station iterates over the supported channels and waits for so-called *beacon* frames which contain information about the BSS that sent them.

Second approach is called active scanning. Rather than passively listening for network announcements, the station construct a special management frame called *probe request* and broadcasts it on every channel it wants to examine. BSSs reply with a *probe response* frame, similar to the beacon frame.

Both methods are used when searching for networks in this thesis implementation.

1.2.2 Basic authentication

After selecting the network, the station needs to authenticate against the corresponding access point. Even though this is referred to as *authentication*, this process does not provide significant security guarantees to the network. It can be seen as an initial step in the handshake process that identifies the station to the network.

IEEE 802.11 defines two types of authentication methods: open system authentication and shared key authentication.

In open system authentication, the station sends an authentication request to the selected access point, containing the station's MAC address, and waits for an authentication response verification frame.

With the shared key authentication method, the access point verifies the station's knowledge of a shared secret (network password). Client initiating authentication sends an authentication request to the access point. AP responds with an authentication response including a challenge text. The client is expected to encrypt the challenge text using the shared secret and send it back to the AP. If the result matches the AP's local cipher text, the AP sends back a positive confirmation.

The shared key authentication is the basis of the WEP security algorithm considered deprecated since 2004. For that reason, it is not supported in this thesis' implementation..

1.2.3 Association

To start using a network, a station needs to register itself with an access point. This procedure allows a distributed system to keep track of the station's location so that frames are forwarded to the correct access point. When the association request is granted, the client receives a unique Association identifier (AID). This can be seen as mapping between the client station and an AP's port. It replaces the physical link in a wired LAN.

The whole sequence of basic authentication process can be seen below in Figure 1.1.

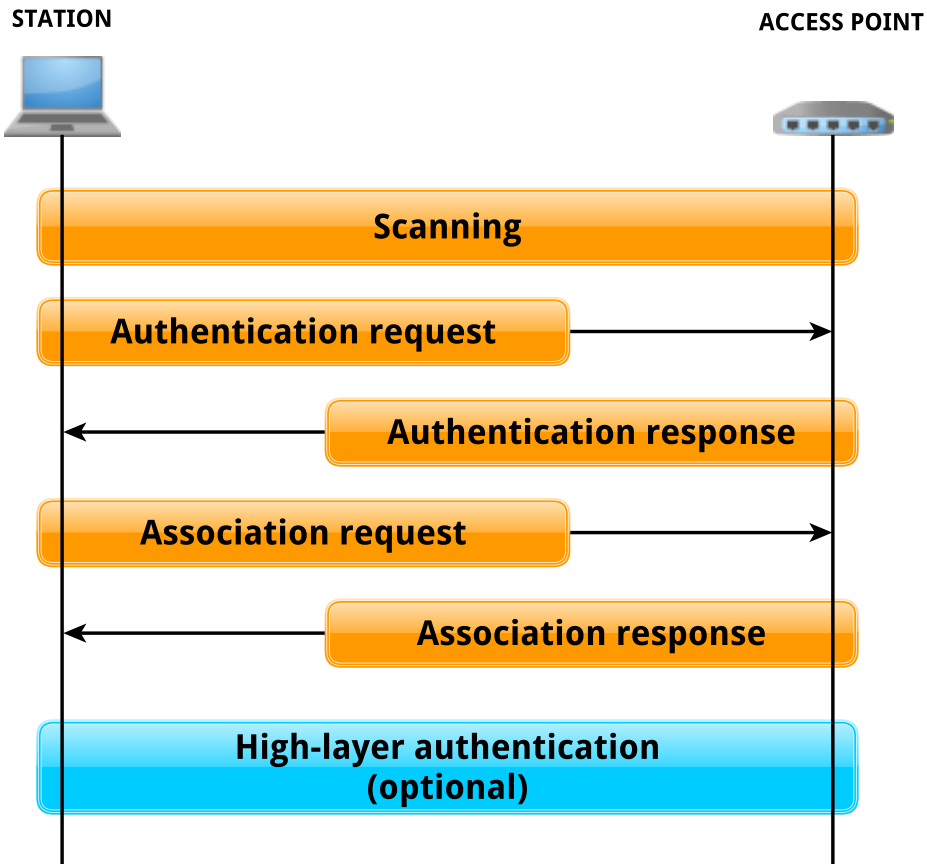


Figure 1.1: Station connection process sequence

1.2.4 High-layer authentication

About 15 years ago, the only link-level security option was using the WEP algorithm introduced as part of the original IEEE 802.11 standard issued in 1997. A few years later it was proclaimed insecure as several successful attacks against it emerged. In order to provide better security, IEEE designed new security protocols issued under IEEE 802.11i amendment in 2004 [1]. The new protocols support strong cryptography key exchange and mutual authentication possibility.

There are two different methods how high-layer authentication can be accomplished which differ in the way that authentication keys are distributed.

One is based on pre-shared keys and without the need for an authentication server. It is called WPA-Personal (also known as WPA-PSK) and was designed for smaller home or office networks. Second, known as WPA-Enterprise (also referred to as WPA-802.1X), requires a RADIUS server for client authentication and provides security suitable for business environments.

Only support for the WPA-Personal mode was implemented with this thesis and hence full description of the WPA-Enterprise mode is omitted.

4-way handshake

In order to establish an encrypted connection, the client needs to undergo a key management procedure to obtain cryptography keys to be used in subsequent data transmission. This process is called the four-way handshake.

At the beginning, both parties possess the secret previously shared during the authentication process. This secret is mapped with a cryptographic function to a hash value known as the pre-shared key (PSK). In the case of WPA-Personal, this value is directly used as the pairwise master key (PMK). The PMK was designed to last the entire session and so it should not be exposed anywhere. To that end, it is hard coded into the access point.

PMK is not used directly for data communication encryption but rather used to derive a pair of temporal transmission keys. These are the pairwise transient key (PTK) for protecting communication between station and the access point and group temporal key (GTK) used for broadcast data reception.

Handshake is separated into four phases which are described as follows:

1. Authenticator sends the client a nonce (ANonce), which is a pseudorandom sequence of numbers.

2. Client creates his own nonce (SNonce) and uses it for derivation of PTK. Then it sends a message to the authenticator containing this nonce. This message is complemented by message integrity code (MIC) constructed from PTK, so that the authenticator can perform verification of the secret used by the client and also to derive the PTK.

3. Third message, sent by the authenticator, contains security parameters (same as in beacon and probe responses frames) and also the GTK encrypted using the PTK. Again, the message is secured by MIC.

4. Client installs decoded temporal keys and sends a confirmation to the authenticator.

After successful completion of the handshake, encrypted communication may begin. Algorithm used for encryption is based on selected security suite during authentication phase. These days, most devices support hardware encryption of data transmission. Support for software encryption of frames is not included in this thesis.

The procedure is illustrated further in Figure 1.2.

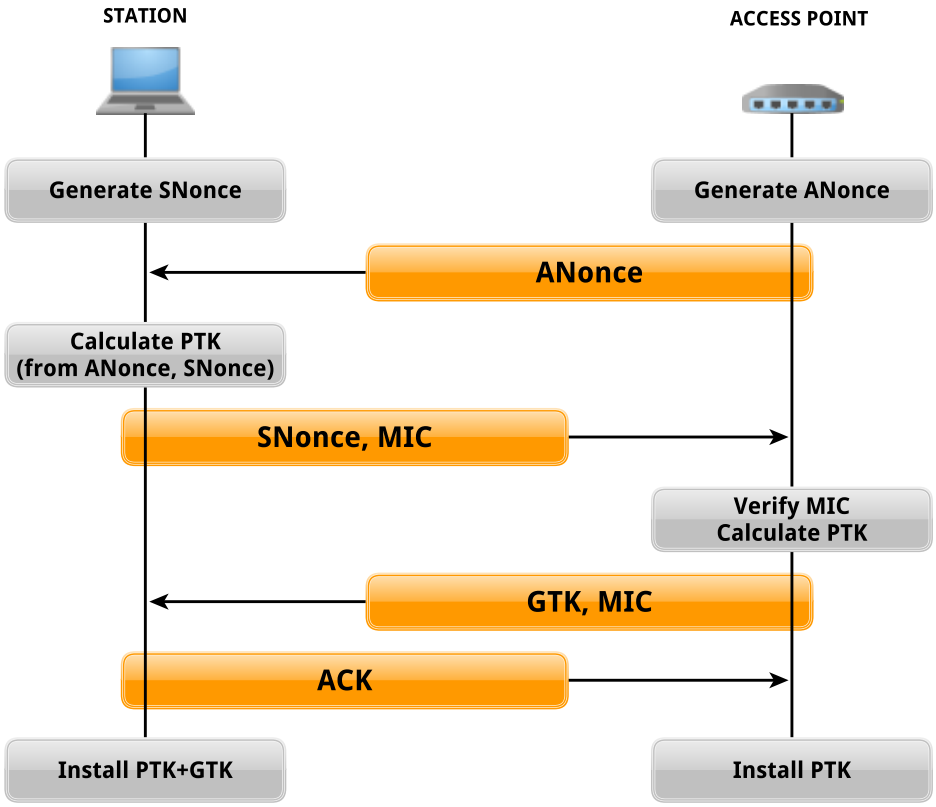


Figure 1.2: 4-way handshake sequence

1.3 Introduction to HelenOS

HelenOS is operating system with microkernel architecture. This is very different from conventional operating systems like Windows, Mac OS or Linux.

In most modern operating systems, we can differentiate between code that is running in user (unprivileged) mode and kernel (privileged) mode. Code running in mode kernel has complete and unrestricted access to the underlying hardware. Execution of a malicious code in kernel mode could therefore have catastrophic consequences on system operation. For that reason, the kernel mode is generally reserved only for the most trusted components of the operating system. In user mode, the executing code has no ability to access hardware or reference physical memory directly. Crashes of applications running in user mode are recoverable from and not dangerous to operation of the system. Applications can request a service from the kernel through system call API which checks authorization of the call and grants access to the execution of routine with higher privileges.

Microkernels require only the very necessary functions to run in privileged mode (scheduling, memory management and inter-process communication). All other system functionality and modules are implemented as programs in user space, usually referred to as servers which communicate with each other through IPC. This separation of operating system functions allows for better design modularity and system reliability.

IPC is a crucial component of HelenOS. System *tasks* communicate with each other by sending messages and waiting for replies. For each inter-process

connection, the IPC library creates a user-space thread called fibril, which is the smallest unit of execution. Multiple fibrils run simultaneously within the kernel which enables concurrent processing of messages.

HelenOS is a research operating system created at Faculty of Mathematics and Physics at the Charles University in Prague. The original SPARTAN kernel was developed between 2001-2004 by the founder of HelenOS Jakub Jermář. Since then, many people (mostly students) have contributed to the system and it is used as a platform for student software projects and theses.

Because of its microkernel design and open-source nature, HelenOS is ideal for demonstrating principles of operating systems and for experimentation without the need for delving deep inside the system implementation details.

2. Analysis

Now we are going to describe the preparation for the development process, external tools leveraged during the work and subsequently the method of integrating the wireless device driver and framework into the existing HelenOS environment.

2.1 Choosing a suitable WiFi device

It is currently not an easy to develop and deploy HelenOS on real hardware but it can be easily emulated under QEMU or VirtualBox. However, contemporary hypervisors do not support virtualized WiFi devices and we therefore need to choose a suitable WiFi device candidate which can be connected through the hypervisor to HelenOS.

When searching for an appropriate device, we have been deciding between two types. First option was to use an integrated WiFi adapter typically connected through MiniPCI or Mini PCIe bus. This method however needs support of Intel VT-d extension for I/O MMU virtualization on host computer. Another possibility is to use an external adapter connected via USB and use the USB pass-through feature of QEMU to delegate device management from host to guest OS.

Another criterion we want to meet is that the target device should have open technical specification or at least an available open-source driver in another operating system, so as to determine the protocol used for the communication between the device and the host.

Wide-spread availability of the chipset on the market is another point to consider.

After evaluating the specified requirements against the product offering on the market, we have found that the Atheros AR9271 chipset fulfils all the given criteria, with free product data sheet available, existing open firmware implementation and Linux driver released under the ISC license [2] compatible with the BSD license used by HelenOS.

2.2 Linux sources

During development of AR9271 device driver we had to deeply explore its Linux driver and whole Linux wireless network stack, because information contained in the product data sheet was insufficient for understanding the device communication protocol.

2.2.1 Debugging Linux code

In order to investigate data flow between the device and the host, we needed to debug the existing Linux implementation. We tried to use original debugging environment implemented for Atheros wireless drivers, but found both the source code and debugging information too complex and verbose, yet

lacking the crucial information about the device initialization protocol which we needed.

To omit the redundant parts of the source and to add more debugging information into the driver code, we had to compile the Linux driver ourselves. To this end, we used Fedora Linux which greatly facilitates this process.

Building the kernel for Fedora requires installing the development tools package from the main repository, followed by creating a directory tree where we will prepare and build the new RPM kernel package. Fedora, unlike many other distributions, does not allow building packages in privileged mode for security reasons, hence this is a directory owned by our normal user account.

Next, we download the kernel source code package matching our kernel version via a special Fedora downloader and install dependent packages needed to build it.

After that we have to set our kernel version as the target platform in the build tree configuration file.

Now we can modify the source code, add the desired debugging output and comment out unnecessary parts of the AR9271 driver code.

Once done with the modifications, we can build our module and insert it into kernel, remembering to remove the original Atheros driver module first.

The commands involved in this process are documented in the appendix at the end of the thesis.

Building external kernel modules is documented in more described in official Linux kernel documentation [3].

2.2.2 Structure of driver

The AR9271 Linux driver is a part of the Atheros Linux wireless driver family. The whole family can be divided into several groups based on technologies (802.11 standard variants) supported in each one, AR9271 being part of the `ath9k` group. In addition, there are different drivers based on the the bus used for communication (either PCI or USB interface), making `ath9k_htc` the driver which supports AR9271.

All drivers share some common parts which are located in the `ath` shared module.

Unlike other drivers in the `ath9k` family, `ath9k_htc` makes use of a special HTC + WMI + HIF framework for host-target communication.

Host/Target Communication layer (HTC) is responsible for communication between the host CPU and the target (firmware located on the USB adapter). It defines a basic API for build communication protocols for a specific target and the interconnecting bus. Multiple endpoints are used both in the host and the target for processing different types of services, e.g. processing control messages, beacon frames, management frames or data frames on different priority levels used to support QoS. HTC uses TX and RX completion callbacks that check the endpoint target and call registered callbacks for each endpoint. HTC does not understand the contents of messages it transports (only WMI does), but it does understand the mechanics of messaging with the Atheros chipset.

The Wireless Module Interface (WMI) is a wireless communication protocol for Atheros wireless devices. It defines a set of commands that can be issued to the target firmware or that the target firmware can send back to the host for processing. WMI supports use of events, i.e. special commands for which callbacks are registered.

Host Interface layer (HIF) is the interconnect driver interface used by a driver. It uses the HTC layer to register special callbacks for device insertion and removal. This layer also defines the callbacks to deal with USB messages coming from/to target device. During device initialization it also uploads the firmware to the device. HTC calls the HIF layer when it needs to access the chipset address space. An HIF implementation exists for each combination of platform and interconnect API (SDIO/MMC stack, USB stack, etc.).

The structure of the Atheros driver framework is pictured in Figure 2.1.

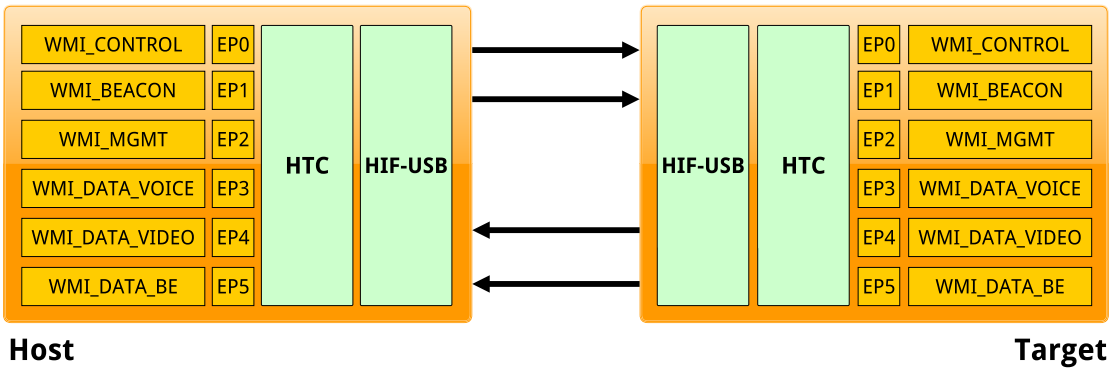


Figure 2.1: Ath9k_htc driver framework layout

2.3 External tools used for development

During the thesis implementation, several additional tools were used in order to simplify and speed up the development process. In this section, the two most important are described in more detail.

2.3.1 Wireshark

When working with data at the lowest level of abstraction as a programmer interpreting sequences of bytes, we have to construct data packets very carefully because the position and value of every single byte may play a role.

Typically when communicating with an embedded device on this level, there is no form of output in the target device which we could intercept and debug ongoing data transmission.

Wireshark is an ideal tool to solve this kind of problem. It is an open-source multi-platform packet analyser used for network transmission analysis, communications protocol development or other network testing purposes. It allows to examine data from ongoing network transmission while displaying them interactively, and also to capture the analyzed data on disk for later examination.

It can interpret a communication from a number of network types, such as Ethernet, IEEE 802.11 and more. It also has a multitude of sophisticated filtering and sorting options.

We used Wireshark to analyse data sent between the wireless adapter and its HelenOS driver running under a QEMU hypervisor. Because the adapter is connected to HelenOS via USB passthrough, there is no possibility to examine data at the link or network layers, because the host Linux OS considers the adapter a generic USB device. Wireshark helps us investigate the raw USB data stream.

Wireshark was also used during implementation of this thesis for the purposes of examining data transfer on the access point side (receiver of the station's packets). This is done by making a wireless hotspot from any computer station running conventional operating system and configuring Wireshark to monitor the target network interface. Wireshark can recognize network packets and interpret TCP communication which is useful for example for verifying successful authentication between the station and the access point or identifying request timeouts or packet loss.

It was sometimes also useful to capture Wireshark output from correctly working Linux implementation and compare it to the output from our HelenOS implementation to identify where the problem might lie.

Examples of the Wireshark application interface are demonstrated in Figures 2.2 and 2.3.

2.3.2 Python

When writing a complex piece of code or working with a substantial code base, it is important to have the ability to quickly prototype a new part of a larger design solution or to briskly test some of the system's functionality.

To this end, we choose Python which is a dynamic programming language, nowadays very popular in the community. Because of its simplicity, there is a huge amount of existing libraries implementing many features from numerical and cryptography functions to graphical transformations. It is therefore suitable for quick prototyping of applications or algorithms without the need for writing too many lines of code and detailed understanding of the internal mechanisms of third-party code.

In this project we used Python mainly for creating and testing cryptography functions prototypes and verifying the process of key derivation during the four-way handshake.

Example script used during development is available in the Appendix B at the end of this document.

2.4 HelenOS integration

In this section we describe how the wireless device driver and IEEE 802.11 stack are incorporated into the HelenOS system, and what other ways of integration exist.

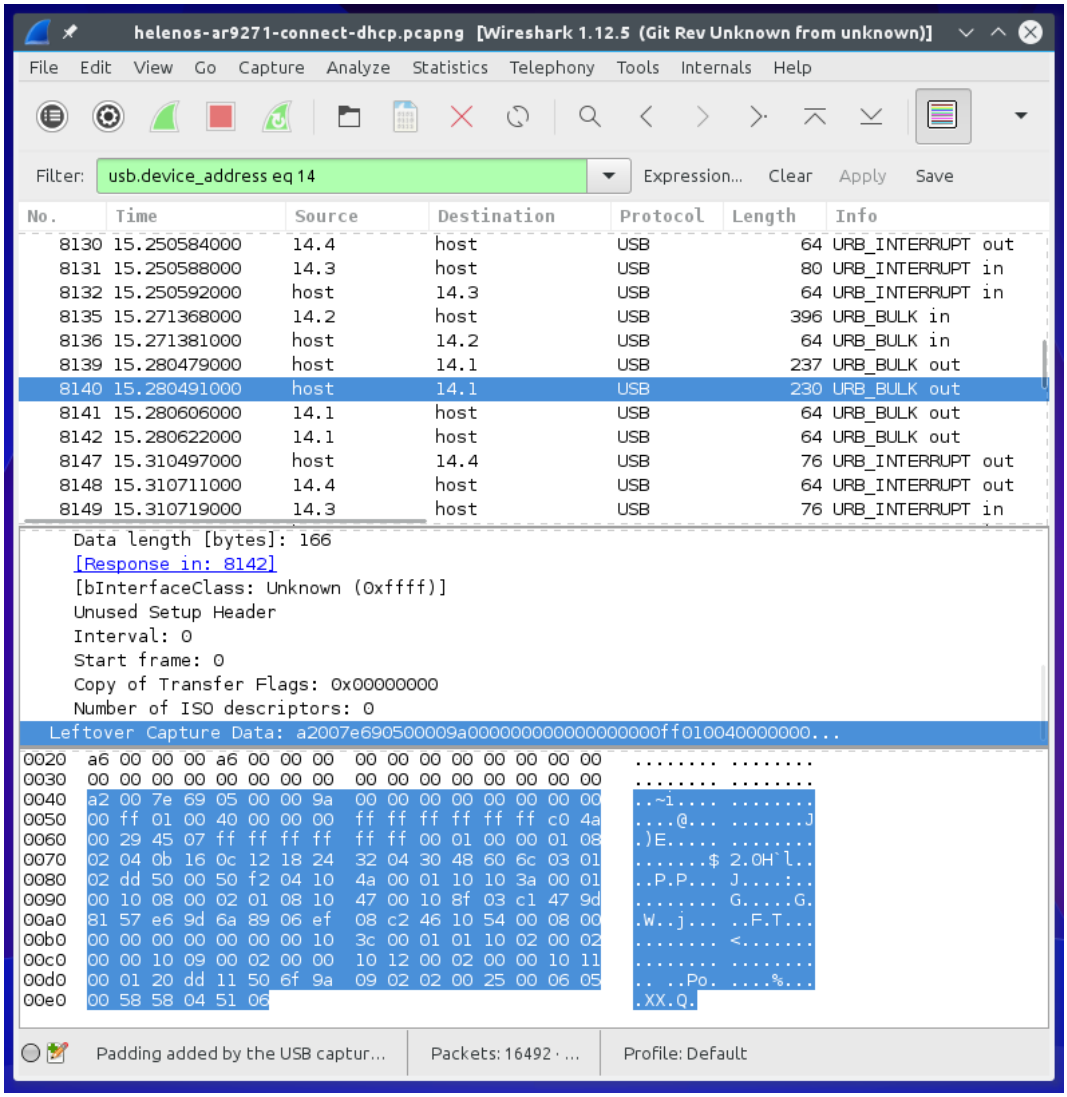


Figure 2.2: Wireshark UI showing raw USB communication with AR9271

2.4.1 Device driver

Device drivers for HelenOS are implemented as individual servers which live in user space and communicate through IPC. This structure is based on the HelenOS microkernel design and we are also using it to implement our wireless device driver.

HelenOS facilitates development of drivers with its device driver framework (DDF). The framework consists of the device manager which is the central point collecting information about available drivers in the system, and the `libdrv` library which is statically linked to every device driver and provides a uniform interface to communicate with other devices.

DDF framework structure

In DDF, the device driver model can be split into multiple parts. The first part is the device-specific protocol for communicating between a device and its driver, for example to read or write a character from/to device. The second

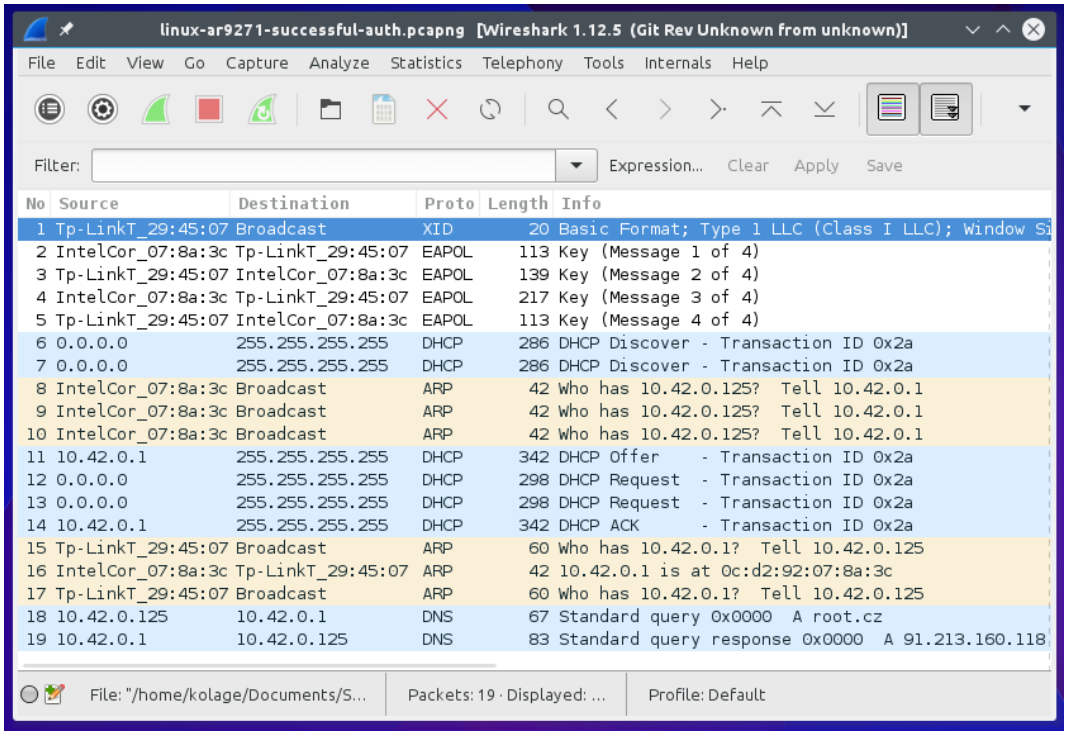


Figure 2.3: Analysis of 4-way handshake followed by DHCP and DNS queries

part are driver-specific operations. These provide generic functionality for this type of driver, such as initialization procedures and operations specific for a given type of device (e.g. for network devices operations like send or receive packet, etc.). The other parts of the driver are composed from logical functions (a.k.a. DDF functions). Each logical function can be either external or internal. External ones have an API exposed to other clients outside of DDF and internal ones are used for attaching child drivers.

Every driver is then attached to a specific device. There could also be more drivers suitable for that device in which case the DevMan decides which one will be used. Driver should implement the device- and driver-specific functionality and provide one or more DDF functions which will be exposed and registered under the selected category of devices, so that users can easily list and manage devices in each category. DDF functions also provide hooks to handle incoming requests. This is typically done inside the driver's entry point for probing new device.

Several frameworks built on top of the DDF architecture already exist in HelenOS and provide support for generic features useful for implementing device drivers in their respective domains.

Bypassing the framework

When implementing a device driver using just the generic DDF framework, the approach is quite straightforward.

We provide a class which implements a callback interface of generic device events and pass it to the DDF device manager. The interface method are then invoked by `libdrv` when corresponding events occur.

Usually we want to include some driver-specific device data inside generic driver structure. This can be done using a special setter function which stores a pointer to our driver-specific structure inside the generic one.

Then in the main function of our device driver we pass a pointer with the driver structure to generic DDF main function in order to initialize driver with DDF framework as shown in Listing 1.

When using a different driver framework built on top of DDF, the process is very similar. Only the resulting framework structure is passed as driver-specific data to the generic DDF driver.

A multitude of device drivers already exists in HelenOS implementing functions from the frameworks they build on. In this regard, our driver uses a slightly different approach, because we want to implement several interfaces.

When using the standard approach provided by one of the driver frameworks we have selected for our device driver, there is no interface for combining this framework with another one.

With regard to our AR9271 device, we want to use the network interface controller (NIC) interface combined with the USB framework. In the case of the USB framework, there already exists a mechanism which allows us to bypass the standard procedure of driver initialization [4] which is also used in the enclosed implementation of this thesis.

```
/** Device driver main function. */
int main(int argc, char *argv[])
{
  ...
  return usb_driver_main(&my_driver);
}

/** Callback when new device is attached. */
static int my_device_add(usb_device_t *dev)
{
  ...
  /* Setup driver specific data. */
  my_device_t *mdev = usb_device_data_alloc(dev, sizeof(my_device_t));

  /* Create DDF function. */
  ddf_fun_t *fun = ddf_fun_create(mdev->ddf_dev, fun_exposed, "port");

  /* Setup device operations. */
  ddf_fun_set_ops(fun, mdev_ops);

  /* Bind function to the system. */
  int rc = ddf_fun_bind(fun);
  ...
}
```

Listing 1: Standard USB framework initialization

```c
/** AR9271 driver main function. */
int main(int argc, char *argv[])
{
  ...
  return ddf_driver_main(&ar9271_driver);
}

/** Callback when new device is attached. */
static int ar9271_device_add(ddf_dev_t *dev)
{
  ...
  /* USB framework initialization. */
  usb_device_t *usb_device = calloc(1, sizeof(usb_device_t));
  int rc = usb_device_init(usb_device, dev, endpoints, &err_msg);

  /* Initialize IEEE 802.11 framework. */
  rc = ieee80211_init(ar9271->ieee80211_dev, &ar9271_ieee80211_ops,
    &ar9271_ieee80211_iface, &ar9271_ieee80211_nic_iface,
    &ar9271_ieee80211_dev_ops);
  ...
}

/** IEEE 802.11 framework initialization. */
int ieee80211_init(...)
{
  ...
  /* Setup implemented IEEE 802.11 and NIC operations. */
  int rc = ieee80211_implement(ieee80211_dev, ieee80211_ops,
    ieee80211_iface, ieee80211_nic_iface, ieee80211_nic_dev_ops);

  /* Setup NIC framework. */
  nic_t *nic = nic_create_and_bind(ddf_dev);
  nic_set_specific(nic, ieee80211_dev);
  nic_set_send_frame_handler(nic, ieee80211_send_frame);

  /* Setup DDF function. */
  ddf_fun_t *fun = ddf_fun_create(ieee80211_dev->ddf_dev,
    fun_exposed, "port0");

  nic_set_ddf_fun(nic, fun);
  ddf_fun_set_ops(fun, ieee80211_nic_dev_ops);
  ddf_fun_bind(fun);
  ddf_fun_add_to_category(fun, DEVICE_CATEGORY_NIC);
  ddf_fun_add_to_category(fun, DEVICE_CATEGORY_IEEE80211);
  ...
}
```

Listing 2: Bypassed USB framework initialization in AR9271 driver

As a result, the AR9271 driver uses the generic DDF framework for driver initialization and in device probe procedure, NIC interface is initialized as driver-specific data to generic DDF driver and the USB framework is initialized separately and stored as a pointer in the AR9271 driver-specific structure. This approach is documented in Listing 2 and the resulting connection of individual modules is illustrated in Figure 2.4.

The USB framework can't be used directly as AR9271 driver initialization framework, because NIC framework doesn't provide ability to bypass the standard framework initialization procedure, so implementing more interfaces would be much more complicated. In the presented solution, the relationship between individual frameworks results from the natural substance of the individual driver layers.

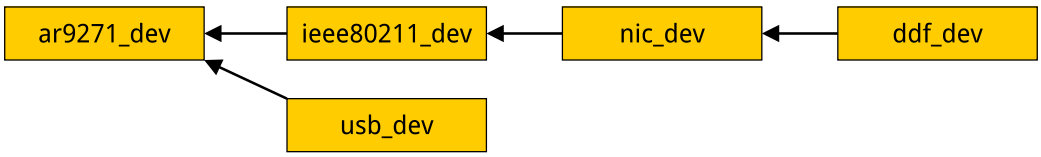


Figure 2.4: AR9271 driver hierarchy

2.4.2 Wireless framework

Design of the IEEE 802.11 framework can be divided into two parts described below.

Library with generic operations

Generic IEEE 802.11 functionality is packaged as a library that can be included by wireless device drivers.

It uses a similar approach as existing driver frameworks in HelenOS. A device driver which wants to implement IEEE 802.11 functions uses initialization functions of the framework to pass in data structures with implemented IEEE 802.11 and NIC operations. Some of the generic procedures, such as network scanning, are already implemented and can be used directly by most wireless devices. They do, however, built on elementary operations which must be implemented by each target device driver, e.g. switching to different operating frequency of device, otherwise it will not operate correctly.

Main function of the IEEE 802.11 framework is naturally the processing of network frames.

To handle incoming packets received by a wireless device, they must first be loaded from the device through communication bus into host computer and consequently stripped off to retrieve the IEEE 802.11 data frame itself. This is done by the wireless device driver. After processing of a received packet is completed, it is expected to pass its data further to the IEEE 802.11 layer, performed by calling the IEEE 802.11 RX handler. Framework then parses the frame and processes it directly if it is a IEEE 802.11 management frame. Otherwise it hands it over to the NIC framework which passes it

further to the Ethernet protocol layer (`ethip`). This layer also implements the ARP protocol to which it is logically connected. Based on the protocol type it either processes it directly (in case of ARP) or passes it to the server which implements the IP protocol (`inetsrv`). From there the packet is sent either to the TCP or UDP transport layer and then to a respective socket which is connected to a user application the data are addressed for. The scheme of the communication between individual layers is depicted in Figure 2.5.

Similarly, the packet is successively processed from the client to a wireless device. IEEE 802.11 implements the TX handler that is registered with the NIC framework for processing outgoing packets. It packs it with information from the IEEE 802.11 layer and passes it to the device driver TX callback which then sends it through the communication bus to the wireless device.

Another possibility to implement this part of IEEE 802.11 framework is to separate it to a server task and provide its functionality as a service. HelenOS network services such as `ethip` or `inetsrv` are implemented in this manner. We have chosen the solution using the library that can be included by device drivers, because IEEE 802.11 framework is connected only with wireless devices and also we do not want to add another level of indirection coming from the IPC task communication.

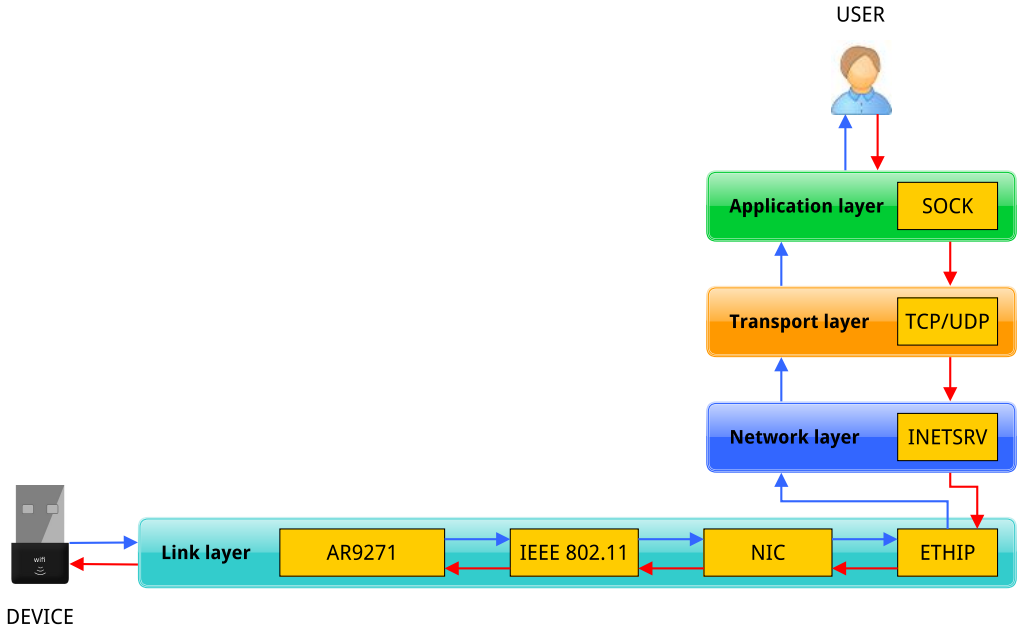


Figure 2.5: Layered communication scheme

Interface for client applications

There is also a second part of the IEEE 802.11 framework which exposes an interface for client applications. Currently, it consists of functions for connecting or disconnecting a wireless device to/from specified network and a function for gathering results from network scanning. This interface is registered as an RPC skeleton that is used by a client application (implementation enclosed) for managing wireless network connections.

Client applications in HelenOS are always connecting to a specific device managed by its device driver. This is done using the HelenOS IPC mechanism. When processing client requests there are two parts that are communicating with each other. Application-side implementation of functions marshalls arguments into an IPC call together with the identification of the interface and method. If IPC request is received, the service routine is selected from the remote interface which is responsible for IPC communication, unmarshalling parameters and calling the proper high-level callback from connected interface supplied by the driver. This part is implemented by `libdrv` library [5]. An example of the RPC functions implementation is shown in Listing 3.

```
/** Application-side implementation. */
int ieee80211_get_scan_results(async_sess_t *dev_sess,
    ieee80211_scan_results_t *results, bool now)
{
    assert(results);

    async_exch_t *exch = async_exchange_begin(dev_sess);

    /* Send IPC request. */
    aid_t aid = async_send_2(exch, DEV_IFACE_ID(IEEE80211_DEV_IFACE),
        IEEE80211_GET_SCAN_RESULTS, now, NULL);
    int rc = async_data_read_start(exch, results,
        sizeof(ieee80211_scan_results_t));
    async_exchange_end(exch);

    sysarg_t res;
    async_wait_for(aid, &res);

    if(res != EOK)
        return (int) res;
    else
        return rc;
}

/** Remote implementation. */
void remote_ieee80211_get_scan_results(ddf_fun_t *fun, void *iface,
    ipc_callid_t callid, ipc_call_t *call)
{
    /* Get the interface of driver callbacks. */
    ieee80211_iface_t *ieee80211_iface = (ieee80211_iface_t *) iface;
    assert(ieee80211_iface->get_scan_results);

    ieee80211_scan_results_t results;
    memset(&results, 0, sizeof(ieee80211_scan_results_t));

    bool now = IPC_GET_ARG2(*call);

    /* Call the function implemented by the driver. */
    int rc = ieee80211_iface->get_scan_results(fun, &results, now);
```

```
    if (rc == EOK) {
        ipc_callid_t data_callid;
        size_t max_len;
        if (!async_data_read_receive(&data_callid, &max_len)) {
            async_answer_0(data_callid, EINVAL);
            async_answer_0(callid, EINVAL);
            return;
        }
        if (max_len < sizeof(ieee80211_scan_results_t)) {
            async_answer_0(data_callid, ELIMIT);
            async_answer_0(callid, ELIMIT);
            return;
        }

        async_data_read_finalize(data_callid, &results,
            sizeof(ieee80211_scan_results_t));
    }

    async_answer_0(callid, rc);
}
```

Listing 3: Example of RPC functions implementation

User applications are simple programs which just parse input data from clients and call RPC functions. The nature of the problem itself is resolved at the local part of the interface implementation.

In other systems, wireless authentication mechanisms are implemented directly in client applications. For example, processing the four-way handshake is a complex problem and it is better to manage it outside the device driver. In our current implementation the handshake is tightly bound to the IEEE 802.11 framework internals, so it is included in it, but in the future it would be better to create an additional level of abstraction which would expose the necessary data structures of the wireless driver, so that the client application can implement the entire authentication process outside the IEEE 802.11 framework.

3. Implementation

This chapter documents the implementation details of this thesis. It also describes the changes performed to the existing HelenOS source code and outlines the problems which occurred during development.

3.1 Changes to the existing code

Development of the driver required modifications of some parts of the HelenOS source code in order to make driver work correctly. This section documents these changes.

3.1.1 Framework registration

So as to allow client applications to use methods defined in the remote IEEE 802.11 interface, we had to register them first. The device driver remote interfaces dispatch table is located in the `dev_iface` module where we had to create an entry pointing to our interface.

A new category for IEEE 802.11 devices was also added, so user can then identify, list and manage these devices. This involved extending the list of device categories in HelenOS's location service in the `locsrv` task.

3.1.2 DHCP

After successful authentication between a station device and an access point, an IP address needs to be acquired before processing data from the network. This can be managed by the DHCP service, a HelenOS server task, that handles communication between a network device and a DHCP server of the target network.

Before this thesis was implemented, the DHCP client service was invoked immediately after a new network device was initialized in the system. It scheduled a discovery procedure to find the network's DHCP server and if successful, it followed with a DHCP request message for an IP address.

DHCP cannot be used for wireless devices in the same manner because the device is not yet connected to a network at the time of device initialization and the device must first connect to one, following the authentication protocol described earlier.

Because of that we had to separate and expose the DHCP discovery procedure, allowing to invoke it from the IEEE 802.11 remote interface implementation of device connection method after successful authentication.

Another issue related to DHCP communication arose while testing wireless device connection with different types of wireless routers. Some DHCP servers do not send DHCP acknowledgement messages (ACK) by broadcast but rather to a unicast address previously offered to the client. But in the HelenOS DHCP client, the device does not get an assigned IP address until the

whole process of DHCP request procedure is done, making the unicast address unreachable and the message dropped by the network framework. This could be solved by setting the `giaddr` and `ciaddr` fields to zero in the DHCP request message according to RFC 2131 [6]. After that the DHCP server sends the ACK by broadcast, message is correctly received, forwarded to the DHCP client task and the whole procedure is successfully completed.

3.2 AR9271 driver

The wireless device driver is divided into several modules which implement functions from different levels of abstraction. The full source code is located in `uspace/drv/nic/ar9271` directory.

Based on the structure of the Linux driver and device firmware we are using the same interfaces for communication between host and target. As mentioned before in Section 2.2.2, there are three interfaces used for host-target communication. HTC (message transport layer) is implemented in `htc.c` module, WMI (message protocol layer) is located in `wmi.c` module and finally USB implementation of HIF (host interconnect framework) is in `ath_usb.c`.

The module for processing hardware related functions is placed inside `hw.c` file. These are mainly functions related to the device's EEPROM registry.

The crucial part of the driver is in `ar9271.c`. It contains the main function of the driver task, device initialization interface, implementations of IEEE 802.11 and some of NIC framework functions.

3.2.1 Problems during development

During implementation of the driver, we had to deal with several low-level issues that were sometimes quite hard to identify and resolve.

First problem was with determining the communication protocol used in incoming responses to our requests sent to HTC layer. From the source code used in the Linux driver, we first deduced that when reading registry values from the device, the incoming response message contains just the value read from the device's EEPROM. When the function for the reading registry value was executed, it successfully sent the request to the device, but the driver would freeze when initiating procedure for reading the response from device. More precisely, it got stuck in the `usb_pipe_read` function used for reading data from a USB pipe. The reason of the problem was not identified until we used the Wireshark packet analyzer. When we attached it to the respective USB hub where our device was connected, it returned a `-EOVERFLOW` error status after reading from the USB input pipe. We concluded that the problem was due to an incorrect size of the input buffer, deduced that there are several headers included in the response message and finally resolved the issue.

Another problem with processing responses of the device came from the fact that when an incorrect protocol is used, e. g. wrong message type value is used in message header or payload size from header does not match the actual message payload length, the device firmware may get stuck, causing

the `usb_pipe_read` procedure to stall. This is difficult to debug and we had to intercept the Linux driver communication with Wireshark to detect the minor differences in the host-target communication protocol.

Getting the device ready to be able to process network packets is a very complex procedure. There are lots of hardware components to be initialized and setting up single component with slightly different value can cause the whole device to malfunction. We had to examine thousands of lines of Atheros wireless Linux driver source code in detail to find out all the components which need to be initialized.

A big complication was finding the correct initial value of the phase-locked loop (hardware control system of device). In many parts of the Linux driver code there are code branches based on device code identification and revision number. We expected that as device identification number value rises, the value used for the respective property in the program also increases. It is almost always true, but there are some exceptions. This meant that we have originally used an initial value meant for a different Atheros product range, preventing it from processing any network packets. Identifying the source of this issue was a lengthy process. We had to compile the Linux driver, adding detailed debugging output including a list of all component initialization values and then comparing it to the output from our own HelenOS driver.

A fragment of relevant code from the `drivers/net/wireless/ath/ath9k/reg.h` file taken from the Linux kernel 4.1 sources is shown below in Listing 4. It was assumed that as the version numbers of Atheros devices increase, the corresponding values defined by macros would increase too, but this property is broken between the AR9271 and AR9285 models.

```
769 #define AR_SREV_VERSION_9160            0x40
770 #define AR_SREV_REVISION_9160_10        0
771 #define AR_SREV_REVISION_9160_11        1
772 #define AR_SREV_VERSION_9280            0x80
773 #define AR_SREV_REVISION_9280_10        0
774 #define AR_SREV_REVISION_9280_20        1
775 #define AR_SREV_REVISION_9280_21        2
776 #define AR_SREV_VERSION_9285            0xC0  /*< Linearity broken. */
777 #define AR_SREV_REVISION_9285_10        0
778 #define AR_SREV_REVISION_9285_11        1
779 #define AR_SREV_REVISION_9285_12        2
780 #define AR_SREV_VERSION_9287            0x180
781 #define AR_SREV_REVISION_9287_10        0
782 #define AR_SREV_REVISION_9287_11        1
783 #define AR_SREV_REVISION_9287_12        2
784 #define AR_SREV_REVISION_9287_13        3
785 #define AR_SREV_VERSION_9271            0x140 /*< Our device. */
786 #define AR_SREV_REVISION_9271_10        0
787 #define AR_SREV_REVISION_9271_11        1
788 #define AR_SREV_VERSION_9300            0x1c0
```

Listing 4: Linux Atheros wireless driver device numbering

3.2.2 Host interconnect framework

Because there exist multiple interconnection interfaces between host and device (for instance PCIe interface besides USB) used with Atheros wireless devices, we create a unified interface for them to hide implementation details of the individual variants. Every implementation of HIF will then implement specified callbacks which are used by the HTC layer to communicate with the device. The structure of a generic Atheros device is documented below in Listing 5.

```
struct ath;

/** Atheros wifi operations. */
typedef struct {
  int (*send_ctrl_message)(struct ath *, void *, size_t);
  int (*read_ctrl_message)(struct ath *, void *, size_t, size_t *);
  int (*send_data_message)(struct ath *, void *, size_t);
  int (*read_data_message)(struct ath *, void *, size_t, size_t *);
} ath_ops_t;

/** Atheros wifi device structure */
typedef struct ath {
  /** Maximum length of data response message. */
  size_t data_response_length;

  /** Maximum length of control response message. */
  size_t ctrl_response_length;

  /** Implementation specific data. */
  void *specific_data;

  /** Generic Atheros wifi operations. */
  const ath_ops_t *ops;
} ath_t;
```

Listing 5: Communication interface for Atheros devices

For the purposes of this thesis we implemented USB HIF communication located in `ath_usb` module. After successful initialization of the USB framework to configure our device driver, we pass the `usb_device_t` structure pointer to `ath_usb_init` function. In this procedure, pipes for data and control messages communication are mapped onto USB pipe indices to allow for correct communication with the device firmware and sizes of response messages are set up. An `ath_t` structure instance is returned as an output parameter and it is then used within the HTC layer.

Procedures for processing messages between host and target use standard functions from the USB framework for reading and writing data through USB pipes: `usb_pipe_read`, `usb_pipe_write`. Control messages are sent as they come from the HTC layer. Data messages are supplied with prepended header

including the payload size and a special tag different for TX/RX messages to identify corrupted messages.

3.2.3 Host-target communication

Information about the HTC layer is located inside the `htc_device_t` structure and included in the AR9271 driver structure, but it can be reused by different Atheros wireless devices using the HTC communication interface.

During HTC layer initialization, we must first check whether firmware initialization inside device has succeeded. This can be determined by checking that a special ready message is present inside the device. We simply try to read the control message via the control pipe and check if message ID matches the ready message.

If the firmware has been loaded successfully, we initialize the HTC services and store endpoint numbers that will be used for communication between host and target as described in Section 2.2.2. We send special management messages for every service we want to connect and wait for a reply from the device which includes an endpoint identifier to be used in subsequent communication.

After initialization of services is done, we additionally send two confirmation messages indicating that the HTC initialization finished successfully.

The HTC layer defines a messaging interface to send or receive data and control messages, and uses the HIF layer for the actual transmission. HTC related information (target endpoint number and payload length) are prepended to the data buffer the same way as in the HIF layer. Data messages are processed directly by the HTC layer, control messages are handled by the WMI. These are used solely for processing inside the device driver, unlike the data messages which are usually forwarded to the IEEE 802.11 layer and typically further to network and higher layers in case of data frames.

3.2.4 Wireless module interface

WMI comprises interface for communicating with device through several defined control commands recognized by the device's firmware.

`wmi_send_command` handles processing of these commands. It takes a pointer to the HTC device used for communication and a value used for command identification. If the command uses any arguments, then these are passed in an attached command buffer. In order to process the response value we have to allocate a response buffer which the reply is written into. `wmi_send_command` then constructs a WMI command header containing a command ID and a sequence number uniquely identifying the message within the WMI communication. This header is prepended to the message and sent using HTC. We then synchronously wait for the response from the device, regardless of whether the user is interested in the response value or not, because the reply will be stored into an input queue by the target anyway. At this point we ignore RX management frames coming through RX control pipe. These contain detailed information about data messages and also statistics and debugging information.

Several wrappers are prepared for most common commands which are operations on the EEPROM registry, allowing the user to write a value to a specific offset in the registry memory. Values are converted to the correct endianness before being sent. When reading a registry value, the result is parsed from a response message and stored into an output parameter allocated by the caller. There is also a buffered version of the function for registry writing to allow processing more values (passed as an array) at once.

3.2.5 Hardware related functions

Many functions used during device initialization or when switching to different a operating channel are located in the hw module. These usually involve bitwise operations related to the device registry and resetting hardware components to a correct state.

For detailed documentation of AR9271 device register description, refer to the device's official data sheet [7]. However, lots of information is missing there, so for complete description of device registry, one has to study the Linux driver source code [8], mainly the reg.h file.

3.2.6 Device initialization

In this section we describe what has to be done after plugging in a AR9271 device before we are able to process any network packets with the device.

Registration with device manager

First, we have to tell HelenOS that we want to manage an AR9271 device by our driver. This is done through the HelenOS's device manager (DevMan) which probes the drivers installed in the system for the list of devices these drivers support. Every device driver is associated with a .ma file that identifies the supported devices. Each line in the `.ma` file identifies one or more devices. It contains a priority value, i.e. how suitable the driver is for such device, and a hardware identifier that differs by the interconnection framework used for device. In our case the USB device contains a vendor ID and a product ID as documented below.

```
10 usb&vendor=0x0cf3&product=0x9271
```

Initialization inside USB framework

The design of a USB bus is organized into a star topology. In the root, there is a USB host that may implement several *host controllers*, each providing one or more USB ports. Multiple USB devices are linked to USB hubs which can extend a USB network to a maximum of 127 ports.

There exist several host controller standards that allow a host controller for USB to communicate with a host controller driver in the operating system. Open Host Controller Interface (OHCI) and Universal Host Controller Interface (UHCI) are based on the original USB 1.x standard. OHCI is a common interface on PCI cards based on the NEC chipsets. UHCI is a proprietary

standard created by Intel. The main difference between these two is that UHCI is more software-driven than OHCI. USB 2.0 devices are using Enhanced Host Controller Interface (EHCI) standard which can operates up to 40 times faster. However, USB 2.0 support is not yet complete in HelenOS, so devices need to use either the UHCI or OHCI standard.

Communication with USB devices is based on *pipes* which connect the host controller with logical entities on device called the *endpoints*. We distinguish two types of pipes. *Message pipe* is bidirectional and it is used for *control transfers*. *Stream pipes* are unidirectional and they are used for transferring data in various modes of transfer. *Interrupt transfers* are used for small transfers when we need the maximum duration of the data transfer to be guaranteed. *Bulk transfers* are used for large amounts of data that are transferred rather sporadically. It has a character of non-periodic and bursty communication. Finally *isochronous* type of transfer is used for time-dependent communication such as multimedia streams. This mode of transfer is not supported in HelenOS yet.

AR9271 device uses the UHCI interface. In HelenOS UHCI root hub is implemented in the microkernel manner as a server task. For every USB port in the system there exists a separate thread executing routine that periodically checks the port status and reports new devices. When new device is attached to the system, new host controller connection is registered by the routine, a control endpoint is initialized and the device is registered with DevMan.

After UHCI initialization is done, the system starts the main routine of the driver task.

Driver frameworks interconnection

In the main function of the device driver, we merely initialize logging and pass control to the DDF.

DDF registers the driver and separates its execution into its own fibril. After the driver is successfully setup within device driver framework, the DDF device structure is allocated and configured, and the callback for passing a new device to the device driver is invoked. Here our driver structures and connected frameworks are initialized.

We allocate the USB device structure and initialize it using the bypassed mechanism as described in Section 2.4.1. Before that we have to configure the USB endpoint descriptors to tell the driver which pipes are used for control and which for data transfers, what type of transfer they use, what are their directions, interface protocols the endpoints belong to, etc. We can find this information for example by connecting the device to a computer running some conventional operating system like Linux or Windows and by using the utility for displaying information about USB buses in the system. In Linux, for example, this is be with the `lsusb` command with the `-v` flag for a verbose listing.

Afterwards, the remainder of frameworks are initialized in a dependency-satisfying order. First, the HIF USB device is initialized followed by the IEEE 802.11 framework structure initialization (that includes setup of the NIC framework), and finally the HTC device structure is allocated and configured.

All pointers to the connected device structures and frameworks are stored in the AR9271 structure, so we can forward operations specific to each one of them.

Warming up the device

Once all structures and frameworks are allocated, we can move to initialization of the device itself.

First steps is uploading the firmware to the device. It is a software program providing necessary instructions for communicating with the host computer. We use the official open-source firmware by Qualcomm Atheros (still under development) which can be downloaded from the Qualcomm Atheros GitHub repository [9]. The firmware file is copied inside AR9271 HelenOS driver directory structure, so it can be loaded during device probe from disk into the memory.

Because the firmware is too large to be sent into the device at once, it is uploaded in chunks that fit inside the USB control pipe. USB specification defines standard device requests sent from host to device using so called setup packets. Firmware transmission uses the USB SET request wrapper implemented in HelenOS to transfer every chunk of firmware into the target device.

After the firmware is loaded, we send a confirmation message using another USB SET request to tell device that whole firmware is loaded inside device RAM at a given address. The device then initiates a firmware configuration phase that may take up to 50ms according to Qualcomm developers. We wait for 200ms to be sure.

The HTC framework initialization follows, including checking the ready message via the input control pipe which should have been created by the device after the firmware has been successfully configured and became ready for communication. More details are available in Section 3.2.3.

Next phase is initialization of the hardware related parts. Majority of this is represented by correctly setting device registry values. Sometimes we have to wait until the process of registry setup is done inside the device before we continue executing the next batch of operations, because it can be connected for example with resetting or powering on some hardware parts of device. To this end, the `hw_read_wait` polling function is implemented to wait until the registry contains an expected value at a given address.

First, however, we have to do a warm reset of the device, followed by reading the device's hardware address (MAC) from EEPROM and reporting it to the NIC framework.

To indicate a successful reset of the device, we also turn on the device's green LED light. This is done by configuring the general purpose I/O (GPIO) port according to the device datasheet. There exist more features that the LED light can perform, including setting its blink rate. Currently it is used only to indicate the power state, but the API for managing GPIO modes and values is ready and can be used more in the future.

Afterwards, the device needs to be initialized within the IEEE 802.11 framework and NIC handlers registered. The NIC framework registers a new de-

vice and invokes a callback indicating that the client has connected to the device. This callback is implemented by the IEEE 802.11 framework and after it handles its own configuration, it forwards the start-up initialization to the AR9271 driver.

Initialization of the wireless device in station mode requires the following steps, mostly WMI commands sent to the device.

First we have to tell the device to flush the receiving messaging queue because we are going to reset it. Hardware initialization is implemented in the `hw` module. We have to deactivate the device's physical layer, which means that no packets can be processed during the initialization. We reset the device and set up the phase locked loop component for operation on 2.4GHz frequencies.

Then we have to initialize the device's EEPROM with a sequence of offset-value pairs. This part is a bit tricky because the meaning of the values is not properly documented and the device will not work properly if set up wrong. We have to copy them from Linux sources and use them as they are. Fortunately, the license of the Linux driver source code uses the ISC software license, compatible with the BSD license used by HelenOS.

We set the device to operate in station mode on a default 2.4GHz channel and pass it on to the IEEE 802.11 layer. Now we can reactivate the physical layer and initiate device calibration procedures. In the reference Linux driver, calibration is performed by continuous processing of data error rate statistics. In our implementation, we set up calibration data only statically during the switch to the different operating channel or during the device initialization.

To process only specific type of network data, we configure RX filtering. By default we accept all types of data packet routing (unicast, broadcast and multicast) together with beacon management frames. When connection with the target access point is initiated, we configure filtering to accept only beacons from our network and normal data packets.

Special message indicating successful station mode initialization is sent to HTC layer of the device.

At the end of the whole process, we create separated fibril to process incoming data from the device and tell the IEEE 802.11 framework that the device is ready to process data from the network.

The entire procedure of device initialization is symbolized in the diagram in Appendix D of this work.

3.2.7 Processing network packets

Data from the network are processed by invoking `htc_read_data_message` in a loop. This function uses the USB HIF layer to request data from the respective USB pipe. When there are no data pending, the data polling fibril is suspended to allow other threads to take over.

It should also be noted that if encryption is enabled, incoming data from the network are decrypted by the device itself and then sent over the USB pipe in plain texts.

Because one packet might be split into multiple data messages, the process is a little bit complicated.

If the received message is smaller than the known header size, it is immediately discarded because it contains only garbage. If the tag value from the HIF USB header does not match the RX tag defined by the protocol, the message is dropped too.

We retrieve an RX status part of the message to check if the payload is not corrupted. If it is, we can exit processing this message only if it is not marked as an aggregated or split packet. Otherwise we must process it anyway to find out the offset where the following aggregated packet starts. Split packet is recognized as a packet where the data buffer size is greater or equal to the maximum message length that can be processed by a USB pipe. Buffer size can be greater if merged with the following message during processing of an aggregated packet, as described below. Aggregated packets start somewhere inside the data message, never at the start.

Complete size of the packet is read from the HIF USB header. Then, if the current message is marked either split or aggregated, we read the remaining packet fragments from the device by calling `htc_read_data_message` again. The resulting packet is then forwarded to the IEEE 802.11 layer.

If there are more packets aggregated in the data buffer, i.e. when there are still some data remaining, we are going to process them by recursively calling the procedure on the rest of the data. If there is just a part of the following packet header in the current data buffer, we have to also fetch the subsequent message and merge it to current buffer to be able to get all the necessary information about this packet during its processing.

To give you some perspective about the whole process of processing data packets, its source code is included below in Listing 6.

```
static void ar9271_process_data(ar9271_t *ar9271, void *buffer,
    size_t max_size, size_t size, bool aggr)
{
    size_t strip_length = sizeof(ath_usb_data_header_t) +
        sizeof(htc_frame_header_t) +
        sizeof(htc_rx_status_t);

    /* Too small message. */
    if(size < strip_length)
        return;

    ath_usb_data_header_t *data_header =
        (ath_usb_data_header_t *) buffer;

    /* Invalid packet. */
    if(data_header->tag != uint16_t_le2host(RX_TAG))
        return;

    htc_rx_status_t *rx_status = (htc_rx_status_t *) ((void *)buffer +
        sizeof(ath_usb_data_header_t) +
        sizeof(htc_frame_header_t));
```

```
    bool aggr_or_split = size >= max_size || aggr;
    bool packet_corrupted = ar9271_rx_status_error(rx_status->status);

    /* We can exit processing this data chain, because there should
     * not be any unaligned packet in the following message. */
    if(packet_corrupted & !aggr_or_split)
        return;

    /* Read remaining packet fragments from device. */
    void *it = buffer + size;
    size_t packet_size = uint16_t_le2host(data_header->length);
    int remain_length = packet_size - size;

    while(aggr_or_split && remain_length > 0) {
        if(htc_read_data_message(ar9271->htc_device, it,
            max_size, &size) != EOK)
            return;
        it += size;
        remain_length -= size;
    }

    if(!packet_corrupted)
        ieee80211_rx_handler(ar9271->ieee80211_dev,
            buffer + strip_length,
            packet_size - strip_length);

    /* Process aggregated packets in data buffer. */
    if(aggr_or_split && abs(remain_length) > CRC_LENGTH) {
        size = 0;

        /* There is just a part of the header of following packet in
         * the current message, so we fetch the next message to be
         * able to get all information. */
        if(abs(remain_length) - CRC_LENGTH < (int) strip_length) {
            if(htc_read_data_message(ar9271->htc_device, it,
                max_size, &size) != EOK)
                return;

        }

        /* Start of the packet is always aligned to multiple of 4. */
        remain_length = (remain_length + 3) & ~0x03;

        ar9271_process_data(ar9271, it + remain_length + CRC_LENGTH,
            max_size, abs(remain_length) - CRC_LENGTH + size, true);
    }
}
```

Listing 6: Processing the data packets in AR9271 driver

3.2.8 Wireless framework related functions

Here we describe the callbacks implemented by the IEEE 802.11 framework. Start-up callback is already described in Section 3.2.6.

`tx_handler`

Data coming from the IEEE 802.11 layer are merely provided with a configured HTC TX header to recognize a packet by the target device's firmware. Management frames are sent unencrypted to the respective HTC management endpoint. When processing data frames, we first query the IEEE 802.11 layer if encryption has to be used and eventually set up the appropriate key type and index in header based on the security suite used for communication. Data are then encrypted by the device itself. For data frames, the HTC best effort endpoint type is always used without analysing the data contained in the message. This could be improved in the future.

`set_freq`

Setting the device's operating frequency first involves telling the device with WMI commands to stop receiving packets and to drop all unprocessed packets from its TX queue. Then the frequency switch itself takes place by calling the hw module. We must request access to device's physical layer and once granted, we can set the respective device registers to configure device to operate on the new frequency. After that we report the new frequency to the IEEE 802.11 layer and reactivate the physical layer. Noise floor is calibrated if the 2.4GHz band was selected, and finally the device is told to start receiving packets again and to set up RX filtering.

`bssid_change`

When the network connection state is changed, we have to configure the device accordingly. This takes place during the (de-)authentication stage.

Once the the device connects to a network, we query its BSSID from the IEEE 802.11 layer. A special management message is constructed and addressed to the HTC layer using which we create a virtual node in the device we are going to communicate with. Also, we have to set up data rates we are able to communicate on. As mentioned before, we only support legacy 2.4GHz rates at the moment.

During device deauthentication from network, we just send a message to remove the virtual node from the device.

In the end, we configure the RX filter with respect to the network connection state and write the BSSID into the device registry..

`key_config`

In order to use encrypted data transmission, we must first configure encryption keys in the device. AR9271 devices use hardware encryption of data

frames, so it is enough to just write keys into the device, mark the frames as protected and the device will handle its encryption and decryption externally.

Target network BSSID is used together with the keys for data encryption, so we first query the IEEE 802.11 layer to retrieve it from the device where it was previously stored. Based on the key type (group or pairwise), we select a location inside the device registry where it will be written. Because registry entry cannot handle the whole key, it is divided into several smaller parts stored consecutively in the device memory.

In case of TKIP security suite, we also have to store the MIC used to authenticate the message which is a part of the key data. This is done similarly to the key itself.

Having configured both the pairwise and group keys, we inform IEEE 802.11 that the data frames are protected now.

3.3 IEEE 802.11 framework

Structure of the IEEE 802.11 framework source code is divided into several parts based on the functionality it serves.

Interface for wireless device drivers together with framework internals implementation are located in the `uspace/lib/ieee80211` directory. Implementation of the core features, which are processing frames, framework initialization functions and IEEE 802.11 device structure getters and setters, is located in the `ieee80211.c` file. Default implementation of the device driver interface and other functions related to the IEEE 802.11 framework are in `ieee80211_impl.c`. IEEE 802.11 user interface functions have their default implementation in the `ieee80211_iface_impl.c` file. Respective headers are stored in the include subfolder, but only `ieee80211.h` is supposed to be used outside of the IEEE 802.11 library folder. This one should be included by wireless device drivers.

User applications that want to interact with IEEE 802.11 devices should use the interface defined in the `uspace/lib/drv/include/ieee80211_iface.h` file. These functions are implemented in an RPC manner inside the `uspace/lib/drv/generic/remote_ieee80211.c` module. User applications call stub functions which send IPC requests to the server. There arguments are processed and passed to the respective implementation within the connected IEEE 802.11 device. After it has executed the function, results are sent back from the IEEE 802.11 framework, through the server skeleton and to the client stub. The structure of user interface operations is defined in `uspace/lib/drv/include/ops/ieee80211.h` and it is passed to the server implementations which use it to delegate the function call further to the implementation of the interface inside the wireless device.

Common definitions of both user and driver interfaces are located in `uspace/lib/c/include/ieee80211/ieee80211.h`.

Only data and management frames are handled by the framework as the control frames are managed by the devices themselves.

3.3.1 Frame format

Frames used in the IEEE 802.11 framework have uniform structure. Both data and management frames have the same header format pictured below in Figure 3.1.

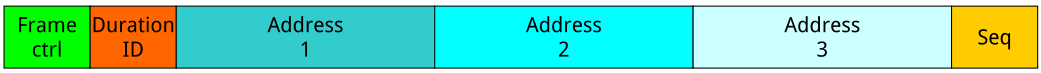


Figure 3.1: IEEE 802.11 frame header format

Header is followed by the message payload that can be of variable size depending on the frame type and the protocol. The last item in the IEEE 802.11 frame is the *frame check sequence* (also known as cyclic redundancy check) that allows devices to verify the integrity of received frames.

Some frames have an additional `Address 4` that is used when transferring packets within a distribution system (between two access points). Since we only support station mode, this frame variant is not included in the implementation.

In some kinds of frames, the message payload is composed of parts that have a common general format. These are called *information elements* (IE) consisting of several fields: element ID, length and then information field of given length. All information element types are described in the IEEE 802.11 specification.

Frame control

This field identifies the type of the frame. Its structure is separated into several subfields consisting of one or more bits. First two bits represent the protocol version which is currently zero with other values reserved for future use.

`type` Frame type indicator — either control, management or data.

`subtype` Depending on the type it describes a specific subtype of the frame, e.g. authentication, association request, RTS, etc.

`to DS`, `from DS` Flags which represent the frame is travelling from or to a distribution system which is used later to determine the meaning of address fields.

`MF` Flag indicating if more fragments of this frame will follow.

`retry` Indicator of packet retransmission.

`PM` Station power saving mode indicator.

`MD` Indicates whether the AP has pending data for the station. This is used for power management purposes by the station to decide whether to go sleep or stay awake.

`PF` Protected frame field indicating whether the frame is transferred as clear text or if it is protected with encryption methods like WPA.

`order` Strict ordering support indicator used when QoS is supported.

The current implementation does not support the `MF, retry, PM, MD, order` fields.

Duration/ID field

Depending on two most significant bits, this field is used for different purposes. In control frames it is used to carry station AID, but mainly it is used to indicate how long it will take to transmit the frame through the air. This helps the other stations in the network to coordinate when to start their own transmissions.

This field is not used in the implementation of this thesis as power management is not supported yet.

Address fields

These fields are used to determine the addresses of the sender and the receiver. Third address is used for filtering purposes. The order of the addresses is defined by DS fields as described before.

Sequence control

This number consists of two subfields. First 4 bits are used as the frame fragment number and the rest of the value identifies message order in the communication between the sender and the receiver.

3.3.2 Initialization

The main task of the IEEE 802.11 layer is obviously processing packets within WiFi networks. Besides that it must also initialize the wireless part of the device driver structure and integrate it into HelenOS environment.

The process starts with initializing the IEEE 802.11 device structure itself. Device driver first calls `ieee80211_device_create` which is a wrapper for allocating memory for the wireless device structure. Its members are then initialized with `ieee80211_device_init` including binding connected NIC to the respective DDF device and setting the NIC driver specific structure to IEEE 802.11. The resulting relationship is described at the end of Section 2.4.1.

The wireless driver has to implement several interfaces and pass them to `ieee80211_init` which is the final step in the IEEE 802.11 layer initialization. These interfaces are: `ieee80211_ops_t` structure with WiFi framework callbacks, `ieee80211_iface_t` which holds application interface implementations, `nic_iface_t` with functions related to the NIC application interface and `ddf_dev_ops_t` which are NIC device operations currently unused by the device driver. When some implementations are left unfilled, default ones from

the IEEE 802.11 layer are used, except the WiFi framework callbacks which are mandatory and need to be implemented by the device driver. The only WiFi framework callback with a default IEEE 802.11 layer implementation is `scan`.

In order to process frames from the NIC layer we need to register a TX handler using `nic_set_send_frame_handler`, create a DDF function and bind it to a connected NIC device.

In the end, we register the resulting device function with DevMan and add it to NIC and IEEE 802.11 categories to be recognized by both applications using generic network devices and just the wireless ones.

Then NIC framework probes the new device and calls `open` callback implemented in the IEEE 802.11 layer. It passes control to the device driver by calling the `start` callback and, once the wireless device driver is ready, we create a `ieee80211_scan` fibril for periodic network scanning. It simply invokes `scan` in defined intervals. Using `async_usleep` ensures that only the running fibril is suspended and not the whole thread.

3.3.3 Network scanning

Generic implementation of software scanning for neighbour networks is located within IEEE 802.11 device operations implementation in `ieee80211_impl.c` file. Unless the other implementation is supplied by the user, this one is used in the scanning fibril.

List of discovered networks is stored inside the IEEE 802.11 device structure in `ieee80211_scan_result_list_t`. Data of each entry are held by a `ieee80211_scan _result_link_t` structure.

Procedure is protected by `scan_mutex` lock inside the IEEE 802.11 device structure, because user should not initiate connection process during scanning as it requires information about the connecting network from this scanning results shared structure.

Scanning is initiated only if the station is in disconnected state, because it involves changing the device's operating frequency that would interrupt the connection between the station and the access point.

First, old entries are removed from the scan results. This operation is also guarded by the mutex to prevent adding newly discovered networks while iterating over the scan results structure. We store the time of the last beacon that came from each known network and if the time that has elapsed from this moment is greater than a defined threshold, we remove the entry from the results.

The scanning itself is a straightforward process. First we store the original frequency the device was operating on before the scan. Then we iterate over all available channels in the 2.4GHz spectrum. The `set_freq` callback is invoked on each channel and a probe request message is constructed. Device stays on each channel for a defined amount of time, waiting for incoming beacons and probe responses before switching to the next channel. In the end, we switch to device to its original frequency and unlock `scan_mutex`.

Probe request is a management frame whose payload is composed from a sequence of values including the network SSID, data rates supported by the

station and the channel number.

3.3.4 Basic authentication

Before processing packets from the selected network, basic authentication needs to be performed as explained in Section 1.2.2.

The whole process is driven by the `ieee80211_iface->connect` operation from the user interface part. This function is invoked by the user application when it wants to connect to a specific network from the scan results list.

We check if the device is in a ready state, otherwise the process cannot proceed. If it already is connected to a network, we initiate the disconnection operation that involves sending a deauthentication message to the currently connected access point. This message is described in more detail later.

Scanning and connecting procedures cannot run concurrently as described in the previous section, so we must wait until the scan procedure has finished if currently running. Because that might take a long time, we set up a connection request indicator. This flag is checked on every channel iteration during the scanning procedure and if it is set, it immediately breaks the execution and unlocks the `scan_mutex` so we can start the connection process.

Once we are allowed to enter the critical section, `scan_mutex` is unlocked, connection request indicator is cancelled and we find if SSID provided by user is matching any of those in scan results list. If we have found a matching network, we can initiate the basic authentication process.

Information about the network we are connecting to is stored inside the IEEE 802.11 device in `ieee80211_bssid_info_t`. There is a link to the respective scan result entry and all other information needed by the IEEE 802.11 framework and the device driver during authentication.

First, we switch to the target network operating channel. Because channel numbers are used in network frames but frequency values are used within device drivers, functions for converting between these two formats are available

Basic authentication starts with sending an authentication request managed by the `ieee80211_authenticate` function. Frame format is the same for requests and responses, and its body is defined in the `ieee80211_auth_body_t` structure. First we load information about the network we want to connect to. We set up a frame header with the respective frame type and subtype and fill in the MAC addresses of both parties. Because we do not support the deprecated WEP shared key authentication, we use fixed values for the authentication algorithm indicators corresponding to an open system mechanism used for all other variants. Then we simply send the constructed buffer using the IEEE 802.11 TX handler.

After that, the user connection process is suspended as it waits on a conditional variable until the response is processed. Waiting has a defined timeout so the process gets terminated if no response is received. In that case, a deauthentication message is sent to the target network.

When an authentication response is received, its status is checked to verify that everything went without any problems. Based on this we set the device's current authentication phase with `ieee80211_set_auth_phase` and

send a signal to the user connection process using the conditional variable that the response was processed.

Constructing association request proceeds in a similar manner. Additional information elements are included, same as in the probe request message. If encryption is enabled, a privacy flag inside the capability field of association message body is set and moreover, if any of IEEE 802.11i security suite is used, RSN security information element taken from scan result entry is also added in the message payload.

The process of waiting for an association response is the same as with an authentication message. During response processing, AID is taken from the message and stored inside the IEEE 802.11 device structure. `bssid_change` callback is then called to configure the device for operating in the new network. The AID value is used by the device driver to set up filtering of packets destined for our station from the given BSSID.

If anything went wrong, deauthentication is sent to the target access point. The message has a simple management frame handled the same way as the other messages before. When deauthentication message is sent, information about the connected BSSID in `ieee80211_bssid_info_t` is cleared and `bssid_change` is called to notify the device that we are not currently connected to any network. If encryption was configured, we also have to deconfigure security keys in the device with the `key_config` callback.

Based on the network security used in the target network, we have either finished authentication process and therefore set the device state to connected, or high-layer authentication, described in the next section, needs to follow.

3.3.5 4-way handshake

The general handshake sequence is described in Section 1.2.4.

All phases of the handshake are processed in one procedure because they share a lot of common code. Communication between the client and the authenticator uses EAPOL-Key frames whose structure is shown below in Figure 3.2.

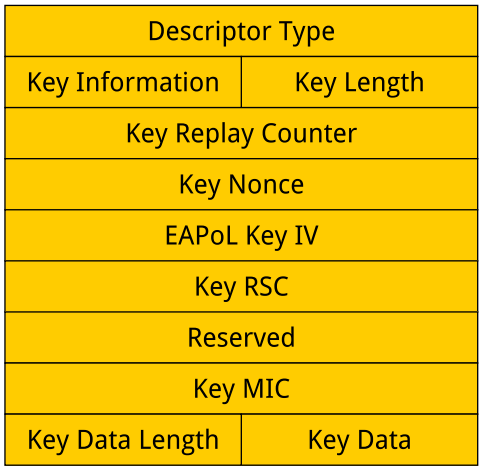


Figure 3.2: EAPOL-Key frame format

Individual phases can be distinguished by differences in their key information field that is used to specify characteristics of the key being transferred. When neither MIC nor the secure bit is set inside key information field, i.e. MIC is not included in message and encryption is not used, then we deduce that this is the first message from the authenticator. Then we have to distinguish between the WPA and WPA2 security suites because the old WPA is using a slightly different approach and the group key is transferred in an extra 5th message as described in this post [12]. Hence we use a `third_or_extra` variable based on the MIC bit to identify the 3rd or 5th message of the handshake, and `key_phase` set from the secure bit to identify the final message with the group key which is the 3rd using WPA2, but the extra 5th when WPA is used.

We clone the incoming EAPOL frame and use it then as the outgoing frame, because usually a lot of field values remain the same. After processing a message from the authenticator, we send back the EAPOL frame protected by MIC. It is computed using a keyed-hash message authentication code (HMAC).

First message contains enough information to derive the pairwise encryption key (PTK). Using password and target network SSID, the pre-shared key (PSK) is generated. This is done with the key derivation function PBKDF2 defined in the IEEE 802.11 standard. SSID is passed into the function as salt, which are just additional data that modify the resulting password hash to provide better security. In case of the WPA-Personal mode, this PSK key is used directly as the pairwise master key (PMK). Then we can derive the PTK using PMK together with a concatenated source and destination address, and the ANonce and SNonce random values. These arguments are passed into the IEEE 802.11 pseudorandom function to generate the resulting PTK.

The key has three parts used for different purposes. Key confirmation key (KCK) is used in the follow-up authentication messages for MIC computation. Key encryption key (KEK) is for encryption of the GTK key. And finally temporal key (TK) is used for the encryption of data. In TKIP security suite, there are also two additional keys used for computation of TX and RX MIC in data frames.

In its second message, the client sends a reply to the authenticator which contains the generated SNonce and a security information element.

Third phase is the GTK derivation. Based on the encryption protocol used, we have to decode the GTK key from the encrypted message. If CCMP is used either as a pairwise or a group security protocol, the NIST AES key wrap algorithm is used. Otherwise it is calculated using the ARC4 algorithm. Then key data are parsed from the encoded stream. With WPA2 security suite the data are enclosed in an information element. In the case of WPA, the encoded stream directly corresponds to the GTK key data.

After this, we can configure both the PTK and the GTK inside the device driver using the `key_config` callback. WPA only installs PTK after its third phase and GTK is then processed after the extra message is received.

Successful handshake completion is signalled to user connection initiating process using a conditional variable and the wireless device is marked as connected.

Encryption algorithms used during the handshake are documented in Section 3.4.

3.3.6 Processing frames

The primary task of the framework is processing the outgoing and incoming frames. Both directions are described separately in the subsections below.

TX handling

Data frames coming from the NIC layer to be sent over the air are managed inside the `ieee80211_send_frame` function. This callback is registered with the NIC framework during IEEE 802.11 framework initialization. Frames are processed only if the device is connected to a network, otherwise they are dropped immediately. Based on the used security suite we have to reserve enough place in header and footer of the frame for additional encryption data. Frames of the IEEE 802 family use additional encapsulation sequence defined in RFC 1042 that we have to append after the frame header [10]. Frame control fields from the frame header are filled with respect to frame TX direction and a protected frame field is set if the frame has to be encrypted. If TKIP security suite is used, we have to compute Michael MIC from the whole frame data and append it. Currently only hardware encryption is supported, but the `ieee80211_encrypt_frame` function is ready for software encryption of frames to be implemented in the future. Resulting buffer with data is sent using `tx_handler` to the device driver for further processing.

RX handling

The device driver manages incoming packets from the wireless device and passes it to the IEEE 802.11 layer. According to a frame control field in the header, it either passes it further to a function for processing management frames or to a function for processing data frames.

Processing of management frames is subsequently divided by the frame subtype. Probe responses and beacons are managed together, because they use a very similar format. First it is checked whether the SSID information element carries any information. Some hidden networks could block broadcasting of their SSID and these frames are useless for our purposes. Then we parse the SSID from IE and check if it is already present in the scan results. If so, we just update their beacon timestamp and exit processing of frame. Otherwise we create a new `ieee80211_scan_result_link_t` entry for this network and fill it with information about the network from the beacon frame (BSSID, SSID, WEP indicator). Some information elements are not placed in a fixed order and there might also be some elements that we are not interested in. Therefore a simple `ieee80211_process_ies` handler was written to parse this information. Security related information is parsed by another function called `ieee80211_process_auth_info`. When using the WPA security suite, this information is located in vendor IE. In the case of WPA2, it is located in RSN IE. The location of the WPA related information was not easy to find

out, because WPA was implemented in a draft of IEEE 802.11i but official documentation is missing. The information was eventually found on [11]. Security IEs are stored in the scan result entry and used later when sending an association request. Other management frames for basic authentication were described earlier.

Data frames are stripped off the IEEE 802.11 header and converted to an 802.3 Ethernet header for processing in higher layers. An NIC frame is constructed from the data buffer and sent to the NIC layer with the `nic_received_frame` callback. During frame processing we check the frame protocol type and in case of the EAPOL frame, we pass it to the four-way handshake procedure. There is also an empty function `ieee80211_decrypt_frame` prepared to be implemented to support software frame decryption

3.3.7 Driver interface

Only necessary functions are exposed to the driver interface. These are the IEEE 802.11 framework initialization functions to allocate and configure wireless framework structure, getter and setter functions to wrap all manipulation of inner attributes and structures, frame type indicators and finally a callback interface that is called by the framework during important events that cannot be handled by the framework itself and must be provided by the device driver.

Together with the function interface, there are several structures and constants exposed to the device driver which it shares with the framework.

The whole interface is defined in file `uspace/lib/ieee80211/include/ieee80211.h` and can be included in the device driver by including the `libieee80211.a` library in its `Makefile`.

Getters and setters

When the device driver wants to get or set any information related to the IEEE 802.11 framework, it uses solely these wrapper functions. Every wrapper is guarded by a `gen_mutex` lock to avoid concurrent execution of procedures by the device driver and user application process accessing the framework via IPC.

Callbacks

All mandatory operations that the framework needs to be provided by the device driver are defined in the `ieee80211_ops_t` structure (except `scan` that has a default implementation in the framework). These functions have already been described before in the places of their use, so we only give a definition of interface to recapitulate in Listing 7. Comments were reduced to not take up too much space.

```
typedef struct {
  /**
   * Function that is called at device initalization. This should
   * get device into running state.
   */
  int (*start)(struct ieee80211_dev *);

  /**
   * Scan neighborhood for networks. There should be implemented
   * scanning of whole bandwidth. Incoming results are processed by
   * IEEE 802.11 framework itself.
   */
  int (*scan)(struct ieee80211_dev *);

  /**
   * Handler for TX frames to be send from device. This should be
   * called for every frame that has to be send from IEEE 802.11
   * device.
   */
  int (*tx_handler)(struct ieee80211_dev *, void *, size_t);

  /**
   * Set device operating frequency to given value.
   */
  int (*set_freq)(struct ieee80211_dev *, uint16_t);

  /**
   * Inform device about BSSID change. This should be called
   * everytime device connection state changes.
   */
  int (*bssid_change)(struct ieee80211_dev *, bool);

  /**
   * Setup encryption key in IEEE 802.11 device. Boolean variable
   * is used to indicate if the key should be inserted or removed.
   */
  int (*key_config)(struct ieee80211_dev *, ieee80211_key_config_t *,
      bool);
} ieee80211_ops_t;
```

Listing 7: IEEE 802.11 driver callback interface

3.3.8 Application interface

Interface for user applications comprises just few functions so far, but it should manage the basic functionality a user needs to operate the WiFi device. The interface currently consists of:

- `connect` Manage connection between the wireless device in station mode and a target network including the whole authentication process and configuring the respective network interface.
- `disconnect` Deinitialization of a network interface and deauthentication of the device from the network.
- `get_scan_results` Fetching the list of scan results from the wireless device.

This is the interface exposed by the wireless framework. It is invoked by a server-side RPC skeleton when processing user application requests from a client-side stub.

Framework side implementation

Functions for managing device connection state were already described before in Section 3.3.4. Disconnection procedure simply checks if a device is already connected to a network and if so, calls the function for deauthentication.

Transmitting scan results is a fairly simple operation. If user requests fresh results, the scan procedure is immediately initiated and the user is blocked until it is finished. Then we copy the result entries into an array passed as an output parameter to the procedure. The copying is necessary here because we cannot pass an internal pointer to the client who is running in a separate process.

The whole procedure is guarded by `results_mutex` which is used in all situations when manipulation of the scan results list takes place. It can be accessed either by a user application or by the IEEE 802.11 framework itself.

RPC stub implementation

User applications access the interface using a client RPC stub that forwards their requests to a server side skeleton and further to the IEEE 802.11 library.

The main job of these functions is to marshall arguments and send them over IPC to the server side. But during connection and disconnection there is also another thing they must handle.

After connection procedure has been successfully completed and positive response has been received from the server skeleton, we have to configure the device's IP address, otherwise it would be useless for networking purposes. IP address can be configured by the DHCP client in HelenOS. We call the `nic_get_address` function to retrieve a MAC address of our interface and then we use it to search for `link_id` of a corresponding entry registered with the `inetsrv` IP server by executing `inetcfg_get_link_list` and searching for an entry matching our MAC address. This identifier is then used to initiate DHCP discover procedure that manages the IP address configuration.

When disconnecting a device from a network, we additionally have to remove the previously obtained DHCP address. This is done by iteration through the list retrieved from `inetcfg_get_addr_list` and deleting the matching entry. After that we also have to remove the corresponding DHCP

static route. This is currently done by searching for route with the currently unique `dhcp` identifier.

3.4 Cryptographic functions

During the thesis development a new cryptographic library was implemented. It comprises functions and algorithms needed for the four-way handshake, but many of these could be also used for other purposes.

Generic functions are packed inside `libcrypto` library and IEEE 802.11-specific functions are located inside `ieee80211_impl.c` file inside the IEEE 802.11 library.

3.4.1 AES

Currently the most widely used WPA2 security suite introduced in the IEEE 802.11i amendment, based on the AES algorithm defined in the FIPS 197 standard [13]. It is a symmetric block cipher created to replace its predecessor DES. Besides being used in the WPA2 protocol, it is used by the U.S. government to protect sensitive information.

We implemented the AES-128 variant using 128bit encryption keys which is used with WPA2. As the computation of AES is rather complex it is placed in separate `aes` module inside the `libcrypto` library.

The encryption algorithm begins with a derivation of series of round keys from the input encryption key. Input text is then copied into the state array that will be used as an intermediate cipher result. Processing loop follows, separated into several steps. Individual iterations are referred to as rounds.

1. **Bytes substitution** — each byte is replaced by another one according to a substitution table.
2. **Shift rows** — cyclic left shift of last three rows by offsets according to their indexes.
3. **Mix columns** — transformation operating on individual columns of the state array by combining their values.
4. **Add round key** — each byte of the state array is combined with a block of the round key using the XOR operation.

In the first round only the `add_round_key` operation is executed and in the last round the `mix_columns` function is omitted. At the end the cipher is stored in the state array and copied into the output buffer.

Individual operations within the processing loop are illustrated on the picture under in Figure 3.3.

Implementation follows the specification of algorithm from the published standard.

Decryption algorithm uses inverse operations during the round phases, and the operations are arranged in a different order.

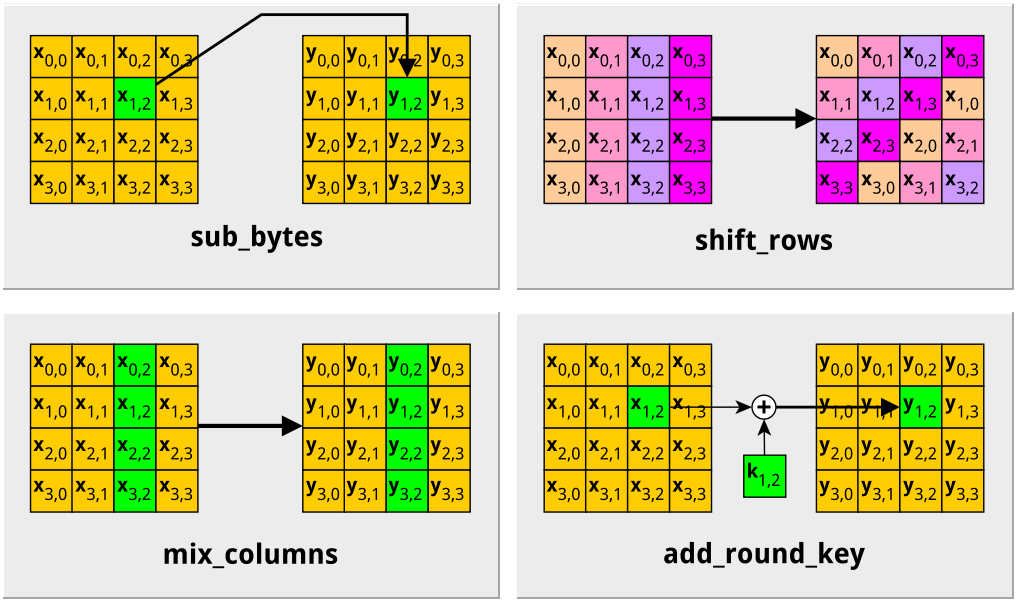


Figure 3.3: AES round phases

3.4.2 RC4

The WPA protocol uses the RC4 (also known as ARC4) stream cipher. It is a simple and fast algorithm that is also used in the HTTPS protocol, but it is more vulnerable than AES. It uses exactly same algorithm for encryption and decryption.

The algorithm uses variable length encryption keys up to 256 bytes long. These bytes are subsequently used to create a substitution box consisting of permutations of all 256 bytes. Their order is based on the values in the key stream. This phase is also known as the key-scheduling algorithm.

Processing loop iterates through the input text while incrementing two indices. One increases linearly, the second based on values in the substitution box. Values at those indices inside the substitution box are swapped and then the same indices are used to retrieve the value from the box as follows: `sbox[sbox[i] + sbox[j]]`. This value is then applied to the current input byte with XOR and the result is appended to the output.

Specification of the RC4 algorithm is described inside an internet draft document by T. Kaukonen [14].

3.4.3 Hash functions

This is the core of authentication-based functions. Hash functions produce fixed-size fingerprints from a variable-length input. They are mainly used to verify the authenticity of some piece of information, e.g. a received message. Hash function is characterized by three main properties. It can be computed very fast for any input because it uses just cheap CPU operations. It is very difficult to calculate the original input text from a given hash, although many so-called rainbow tables exist these days, representing very large dictionaries with lists of plain text input to hash print mappings. The third advantage is

that only a slight modification of the input sequence results in drastic change to the fingerprint.

Two of the most commonly used hash functions, MD5 and SHA-1, are implemented in the enclosed implementation of this thesis.

Since both functions share a large portion of their source, we created a common procedure `create_hash` inside which the specific worker function is called implementing the core computation different for each of the algorithms.

`create_hash`

Hash function indicator `hash_func_t` is passed to a function to select MD5 or SHA-1 version of the hash. Procedure begins with converting the message into a special 64-byte aligned representation. In order to store information about the message's original length, 0x80 is appended to the message and zero-padding added for alignment. The message is then processed in 64B blocks by the worker procedure for the given hash function. When the computation is done, the resulting hash is copied to the output buffer.

`sha1_proc`

In the SHA-1 worker function, each block of scheduled input is processed according to the RFC 3174 specification [15].

`md5_proc`

MD5 uses a similar approach as SHA-1 but the computation is simplified with a shift amount array and a substitution box. Calculation process is taken from RFC 1321 [16].

3.4.4 IEEE 802.11-specific functions

This section describes functions whose functionality is either IEEE 802.11-specific or their implementation is restricted only for use within the IEEE 802.11 framework.

Keyed-hash message authentication code

This construction defined in RFC 2104 [17] is used for calculation of message authentication code (MAC) providing integrity and authentication of transferred data. It is based on MD5 a SHA-1 hash functions using terms HMAC-MD5 and HMAC-SHA1 accordingly.

Based on the given encryption key length, it either creates a hash using one of the specified functions if the key is longer than the block used during HMAC computation, or it uses the key directly. Two special sequences are created by applying XOR operation in iteration on every byte of the key and a special constant, resulting in an outer and an inner key pad. The resulting hash is then created using the following formula: `hash(o_key + hash(i_key + input))`, where the plus operator represents string concatenation.

Password-Based Key Derivation Function

Key derivation function PBKDF2 is used for deriving the pairwise master key (PMK) during a four-way handshake. Besides that it is used in disk encryption software and also in many systems and applications for protecting user passwords [18].

Resulting derivation is computed by parts that are called blocks. Every block is constructed by calling a specific function on the input password and salt sequence. This function involves specified iterations of chained pseudo-random functions (PRF). First PRF is applied on the input password and salt concatenated with the iteration counter. Then next PRF is applied to the password again, but the second parameter is the output from the last PRF and so on.

In our implementation we use HMAC-SHA1 as the PRF function.

IEEE 802.11 PRF

Pairwise key computation is used in the PRF defined in the IEEE 802.11 standard. This function takes the PMK as the encryption key and input text which is concatenation of the destination and source MAC addresses with generated random sequences from client and authenticator (ANonce, SNonce) as illustrated in following pseudocode:

```
concat_string = min(DA,SA) + max(DA,SA) + min(AN,SN) + max(AN,SN)
ptk = prf-x(pmk, 'Pairwise key expansion', concat_string)
```

The `x` in PRF denotes the length of the output which is 512b (64B) with TKIP and 384b (48B) with CCMP.

The 'Pairwise key expansion' string is then inserted before input data (concatenated sequence) and the created string is used in the processing loop. In every iteration the number of iteration counter is appended after the data and the resulting sequence is passed to HMAC-SHA1. The output is appended to the result, so when the procedure finishes, the result is composed from these concatenated blocks.

Key unwrapping

When GTK is received during a four-way handshake, it is encrypted inside the message and we have to use an unwrapping procedure for its derivation. Based on the used security protocol, either ARC4 or NIST AES unwrap is used.

Since encryption and decryption are the same operation for ARC4, we simply call RC4 using the KEK part of the PTK key that is concatenated after EAPOL-Key IV sequence from the handshake message.

In case of the CCMP protocol, an additional algorithm is used for key unwrapping called NIST AES [19]. Cipher text is interpreted as a sequence of 64bit values that are preprocessed with bitwise operations and then passed into the AES decryption procedure. Decrypted block is then appended to the result stream. When the process is done, we check that the initialization

sequence is present at the beginning of the decrypted data, otherwise the message is corrupted and should be dropped.

Michael MIC

When using the TKIP protocol, the MIC has to be computed always on the host side, contrary to CCMP. It uses a Michael MIC format which is the predecessor of the current MIC introduced with WPA2.

Given key is copied and separated into two parts `l, r` which are used to store the temporary result of the function. Frame header is processed separately from the payload. Only destination and source address fields are used, with 4 zero bits inserted before the data. This pseudo header is used only for MIC computation purposes. Michael procedure handles data in 32bit blocks using bitwise operations applied on `l, r` values to produce the resulting MIC. Last block is padded to a multiple of four with special 0x5A value and is processed separately at the end.

Final MIC value is then appended at the end of the data frame.

3.5 User application

A simple client application for managing wireless device is the part of the thesis and it is titled `wifi_supplicant`.

When the application is started, connections to `inetsrv` IP server and `dhcp` client process are initiated to be able to configure a device connection properly. User can list wireless devices, fetch scan results and manage connection to the selected network.

Scan results are displayed in a table containing network SSID, access point MAC address, network operating channel number and security related information (security suite, encryption algorithm used).

If an error occurs, a descriptive error message is shown based on the error code returned from the executed function.

As described before, the application calls functions from the RPC interface that are delegated further to the wireless driver.

User documentation for `wifi_supplicant` is at the end of the thesis in the appendix.

4. Evaluation

In this chapter we briefly summarize the functional and performance aspects of the resulting implementation of this thesis. Implemented features were tested on various wireless networks using different hardware devices.

Whole development was done and tested with one specific wireless USB adapter built on Atheros AR9271 chipset, specifically TP-LINK TL-WN722N which is currently available at almost every electronics store.

4.1 Functionality

In order to test the correct functionality of the resulting framework in a larger scope, we choose several representatives of WiFi access point devices and platforms we used during testing:

- VDSL modem Comtrend VR-3031eu
- Hotspot application on Android OS 4.4.4
- NetworkManager 1.0.2 utility on Linux OS

Prototype implementation of the IEEE 802.11 framework supports necessary processing of data and management frames that allows operation of the wireless device in station mode. It also offers key management and authentication mechanisms used to provide secure data transmission.

The following table is summarizing supported security standards and encryption protocols. WEP technology is not supported due to its deprecated status, TKIP+TKIP combination is implemented, but was not tested yet, because it wasn't available on tested devices.

Security suite	**Pairwise protocol**	**Group protocol**
WPA2 Personal	CCMP	CCMP
	CCMP	TKIP
WPA Personal	CCMP	CCMP
	CCMP	TKIP
	TKIP	TKIP
WPA2/WPA Enterprise	*Not supported*	
WEP	*Not supported*	
None	Supported	

Table 4.1: Implemented security protocols

4.2 Performance

Several tests were performed to compare the wireless stack performance with the existing wired one.

Results of latency tests are slightly worse as regards the wireless stack due to the nature of wireless transmission and the half-duplex principle of its operation. For this purpose, built-in `ping` utility was used. Values in table represent the average round-trip time from 10 iterations of sending echo message.

Throughput tests were done using the existing `download` tool in HelenOS with omitted screen output. Here the measured times are much slower than wired ones which could be caused by insufficient device calibration methods which are in other systems done adaptively, but in our implementation the calibration is set up only statically when switching device operating frequency. This can has a major impact on packet loss rate during data transmission.

Also when using `download` application, it quite often happens that the TCP connection is interrupted and the program halts. After its termination it is not operating correctly any more. When analysing the TCP transfer with Wireshark, it seems that sometimes there is a problem when some TCP segment to be received in HelenOS is lost that it is not accepted afterwards during its retransmission by the sender.

Wired	**Wireless with WPA2**
E1000 NIC	AR9271 IEEE 802.11 NIC
Latency tests Czech Republic server: 217.31.205.50 (nic.cz) := CZ United States server: 69.169.243.128 := US	
CZ — 32 ms US — 175 ms	CZ — 38 ms US — 184 ms
Throughput tests Small file of size 3 kB := SF Large file of size 575 kB := LF	
SF — 136 ms (22 kB/s) LF — 2 204 ms (261 kB/s)	SF — 267 ms (11 kB/s) LF — 11 715 ms (49 kB/s)

Table 4.2: Performance tests

5. Related work

In this chapter we present two main existing open-source implementations of wireless networking stack.

Since our implementation of wireless framework provides only basic functionality so far, we can't compare it to existing solutions thoroughly.

5.1 FreeBSD

FreeBSD is the free Unix-like operating system that is aiming for high performance and easy manageability by its end users. Also because of that, it has more users than the other BSD projects.

Whole IEEE 802.11 functionality is in FreeBSD implemented within the `net80211` module that provides operations required by wireless cards. This code is shared with NetBSD system where the wireless network stack was originated.

The stack is responsible mainly for media access control, cryptography related functionality, packet processing, network node management and converting packets between the 802.3 Ethernet and IEEE 802.11 format.

When new device is registered in the system, it has to call the `ieee80211_ifattach` initialization function. Device control state is stored inside the `ieee80211com` structure. Information there has to be filled by the device driver, such as a pointer to a connected network device, function callbacks and capability flags.

Another important entity in the framework are nodes. These represent the stations when operating in BSS mode. Their structure includes the unicast encryption key, current TX power or various statistics about the transmission history. All neighbour nodes that can be seen by the device are stored inside the `ieee80211com` structure in the `ic_sta` field. Functions for manipulation with nodes including scanning, maintaining the network structure etc. are implemented in `ieee80211_node.c` file.

Cryptography supports encryption and decryption of network frames. Key management is done through the `ioctl` interface inside the IEEE 802.11 layer which is the system call for manipulating with underlying device parameters.

The advantage of the `net80211` wireless stack implementation is that it is very well portable. There was also an attempt to port it on Linux platform within the MadWifi project, but it has ended up rejected and superseded by a brand new `mac80211` Linux stack.

Our prototype implementation of the IEEE 802.11 framework can be compared to the BSD one as it is sharing similar implementation aspects. More modular approach is used in the Linux implementation described in the next section.

5.2 Linux

Linux is the most popular alternative operating system. Its development is largely driven by user communities besides the commercial developers. Like FreeBSD, the source code of Linux are open-source, although published under more restrictive GPL license.

Linux distinguishes between SoftMAC and FullMAC wireless cards. SoftMAC is a type of a network interface that implements management operations like authentication, association etc. on the software side (e.g. AR9271), whereas FullMAC manage them in hardware.

In this case, wireless stack is designed in a more modular way. There exist two separated paths the user space processes communicate with the kernel. The data received from the wireless device are passed to the `netdev` networking core, then through the TCP/IP stack until they are queued on the relevant sockets in the user space where they are awaited by user process. Outgoing packets are sent from `netdev` to wireless device driver using the `ndo_start_xmit` callback. The driver has registered a structure with a set of operation callbacks like other network devices in the system. Operations like scanning and association are managed through the control path. The user space interface is based on the `nl80211` module. We can send control commands through there and register events for responses.

Messages sent via `nl80211` are in kernel handled in the `cfg80211` module. In case of a FullMAC device, the related hardware driver is right below the `cfg80211` layer and it registers a set of callbacks provided in the `cfg80211_ops` structure. For SoftMAC devices there is a `mac80211` MAC layer which is communicating with the `cfg80211` directly instead of the device itself. `mac80211` provides `ieee80211_ops` operations structure for device drivers while implementing `cfg80211_ops` itself.

Initialization of a new device is first managed through the `mac80211` layer by calling its initialization functions and then the configuration is delegated to `cfg80211` layer. Device driver structure is separated into several layers corresponding to wireless stack layers. These are visible only in respective levels and are hidden for other layers.

We can set up more virtual interfaces based on single physical device. These can be seen under different names using the `ifconfig` configuration tool (e.g. wlan0, wlan1, ...). Each of these virtual interfaces is represented by `ieee80211_vif` object containing also structures to be accessed by device driver.

Managing authentication process together with eventual software packet encryption is handled separately in external `wpa_supplicant` application, as was said earlier.

Conclusion

The main goal of the thesis was to implement a native HelenOS driver for an IEEE 802.11-compliant device. This task was accomplished and the result is a driver for Atheros AR9271, a popular WiFi chipset. During the development of the driver, a wireless framework providing generic IEEE 802.11 functionality was created and can be reused by drivers for other devices. This was the second main goal of this work.

As a by-product, a library with cryptographic algorithms was also implemented to provide the needed functionality for secure data transmission within a wireless network. It makes it possible to communicate using the currently most widespread wireless LAN security standard WPA2. Many of the implemented functions are not specific to IEEE 802.11 and can also be used by other applications to provide data integrity or other forms of authentication.

The enclosed implementation is also accompanied by a simple utility for managing connection and authentication between a wireless device in station mode and a WiFi network access point.

As shown in the *Evaluation* chapter, the implemented solution is functional, but it will need to be optimized to provide the same levels of performance as existing implementations in conventional operating systems. On the other hand, the presented solution is more robust regarding the system security, because In Linux or Windows a buggy device driver can crash the whole system. This is not a case of the HelenOS system thanks to its microkernel design providing the application isolation by its nature.

Future work

The current implementation could be improved and extended in many ways.

At the moment, the resulting driver allows the device to operate only in station mode. Although it is the most used form of a WiFi device operation in desktop operating systems, it would also be attractive to be able to run the device in hotspot mode and host the wireless network. This would involve implementing support for processing more types of frames than in the current implementation and also managing and keeping track of the connected stations and forwarding their messages to their respective recipients in the network.

Another feature that the current implementation lacks is any form of power management. To avoid keeping the device awake for the whole time the HelenOS is running, we should put the device to sleep mode when there are no data ready for it and keep it awake only if there are more data pending in the sender's queue. This could be detected from the frame header.

If it becomes necessary for the IEEE 802.11 framework to handle multiple tasks simultaneously, it would be better to process incoming and outgoing packets asynchronously to make the system more scalable. As said before, it would also be better to extract the authentication process from the IEEE 802.11 framework for a more modular approach.

To add support for devices that are not able to manage encrypted packets themselves, software encryption and decryption must be implemented. This implies programming several new cryptographic functions that are not yet included in the current solution.

As described in *Evaluation* chapter, it would also be useful to improve the device calibration procedures to operate adaptively based on recent signal and noise floor values.

Bibliography

[1] IEEE 802.11i-2004: Amendment 6: Medium Access Control (MAC) Security Enhancements, on-line at http://standards.ieee.org/getieee802/download/802.11i-2004.pdf

[2] Wikipedia: Comparsion of open-source wireless drivers, on-line at https://en.wikipedia.org/wiki/Comparison_of_open-source_wireless_drivers

[3] Linux kernel documentation: Building external modules (section 2), on-line at https://www.kernel.org/doc/Documentation/kbuild/modules.txt

[4] USB subsystem in HelenOS: Bypassing the framework (page 63), on-line at http://helenos-usb.sourceforge.net/helenos-usb-doc.pdf

[5] L. Trochtová: Device drivers interface in HelenOS system (in Czech, page 54), on-line at http://www.helenos.org/doc/theses/lt-thesis.pdf

[6] RFC 2131: Dynamic Host Configuration Protocol (page 22), on-line at https://www.ietf.org/rfc/rfc2131.txt

[7] Atheros AR9271 product datasheet, on-line at http://www.cqham.ru/forum/attachment.php?attachmentid=155133&d=1383397504

[8] Ath9k wireless devices family Linux kernel 4.0 source code, on-line at http://lxr.free-electrons.com/source/drivers/net/wireless/ath/ath9k/?v=4.0

[9] The firmware for QCA AR7010/AR9271 802.11n USB NICs, on-line at https://github.com/qca/open-ath9k-htc-firmware

[10] RFC 1042: A Standard for the Transmission of IP Datagrams over IEEE 802 Networks, on-line at https://www.ietf.org/rfc/rfc1042.txt

[11] A closer look at WiFi Security IE, on-line at http://community.arubanetworks.com/t5/Technology-Blog/A-closer-look-at-WiFi-Security-IE-Information-Elements/ba-p/198867

[12] HostAP mailing list: Difference between WPA1-PSK CCMP and WPA2-PSK CCMP, on-line at http://lists.shmoo.com/pipermail/hostap/2006-August/014137.html

[13] FIPS PUB 197: Advanced Encryption Standard (AES), on-line at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[14] T. Kaukonen: A Stream Cipher Encryption Algorithm "Arcfour", on-line at https://tools.ietf.org/id/draft-kaukonen-cipher-arcfour-03.txt

[15] RFC 3174: US Secure Hash Algorithm 1 (SHA1), on-line at https://tools.ietf.org/rfc/rfc3174.txt

[16] RFC 1321: The MD5 Message-Digest Algorithm, on-line at http://tools.ietf.org/rfc/rfc1321.txt

[17] RFC 2104: HMAC: Keyed-Hashing for Message Authentication, on-line at https://tools.ietf.org/rfc/rfc2104.txt

[18] RFC 2898: Password-Based Cryptography, on-line at https://www.ietf.org/rfc/rfc2898.txt

[19] RFC 3394: Advanced Encryption Standard (AES) Key Wrap Algorithm, on-line at https://www.ietf.org/rfc/rfc3394.txt

[20] IEEE 802.11-2012: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, on-line at http://standards.ieee.org/getieee802/download/802.11-2012.pdf

[21] J. Berg: mac80211 overview, on-line at https://wireless.wiki.kernel.org/_media/en/developers/documentation/mac80211.pdf

[22] L. Mejdrech: Networking and TCP/IP stack for HelenOS system, on-line at http://www.helenos.org/doc/theses/lm-thesis.pdf

[23] HelenOS NICF documentation, on-line at http://www.helenos.org/doc/helnet.pdf

[24] Networking stack Reference Manual, on-line at http://www.helenos.org/doc/refman/networking-0.4.2

[25] J. Corbet, A. Rubini and G. Kroah-Hartman: Linux Device Drivers, Third Edition. O'Reilly Media Inc., 2005. ISBN 978-0596005900.

List of Abbreviations

In this section are described the abbreviations connected with network terms used in the thesis.

AES Advanced Encryption Standard, standardized block cipher encryption algorithm.

AID Association Identifier, unique value which allows the access point in infrastructure WiFi network to keep track of client stations.

AP Access Point, the device to which clients in infrastructure WiFi network are connecting. It is a central point for managing communication between clients.

BSS Basic Service Set, basic unit of infrastructure WiFi network consisting of two or more computers communicating between each other.

CCMP Counter mode Cipher block chaining Message authentication code Protocol, encryption protocol designed for WLAN providing data confidentiality, authentication and access control. It is based on AES standard.

CSMA/CA Carrier Sense Multiple Access with Collision Avoidance, network multiple access method using carrier sensing.

DCF Distributed Coordination Function, type of media access control distributed mechanism used in WLAN. It is using CSMA/CA algorithm for collision avoidance on shared transmission channel.

DHCP Dynamic Host Configuration Protocol, standardized network protocol used for assigning the IP addresses to network devices.

DS Distribution System, method for interconnection between multiple AP in WLAN.

EAPoL Extensible Authentication Protocol over LAN, simple encapsulation of EAP that can run over LAN. EAP is an authentication framework used in wireless networks.

ESS Extended Services Set, component of the WLAN architecture composed from two or more BSS.

GTK Group Temporal Key, encryption key used in multicast communications by CCMP and TKIP protocols.

HIF Host Interface, interconnect driver interface used in Atheros wireless drivers.

HTC Host/Target Communication, transport layer used in Atheros devices responsible for communication between the computer and device firmware.

IEEE Institute of Electrical and Electronics Engineers, the world's largest professional association for the advancement of technology.

ISO/OSI International Organization for Standardization / Open System Interconnection, network reference model for how applications can communicate over a network. It consists of seven layers - physical, data link, network, transport, session, presentation and application.

ISP Internet Service Provider, an organization that offers Internet services to the end users.

LAN Local Area Network, computer network covering small geographical areas, such as home, schools or office buildings.

MAC Media Access Control, sublayer of data link layer in ISO/OSI model providing addressing and shared channel access methods.

MIC Message Integrity Code, value generated from cryptographic algorithm used to protect data from unauthorized alternation.

MIMO Multiple-Input and Multiple-Output, abstract mathematical model for multiple antennas communication systems, recently used in radio communication area for significant increase of data throughput and network coverage.

PCF Point Coordination Function, centralized technique for media access control used in WLAN.

PSK Pre-Shared Key, shared secret between the AP and client in infrastructure WiFi network used to construct keys for encrypted data transmission.

PTK Pairwise Transient Key, encryption key used in unicast communications by CCMP and TKIP protocols.

QoS Quality of Service, level of the overall performance of computer network as seen by the users.

RC4 Rivest Cipher 4, stream cipher.

RFC Request For Comments, series of documents describing internet protocols and systems. They are rather recommendations than standards, but large majority of the Internet is following them.

RTS/CTS Request To Send / Clear To Send, flow control mechanism in IEEE 802.11 wireless networking used to avoid hidden and exposed node frame collision problems.

SSID Service Set Identification, unique identifier for a WLAN.

TCP Transmission Control Protocol, main protocol of transport layer in TCP/IP protocol family that guarantees reliable communication and delivering messages in correct order.

TKIP Temporal Key Exchange Protocol, encryption protocol defined in IEEE 802.11i amendment as an enhancement to WEP.

WEP Wired Equivalent Privacy, deprecated security algorithm for IEEE 802.11 wireless networks.

WiFi A popular label for certain types of WLAN networks based on IEEE 802.11 specifications.

WLAN Wireless Local Area Network, interconnection method between multiple computers using microwave radio signals.

WMI Wireless Module Interface, wireless protocol for communication used in Atheros devices.

WPA Wireless Protected Access, the WiFi Alliance certification of IEEE 802.11 amendment. Original WPA version uses TKIP encryption, WPA2 uses CCMP/AES.

Appendices

A. User application documentation

The work is supplied with a simple client application called `wifi_supplicant` for managing connections to wireless networks.

Available parameters

Here are described all the options that can be used when running the application.

```
Usage: wifi_supplicant [<cmd> [<args...>]]

Available <cmd> commands:

list
```

List available WiFi devices in *index: name* format.

```
scan <i> [-n]
```

Output scan results for the device at index `i`. `-n` parameter is to force initiation of scan procedure immediately. Example of results output is shown further in Figure A.1.

```
connect <i> <ssid> [<password>]
```

Connect the device with tag `i` to network which SSID starts with `ssid` prefix. If encrypted communication is used, secret passphrase should be supplied in the `password` field.

```
disconnect <i>
```

Disconnect the device `i` from currently connected network.

Example usage

First we need to find out what is the index of wireless device we want to use.

```
wifi_supplicant list
```

This will output indexes of wireless devices together with their service name (logical path in system). If there is just single WiFi device in the system, it has index value `0`. Next step is to list available WiFi networks in neighbourhood.

```
wifi_supplicant scan 0
```

Now we can choose the desired WiFi network from list and connect to it.

```
wifi_supplicant connect 0 my_network my_password
```

If everything went OK, the program should print positive notification on the screen output. Otherwise some error occurred. When program states that the device is not ready yet, it means that the initialization of the wireless

device has not finished yet. This could be detected for example by checking device log file. Sometimes it may happen that the timeout occurred when authenticating to network. This can signalize a packet loss during the process, we can try connecting again.

After the device has successfully connected to a network, the application composes a request for IP address assignment which passes to HelenOS DHCP client. Currently the success of this procedure is not verified. To check that this process was completed successfully, we need to check that the DHCP route connected with this wireless device is created by running `inet` command. In output of this command there should be configured address with `dhcp4a` value in `Addr-Name` field.

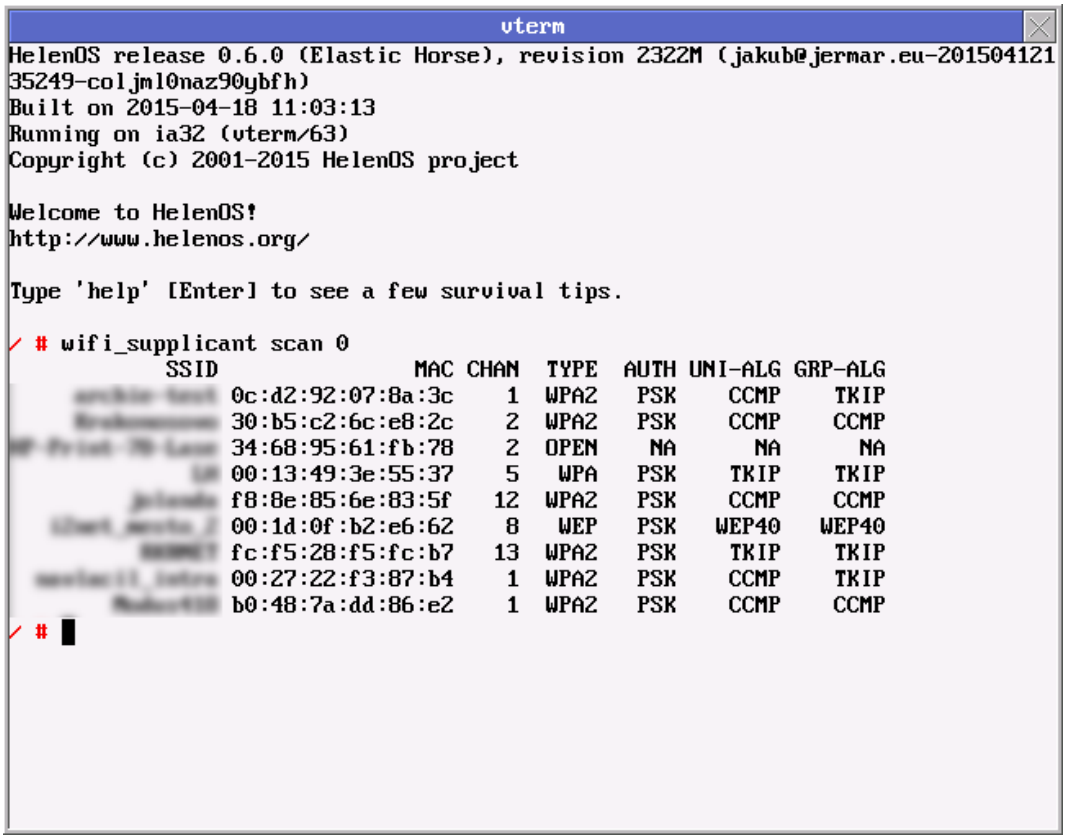


Figure A.1: Example of scan results output

B. Example Python script

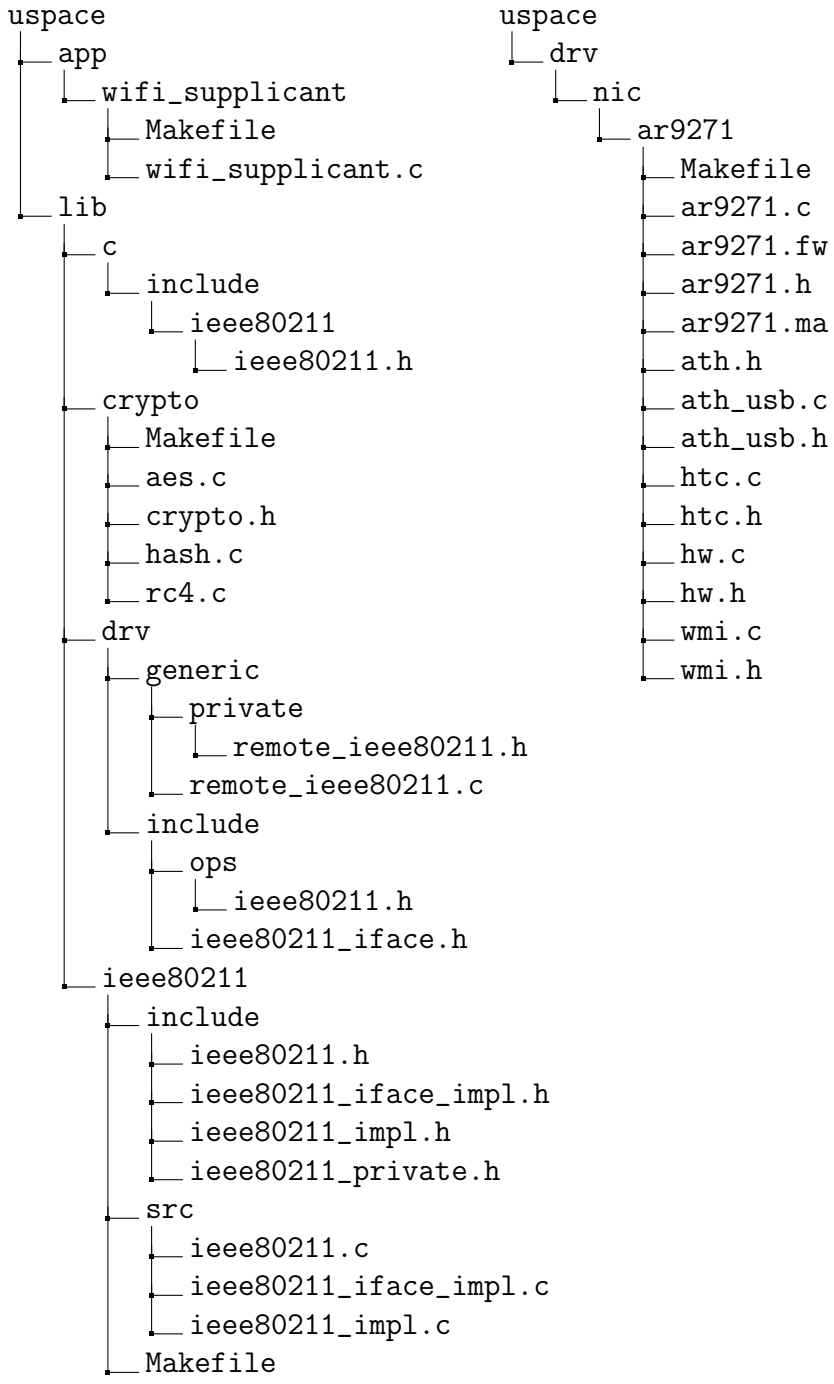
```
import hashlib
import hmac
import struct

from Crypto.SelfTest.st_common import a2b_hex, b2a_hex
from Crypto.Util.py3compat import tobytes, b
from Crypto.Hash import SHA as SHA1, HMAC
from Crypto.Util.strxor import strxor

# IEEE 802.11-specific pseudorandom function.
def custom_prf(key, a, b, l):
    i = 0
    r = ''
    while i < ((l + 159) / 160):
        hmacsha1 = hmac.new(key, a + chr(0x00) + b + chr(i), hashlib.sha1)
        r += hmacsha1.digest()
        i += 1
    return r[:l/8]

# PBKDF2 key derivation function
def pbkdf2(password, salt, key_len=32, count=4096):
    password = tobytes(password)
    prf = lambda p, s: HMAC.new(p, s, SHA1).digest()
    key = b('')
    i = 1
    while len(key) < key_len:
        salt_str = salt + struct.pack(">I", i)
        u = prev_u = prf(password, salt_str)
        for j in xrange(count-1):
            prev_u = t = prf(password, prev_u)
            u = strxor(u, t)
        key += u
        i += 1
    return key[:key_len]

# Values used for PTK computation
pass_phrase = "test_phrase"
ssid = "archie-net"
ap_mac = a2b_hex("14f65aaf5eb7")
client_mac = a2b_hex("c04a00294507")
a_nonce = a2b_hex("05994a54bfbe2064c8dbccef24c434c862b599dff4b99ba1378a3314"
                  "f422bd68")
s_nonce = a2b_hex("bff89495ee27154b24ec5b1a9ae1bd16ee385366b292cd77b4cc9c1b"
                  "08b7f2ef")

A = "Pairwise key expansion"
B = min(ap_mac, client_mac) + max(ap_mac, client_mac) + \
    min(a_nonce, s_nonce) + max(a_nonce, s_nonce)

raw_data_stream = "0103007502010a0000000000000000000001bff89495ee27154b24ec" \
                  "5b1a9ae1bd16ee385366b292cd77b4cc9c1b08b7f2ef0000000000" \
                  "000000000000000000000000000000000000000000000000000000" \
                  "000000000000000000000000000000001630140100000fac0401" \
                  "00000fac040100000fac020c00"
```

```
data = a2b_hex(raw_data_stream)

pmk = pbkdf2(pass_phrase, ssid)
ptk = custom_prf(pmk, A, B, 384)

print "PMK: " + b2a_hex(pmk)
print "PTK: " + b2a_hex(ptk)
print "MIC: " + hmac.new(ptk[:16], data, hashlib.sha1).hexdigest()[:32]
print "TK: " + b2a_hex(ptk[-16:])
```

Listing 8: Verification of pairwise key derivation process

C. Source tree structure

Only newly created files are presented in the following diagram.



D. AR9271 initialization diagram