WHITEPAPER (/CATEGORY/WHITEPAPER)

Emulating WLAN in Linux - part I: the 802.11 stack

Par Geoffrey Le Gourrierec (http://linuxembedded.fr/liste-articles-auteur/152) le 28 mai 2020 (/2020/05/emulating-wlan-in-linux-part-i-the-80211-stack) • pas de commentaire (/2020/05/emulating-wlan-in-linux-part-i-the-80211-stack#comments)

WLAN networks are a hassle to set up, even more than "physical" cables and RJ45 plugs. While wireless communication is a commodity for the end user, the engineer, in charge of **developing** and **testing** it at software level, can be quickly annoyed. This article is written from the point of view of someone having to develop Wi-Fi related code, e.g. traffic analysis tools, or tweaking the network stack.

Indeed, as there is no shortage of Ethernet-based networking tools, including in virtualized environments, 802.11 (the protocol family behind "Wi-Fi") does not enjoy similarly broad support. There are many reasons behind this.

- 802.11 frame formats and semantics are much more complex than their straightforward Ethernet counterparts; so is the code needed to handle them *(hint: code is needed to handle them, hardware is not enough)*.
- Ethernet predates Wi-Fi by nearly two decades, making it more pervasive. It is the favored technology when setting up network support for any virtual or physical platform.
- While the 802.11 protocol family is a set of IEEE open standards, just like Ethernet (802.3), the industry-backed Wi-Fi consortium did not push forward open implementations in the various Wi-Fi hotspots and gateways products seen everywhere nowadays. You can be pretty sugsdfgsdre that a Cisco router is using a very different implementation from a TP-Link one. Low-level software tooling, then, is quite often proprietary.

As a consequence, setting up a test bench, with more than a couple of devices (at the very least, one client and one access point), requires handling following problems:

- **Setting up network configurations** on each device, probably **with distinctive tools** if they're different brands (as it is most likely in a R&D environment).
- **Making sure the stations operate in a "clean" space** in wireless terms, i.e. not having unwanted traffic on used channel(s).
- **Depending on proprietary software stacks** with different, undocumented behaviours where the standards give leeway.
- **Depending on proprietary hardware stacks** with different, undocumented behaviours because firmware blobs are as much a reality as in the graphic cards or Bluetooth industry.

But we should not despair! As Linux-based platforms become more prominent, open 802.11 stacks for client, or even station devices, are gaining more weight. Today, we shall look into a prominent **testing device** made possible by this openness: **the *mac80211_hwsim* Linux kernel module, which offers support for emulated WLAN adapters.**

What is mac80211_hwsim about ?

As said before, we're looking at a Linux kernel module. Its goal is simple: emulating WLAN adapters. After a quick presentation on how 802.11 is handled in Linux, we'll practice using *mac80211_hwsim* with a few common tools. Finally, we'll dive into the code, and understand how this all works.

Thinking about the previously mentioned issues we faced, an open and virtual WLAN adapter offers the following solutions:

- **Unified tooling** for device setup and configuration. Plus, time saving: no need to physically move stuff around.
- Wireless traffic is entirely emulated too, which means **complete control** over it.
- The Linux 802.11 stack is open-source, of course. As we will see later, virtual devices emulated by *mac80211_hwsim* are completely orthogonal to the network stack, which makes them **useable just as any "real" adapter**.
- **No hidden firmware** to worry about: behaviour is predictable, which gives us repeatable network tests; a great boon to whoever wants to achieve functional testing.

Part I: 802.11 in Linux

In this first part, we will have a quick introduction to how WLAN devices are used and represented in Linux. A second part will delve into specificities of the mac80211_hwsim module.

WLAN devices

Network devices as seen from the kernel

Linux network devices and drivers are very different from their "character" or "block" cousins. Their common language is the `struct sk_buff` declared in *linux/skbuff.h*, which allows manipulation of network packets, and the `struct net_device` , which abstracts the underlying transport medium. However, devices do not show themselves in the *devfs*, and drivers do not implement specific APIs; each protocol does its own stuff, at the layer it's supposed to be in. At the end of the road, each one is just manipulating buffers.

By working on 802.11 devices, we are at the very bottom of this layered model. Linux maps WLAN interfaces to the generic interface object one can see using `ip link` . Commands such as `ip` "abstract" away the physical medium used: Ethernet, TAP, WLAN...because `ip` works at a higher network layer. On my computer, here we see the loopback device, an Ethernet card and a WLAN interface (an Intel Centrino 8260 [4]), all very different in nature but abstracted under a common representation:

```
$> ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s31f6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT group default
 qlen 1000
    link/ether 18:db:f2:55:0c:34 brd ff:ff:ff:ff:ff:ff
3: wlp1s0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 34:f3:9a:eb:4a:38 brd ff:ff:ff:ff:ff:ff
```

WLAN physical adapters map to the `struct wiphy` object, described in *net/cfg80211.h* (we'll come back to this header later). wlp1s0 is a "virtual" interface on top of it, and is represented by `struct ieee80211_vif` in the kernel. Several virtual interfaces just like wlp1s0 can co-exist on a single `wiphy`; and each can behave with a certain *mode*.

WLAN modes

Modes influence **how the WLAN interface interacts with the network.** Availables modes **completely depend on the hardware**: some cards support some modes that others do not. One should be careful about the modes that a card offers, before buying it for a project !

The mode is also a software knob, and can be changed if the interface is not currently up. In this manner, what the software see is only a direct mapping of what the hardware offers. No mode is implemented in software.

Modes are not defined by the 802.11 standard; there is, however, a consensus on possible modes:

- **Managed**: the most common mode; a client connects to an access point to exchange traffic. That's the one you're using when you're surfing the Web. Also called "infrastructure" mode.
- **Access Point**: master of a local networking service. Able to relay frames between stations, identified by their MAC addresses. One can add an IP address, a DNS relay and DHCP server, e.g. `dnsmasq` in Linux, to bridge the gap to a TCP/IP network like the Internet.
- **Mesh**: this peculiar mode, sometimes called IBSS mode, allows to join mesh-like networks. Let us recall 802.11 was originally conceived with star-like networks in mind, with the access point at the center. This possibility is nowadays rarely explored. However, the ongoing Vehicular Communications Systems efforts may lead to renewed interest in this mode.
- **Monitor**: solely used for "manual" traffic inspection; allows maximum control by seeing every frame involved, not only handshakes and data frames. This is what any software directly working on 802.11 level wants, and what the usual Linux "raw sockets" API allows to see.

Several other modes exist, but those were the main ones for the purpose of this article. We find the same names across different OS.

"Why not having a single, do-everything mode," you may ask ? The answer is in two pieces:

1. The **hardware cannot perform all modes** simultaneously.
2. **Current OS designs** need to know how **userspace code wants to use the 802.11 stack.** Managed and access point modes order your kernel to enable certain codepaths responding to clearly defined roles. Again, acknowledging the complexity of the protocol sheds some light on this: data frames are only one of several types of possible frames, all playing some role in collision avoidance, client authentication, and more. I count about 25 different types in my copy of the (2016-revised) base 802.11 standard [3], without worrying about encryption methods (WEP, WPA, WPA-2…) or cipher suites (TKIP, CCMP…).

A mindful note for the curious: the monitor mode is essential to traffic analysis and understanding, but its availability depends on several factors (yet another reason to want an emulated adapter :)). As explained in the Wireshark docs:

> **Unfortunately, changing the 802.11 capture modes is very platform/network adapter/driver/libpcap dependent, and might not be possible at all (Windows is very limited here).**
>
> https://wiki.wireshark.org/CaptureSetup/WLAN (https://wiki.wireshark.org/CaptureSetup/WLAN)

Manipulating WLAN devices: basic tools

Just as `ip` acts on interfaces at the IP level, we use `iw` for WLAN ones. This handy command-line tool allows us to list, inspect, and configure wiphys and their associated virtual interfaces. Here is its output for me:

```
# Note: you may need to have elevated privileges to use iw
# Some distributions, like Debian, put it in /sbin/
$> iw dev
phy#0
        Interface wlp1s0
                ifindex 3
                wdev 0x1
                addr 34:f3:9a:eb:4a:38
                type managed
```

We can see I have a single wiphy, with one interface on top: `wlp1s0`. It is also in managed mode, and is not currently active. Notice the generic `ifindex` that Linux uses for pretty much all network interfaces, needed whenever interacting through the socket API.

I can switch to monitor mode on channel 11 (2,4Ghz band) like this:

```
#> ip link set dev wlp1s0 down
#> iw dev wlp1s0 set type monitor
# If your WLAN adapter does not fully support monitor mode, which is
# pretty common for embedded cards in laptops, this may not work.
#> iw dev wlp1s0 set channel 11
#> ip link set dev wlp1s0 up
```

You might have noticed `wlp1sO` is put under `phy#0`. This is the underlying wiphy device; typing `iw phy phy0 info` (omit the '#') tells me a lot of information on what this card can actually do. (Output cut for clarity)

```
Wiphy phy0
        Supported interface modes:
                 * IBSS
                 * managed
                 * AP
                 * AP/VLAN
                 * monitor
                 * P2P-client
                 * P2P-GO
                 * P2P-device
        Band 1:
                Frequencies:
                        * 2412 MHz [1] (22.0 dBm)
                        * 2417 MHz [2] (22.0 dBm)
                        * 2422 MHz [3] (22.0 dBm)
                        * 2427 MHz [4] (22.0 dBm)
                        * 2432 MHz [5] (22.0 dBm)
                        * 2437 MHz [6] (22.0 dBm)
                        * 2442 MHz [7] (22.0 dBm)
                        * 2447 MHz [8] (22.0 dBm)
                        * 2452 MHz [9] (22.0 dBm)
                        * 2457 MHz [10] (22.0 dBm)
                        * 2462 MHz [11] (22.0 dBm)
                        * 2467 MHz [12] (22.0 dBm)
                        * 2472 MHz [13] (22.0 dBm)
                        * 2484 MHz [14] (22.0 dBm)
        Supported commands:
                 * new_interface
                 * set_interface
                 * new_key
                 * start_ap
                 * new_station
                 * new_mpath
                 * set_mesh_config
                 * set_bss
                 * authenticate
                 * set_channel
                 # Many more commands...
```

For instance, we observe the complete list of WLAN "modes" available; and there's quite a few !

We can also see my WLAN adapter can listen to all possible 14 channels defined for the 2,4GHz band, even though some might not be useable depending on the country I'm currently in (if you're wondering why, may I suggest taking a look at [2]).

Finally, I included part of the "Supported commands" paragraph, to introduce how the kernel "talks" to these interfaces. All these commands are linked to the nl80211 module, which forms the "upper" part of the 802.11 stack, and exposes an actual API for devices. As you might guess, this is also what a tool like `iw` uses to grab all its data. When I typed `iw dev wlp1s0 set channel 11` , I really invoked the `set_channel` command with the appropriate frequency.

Manipulating WLAN devices: high-level tools

Of course, nobody wants to connect to her home Wi-Fi by launching a few commands in a shell. Network managers, equipped with GUI, are there to fill the need for user-friendly tooling. Those vary with the distro, as all tools do in the Linux world, but we can mainly cite `conman` and `NetworkManager` .

As *backends*, however, there are terribly few tools: `wpa_supplicant` if you want to connect, and `hostapd` if you want to be an access point. The reason ? Complexity again. Linux does not actually try to implement everything in kernel, because it would be too big, too difficult to maintain, and most likely out of date at the moment it will be shipped.
Linux offers crypto and keys storage subsystems, so that companies wishing to implement complex corporate networks (Wikipedia counts more than a dozen EAP methods alone [5]) can ship customized versions of `wpa_supplicant` (and perhaps `hostapd`) precisely for their use case. In fact, that is the way those tools evolve in Linux, as Marcel Holtmann describes very well in his 2018 talk at ELCE [6], which both enumerates issues with Linux tooling, and why he is hacking away new ones with the help of Intel colleagues.

The 802.11 stack

The Linux 802.11 stack is split into three parts, each one implemented with a different module in the kernel. We shall study what are the responsibilities of each of them. Device drivers follow, of course, and tie the knots by implementing whatever APIs the base stack imposes.

Let us not forget that hardware offloading is common, and this trend is even rising. Offloading is supported for a number of operations, even though we will not touch this subject; after all, we're about to see a full software device.

The stack from afar

Let us keep in mind the different aspects of WLAN communication we have to juggle with. The three parts of the stack tackle those problems.

- First, **sending and receiving data**; the end need. This is the job of **"data frames"**.
- 802.11 is a protocol that offers encrypting data frames; for this, an **authentication mecanism** framework is built into the standard, and special frames named **"management frames"** are dedicated to this job. From the moment a client is searching for an access point to the moment it actually starts sending data to one, this preliminary work has to be done.
- As for all wireless mediums, 802.11 has to offer sufficient **collision handling** so that communications on the same channel are not complete chaos; this is the responsibility of **"control frames"**. In this case, 802.11 uses *collision avoidance* with time windows for "talking".
- **Reconfiguring our adapter** at runtime, the simplest example being changing channel.
- Finally, the userland should know about **WLAN events**, like losing connection to the BSS because the access point is too far.

Overview of the 802.11 stack in Linux
Now here are the **three 802.11 stack modules**, and what they do:

- **mac80211**: This is the lower layer, and the most associated to hardware offloading. In fact, by definition, **whatever code needed here is not provided by the hardware, or is preferred in software** to give better control for developers. In short, the 802.11 protocol state machine lives here, for *every* type of frame mentioned earlier. Also called the "Soft MAC" module, as opposed to "Hard MAC" (i.e. the device's firmware does it all). Not surprisingly, reality is not black and white: a compromise is often used.
- **cfg80211**: This middle-layer **handles everything configurable** for your WLAN adapters; contrary to mac80211, it is mandatory for drivers to interface with it. The "current channel" earlier example is maintained here. The regulatory domain is also implemented here, which is a good example on how this module really acts as a bridge between the kernel and userspace. However, it is not cfg80211 that defines the command set seen earlier through `iw phy`; this is where the last module comes in.
- **nl80211**: The **API between user-land and kernel-land**. Relies on the *netlink* protocol to exchange messages between the two worlds; basically a front-end for cfg80211. This will also allow to generate appropriate "events": messages that consumers will have to listen to.
  Netlink is a Linux-specific socket type used for events needing to cross the kernel-user border. A well-known example of its use is the Udev daemon, which is the program that manages device hotplug and relays hardware events; for instance, telling your file explorer that an USB key has just been inserted.
  As nl80211 constitutes "just" an interface, it is mostly about packaging data into appropriate formats and tunneling them down or up. We're not going to discuss it much further.

mac80211

Let's start with the bottom: mac80211. As mentionned just before, the WLAN driver writer that wants to handle some logic in software can use this module's interface; we find it in *net/mac80211.h*.

The header file is huge, but we do not need to look at it all; 802.11 is a *family of standards,* with a base and a multitude of extensions. For example, the *quality of service* is an extension (802.11e) that aims at prioritizing some data over data, using "tags" that describe its nature and thus, its latency sensitivity (voice and video come to mind as demanding). I call to your attention on a number of structures seen here, that we will find central to the implementation of *mac80211_hwsim*.

- `struct ieee80211_vif` is the "virtual" interface configuration, one per interface, that we mentionned earlier. As expected, it is passed as a parameter to a whole lot of functions. Here is a simplified version, that goes to the point.

```
struct ieee80211_vif {
        // "Type" == "Mode": monitor, managed...
        enum nl80211_iftype type;

        // The dynamic configuration of the access point we're
        // currently connected to: authentication status, association
        // status, capabilities, beacon interval...
        struct ieee80211_bss_conf bss_conf;

        // Channel context: what central frequency am I on ? What is
        // the channel's width ? WLAN allows several ones, from 20MHz
        // to 160 MHz...
        struct ieee80211_chanctx_conf __rcu *chanctx_conf;

        // A debugfs entry, if fancied by the driver author
#ifdef CONFIG_MAC80211_DEBUGFS
        struct dentry *debugfs_dir;
#endif
};
```

- `struct ieee80211_sta` is a station we are talking to; it evokes the `struct ieee80211_bss_conf` (storing the configuration about some other node), except it's about a regular "client" node, not an access point. Recall we have to negotiate systematically "how" to talk with any node: what bitrate to use, with which capabilities; if it supports sending A-MSDU frames (akin to "Jumbo" frames in the Ethernet world); etc.
- `struct ieee80211_hw` is the low-level representation of a physical device. It is, in fact, linked to a `struct wiphy` device, and both complete each other. We're more interested in practical HW limitations here, like bitrate, number of transmit queues, etc. Again, most of the fields have been omitted for clarity.

```
struct ieee80211_hw {
        struct wiphy *wiphy;

        unsigned long flags[BITS_TO_LONGS(NUM_IEEE80211_HW_FLAGS)];

        // Number of hardware transmit queues
        u16 queues;

        // The number of rates we can try out when negotiating with
        // a peer station, and how many retries we tolerate
        u8 max_rates;
        u8 max_rate_tries;

        // On ciphering data frames: TKIP, CCMP...recall other Linux
        // subsytems will do the bulk of the job, however
        u8 n_cipher_schemes;
        const struct ieee80211_cipher_scheme *cipher_schemes;

        // Maximum Transfer Unit
        u32 max_mtu;
};
```

- Most importantly, the `struct ieee80211_ops` defines the actual API any soft MAC driver can implement here, at the *mac80211* level. If you ever wrote down a simple character mode Linux device driver, this is the same logic as the `struct file_operations` object. It is quite enormous; in Linux 5.6, I listed 102 functions. Of course, here we take note of the most basic ones, just to get a feel for them:

```
struct ieee80211_ops {
        void (*tx)(struct ieee80211_hw *hw,
                   struct ieee80211_tx_control *control,
                   struct sk_buff *skb);

        int (*start)(struct ieee80211_hw *hw);
        void (*stop)(struct ieee80211_hw *hw);

        int (*add_interface)(struct ieee80211_hw *hw,
                             struct ieee80211_vif *vif);
        int (*change_interface)(struct ieee80211_hw *hw,
                                struct ieee80211_vif *vif,
                                enum nl80211_iftype new_type, bool p2p);
        void (*remove_interface)(struct ieee80211_hw *hw,
                                 struct ieee80211_vif *vif);

        int (*config)(struct ieee80211_hw *hw, u32 changed);
        void (*bss_info_changed)(struct ieee80211_hw *hw,
                                 struct ieee80211_vif *vif,
                                 struct ieee80211_bss_conf *info,
                                 u32 changed);

        // Many, many more...
};
```

One question might still remain: why would anyone want to do work in mac80211, if it can be done in hardware ? Let us be more explicit about the advantages:

- **Written-once, quality code** maintainted in Linux
- An interface reworked over time to **fit everyone's usecases**
- **Good debugging support** through sysfs, debugs, possibly ftrace
- Support in hardware does not necessarily mean faster code; the bottleneck remains wireless issues (i.e. collisions and tx/rx problems) and antenna access

cfg80211

We arrive at the bridge between the device and userspace. If the job of *mac80211* is to allow listing all capabilities of the hardware and interacting with it, *cfg80211* will keep track of the actual state your WLAN adapter is currently in.

But there's more: some concepts are made more user-friendly here, like the `struct ieee80211_supported_band` that could allow one to define in one object the whole range of 2,4GHz channels available with his device. This looks easier than reasoning purely with frequencies and channel width.
It is also here that regulatory domains are defined, a concept that is absolutely not part of the 802.11

standard but imposed by telecommunications rules across the globe.
Finally, note that it is this layer that really makes a distinction between concurrent *virtual interfaces*, as will be illustrated by some of the important objects below.

- Remember `struct wiphy` ? Well, it is actually defined here ! This "high-level" companion to `struct ieee80211_hw` is used almost everywhere in the functions composing the *cfg80211* API. In short, it looks like this:

```
struct wiphy {
        // A wiphy can have one "official" MAC address, but the
        // virtual addresses have their own and in practice, have
        // the last word. An address mask allows to generate a number
        // of addresses for new interfaces, however:
        // wlp1s0 - ca:fe:ca:fe:00:01
        // wlp2s0 - ca:fe:ca:fe:00:02
        // ...
        u8 perm_addr[ETH_ALEN];
        u8 addr_mask[ETH_ALEN];
        struct mac_address *addresses;

        // Supported interface modes (mask)
        u16 interface_modes;

        // There is always some firmware blob creeping in, might as
        // well acknowledge it
        char fw_version[ETHTOOL_FWVERS_LEN];
        u32 hw_version;

        u32 available_antennas_tx;
        u32 available_antennas_rx;

        // Our "per-band list of channels" array: 2,4GHz, 5GHz...
        struct ieee80211_supported_band *bands[NUM_NL80211_BANDS];

        /* dir in debugfs: ieee80211/<wiphyname> */
        struct dentry *debugfsdir;
};
```

- At the beginning of the article, I told you `iw` showed you `wiphy` and `ieee80211_vif` objects. I lied... :) It does manipulate `wiphy` , but the more generic `struct net_device` objects is actually used instead of `ieee80211_vif` ; though it might seem a bit confusing at first. We will find it almost everywhere in the set of functions below.
- And here is the API (I count 112 functions here) in `struct cfg80211_ops` ; as usual, shortened for clarity. The most observant might start to recognize some of the commands that `iw` offers...

```
struct cfg80211_ops {
        // Will be called by: iw dev phy0 add wlp1s0 type managed
        struct wireless_dev * (*add_virtual_intf)(struct wiphy *wiphy,
                                                  const char *name,
                                                  unsigned char name_assign_type,
                                                  enum nl80211_iftype type,
                                                  struct vif_params *params);
        // Will be called by: iw dev wlp1s0 del
        int     (*del_virtual_intf)(struct wiphy *wiphy,
                                    struct wireless_dev *wdev);
        // Might by called by a plethora of commands, including
        // those changing the type/mode
        int     (*change_virtual_intf)(struct wiphy *wiphy,
                                       struct net_device *dev,
                                       enum nl80211_iftype type,
                                       struct vif_params *params);

        // We also find functions clearly oriented towards making
        // us an access point on our own
        int     (*start_ap)(struct wiphy *wiphy, struct net_device *dev,
                            struct cfg80211_ap_settings *settings);
        int     (*change_beacon)(struct wiphy *wiphy, struct net_device *dev,
                                 struct cfg80211_beacon_data *info);
        int     (*stop_ap)(struct wiphy *wiphy, struct net_device *dev);

        // Steps to the authentication mecanism of 802.11; challenges
        // and extra handshakes might happen inbetween
        int     (*auth)(struct wiphy *wiphy, struct net_device *dev,
                        struct cfg80211_auth_request *req);
        int     (*assoc)(struct wiphy *wiphy, struct net_device *dev,
                         struct cfg80211_assoc_request *req);
        int     (*connect)(struct wiphy *wiphy, struct net_device *dev,
                           struct cfg80211_connect_params *sme);
};
```

As noted earlier, delving into nl80211 would not be that interesting as it is mostly reformatting what we saw in cfg80211.

In part II, we'll dive into *mac80211_hwsim* and see how it uses these APIs to implement a simple, virtual WLAN adapter. We'll show **practical examples,** from `iw` to the driver, and see how it fits together.

Sources

1. A brief, practical introduction to mac80211_hwsim: https://wireless.wiki.kernel.org/en/users/Drivers/mac80211_hwsim (https://wireless.wiki.kernel.org/en/users/Drivers/mac80211_hwsim)

2. Short talk I gave at Smile ECS about WLAN regulation domains, and how it constraints communication in various countries: https://www.youtube.com/watch?v=uR6xqqsw7X0&feature=youtu.be (https://www.youtube.com/watch?v=uR6xqqsw7X0&feature=youtu.be)
3. IEEE 802.11 standard part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. Available here: https://ieeexplore.ieee.org/document/7786995 (https://ieeexplore.ieee.org/document/7786995)
4. The Intel Centrino 8260 taken as example for the beginning of this article: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/dual-band-wireless-ac-8260-brief.pdf (https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/dual-band-wireless-ac-8260-brief.pdf)
5. Wikipedia page on EAP authentication framework: https://en.wikipedia.org/wiki/Extensible_Authentication_Protocol (https://en.wikipedia.org/wiki/Extensible_Authentication_Protocol)
6. Marcel Holtmann's talk on the need for better WLAN tooling in Linux (starting to be available): https://www.youtube.com/watch?v=QIqT2obSPDk (https://www.youtube.com/watch?v=QIqT2obSPDk)
7. Steve deRosier's talk on integrating a WLAN chip into your board, and how to debug when it goes astray: https://www.youtube.com/watch?v=dDMNNDTzjQ0 (https://www.youtube.com/watch?v=dDMNNDTzjQ0)

Mots-clés : kernel (/tag/kernel) Linux (/tag/linux) network (/tag/network) wifi (/tag/wifi) wlan (/tag/wlan)

← La cuisine Alsa (/2020/04/la-cuisine-alsa)

Ordonnancement temps réel souple et affinité CPU sous Linux Vanilla → (/2020/06/ordonnancement-temps-reel-souple-et-affinite-cpu-sous-linux-vanilla)

Laisser un commentaire

Votre adresse de messagerie ne sera pas publiée.

Commentaire

Nom

Adresse de messagerie

Site web

Laisser un commentaire

Catégories

Actualité (/category/actualite)
HowTo (/category/howto)
Interview (/category/interview)
Non classé (/category/non-classe)
Technologie (/category/technologie)
WhitePaper (/category/whitepaper)

Archives

Mars 2021 (/liste-articles/202103)
Janvier 2021 (/liste-articles/202101)
Décembre 2020 (/liste-articles/202012)
Octobre 2020 (/liste-articles/202010)
Septembre 2020 (/liste-articles/202009)
Août 2020 (/liste-articles/202008)
Juillet 2020 (/liste-articles/202007)
Juin 2020 (/liste-articles/202006)
Mai 2020 (/liste-articles/202005)
Avril 2020 (/liste-articles/202004)

Mars 2020 (/liste-articles/202003)
Février 2020 (/liste-articles/202002)
Janvier 2020 (/liste-articles/202001)
Décembre 2019 (/liste-articles/201912)
Novembre 2019 (/liste-articles/201911)
Octobre 2019 (/liste-articles/201910)
Septembre 2019 (/liste-articles/201909)
Août 2019 (/liste-articles/201908)
Juillet 2019 (/liste-articles/201907)
Juin 2019 (/liste-articles/201906)
Mai 2019 (/liste-articles/201905)
Avril 2019 (/liste-articles/201904)
Mars 2019 (/liste-articles/201903)
Février 2019 (/liste-articles/201902)
Décembre 2018 (/liste-articles/201812)
Novembre 2018 (/liste-articles/201811)
Octobre 2018 (/liste-articles/201810)
Septembre 2018 (/liste-articles/201809)
Août 2018 (/liste-articles/201808)
Juillet 2018 (/liste-articles/201807)
Juin 2018 (/liste-articles/201806)
Mai 2018 (/liste-articles/201805)
Avril 2018 (/liste-articles/201804)
Mars 2018 (/liste-articles/201803)
Février 2018 (/liste-articles/201802)
Janvier 2018 (/liste-articles/201801)
Décembre 2017 (/liste-articles/201712)
Novembre 2017 (/liste-articles/201711)
Octobre 2017 (/liste-articles/201710)
Septembre 2017 (/liste-articles/201709)
Août 2017 (/liste-articles/201708)
Juillet 2017 (/liste-articles/201707)
Juin 2017 (/liste-articles/201706)
Mai 2017 (/liste-articles/201705)
Avril 2017 (/liste-articles/201704)
Mars 2017 (/liste-articles/201703)
Février 2017 (/liste-articles/201702)
Janvier 2017 (/liste-articles/201701)
Décembre 2016 (/liste-articles/201612)
Novembre 2016 (/liste-articles/201611)
Octobre 2016 (/liste-articles/201610)
Septembre 2016 (/liste-articles/201609)
Août 2016 (/liste-articles/201608)
Juillet 2016 (/liste-articles/201607)
Mai 2016 (/liste-articles/201605)

Mars 2016 (/liste-articles/201603)
Février 2016 (/liste-articles/201602)
Décembre 2015 (/liste-articles/201512)
Octobre 2015 (/liste-articles/201510)
Août 2015 (/liste-articles/201508)
Juillet 2015 (/liste-articles/201507)
Juin 2015 (/liste-articles/201506)
Mai 2015 (/liste-articles/201505)
Avril 2015 (/liste-articles/201504)
Mars 2015 (/liste-articles/201503)
Février 2015 (/liste-articles/201502)
Janvier 2015 (/liste-articles/201501)
Décembre 2014 (/liste-articles/201412)
Novembre 2014 (/liste-articles/201411)
Octobre 2014 (/liste-articles/201410)
Septembre 2014 (/liste-articles/201409)
Août 2014 (/liste-articles/201408)
Juillet 2014 (/liste-articles/201407)
Juin 2014 (/liste-articles/201406)
Mai 2014 (/liste-articles/201405)
Avril 2014 (/liste-articles/201404)
Mars 2014 (/liste-articles/201403)
Février 2014 (/liste-articles/201402)
Janvier 2014 (/liste-articles/201401)
Décembre 2013 (/liste-articles/201312)
Novembre 2013 (/liste-articles/201311)
Octobre 2013 (/liste-articles/201310)
Septembre 2013 (/liste-articles/201309)
Août 2013 (/liste-articles/201308)
Juillet 2013 (/liste-articles/201307)
Juin 2013 (/liste-articles/201306)
Mai 2013 (/liste-articles/201305)
Avril 2013 (/liste-articles/201304)
Mars 2013 (/liste-articles/201303)
Février 2013 (/liste-articles/201302)
Janvier 2013 (/liste-articles/201301)
Septembre 2012 (/liste-articles/201209)
Avril 2012 (/liste-articles/201204)
Mars 2012 (/liste-articles/201203)
Janvier 2012 (/liste-articles/201201)
Décembre 2011 (/liste-articles/201112)
Octobre 2011 (/liste-articles/201110)
Août 2011 (/liste-articles/201108)
Juin 2011 (/liste-articles/201106)
Mai 2011 (/liste-articles/201105)

Mars 2011 (/liste-articles/201103)
Février 2011 (/liste-articles/201102)
Janvier 2011 (/liste-articles/201101)
Décembre 2010 (/liste-articles/201012)
Novembre 2010 (/liste-articles/201011)
Octobre 2010 (/liste-articles/201010)
Septembre 2010 (/liste-articles/201009)