Setting up a WiFi Access Point on your Slackware Box

You should read this, regardless of whether you are a Slackware enthusiast or not, because you will find most of it applicable to other distributions too. I have recently reused most of this on a debian/DietPI based NanoPi R1S H3 just because it was fester for me to get something functional like that rather than tinkering with uboot and kernel to get slackwareARM running on it.

1 Preamble

There are endless reasons for wanting to run an AP from a standard Linux environment as opposed to the crippled ones that come in most AP appliances, we're not here to discuss them all but if you're reading this you've found your own motivation for doing it.

Out of the box Slackware can't be used to create a Wireless Access Point, but don't panic it's got almost everything you need, with just one thing that needs to be gotten from outside the official Slackware packages and a bit of configuration you can achieve what most low end AP offer. After all most of the home/small office AP in commerce run a really crippled Linux on top of ridiculously underpowered hardware it's just a matter of getting the necessary software and configuring it ! Let's take a look at how you could go about it.

2 Hardware Requirements

I'm assuming that almost any Linux capable hardware has the ability to deal with wired and wireless NICs, since that's what we'll need in terms of hardware for making an AP and because it doesn't really matter to what bus the NICs are attached to (pci,pcie,pcix or even usb), almost any Linux capable hardware will be ok. You will choose hardware that can handle the traffic load you want to target. One thing you need to check is that your wireless supports software AP by looking at the output of

```
iw list
```

Look at the section

```
 Supported interface modes:
              * AP/VLAN
              * monitor
```

AP needs to be among the Supported interface modes.

Ok there is still some Wireless NICs that support hardware master mode but that's 3 against all the rest I'm not going to bother going into any detail with those 3 exceptions, google is your friend if you want to insist in the hardware master mode way.

Should you want to have Virtual Access Poits (more than one AP on a single physical wireless NIC) you will want to look at the section “valid interface combinations”. Here's the output from two cards :

```
[root@nuc8i5 ~]# for Phy in $(iw dev |grep ^phy); do   echo "$Phy";   iw
$Phy info | grep  "valid interface combinations" -A 2; done
phy#1
      valid interface combinations:
               * #{ AP, mesh point } <= 8,
                 total <= 8, #channels <= 1
phy#0
      valid interface combinations:
               * #{ managed } <= 1, #{ AP, P2P-client, P2P-GO } <= 1, #{
P2P-device } <= 1,
                 total <= 3, #channels <= 2
[root@nuc8i5 ~]#
```

The first one would allow more than on vAP (up to a maximum of 8) but they all got to be on the same channel. The second one would allow for one AP and one managed client to be used at the same time even of different channels but the total number of live APs need to be no more than one.

3 Software Requirements

Here your mileage may vary depending on the type of setup you want but let's keep things simple to get started by offering similar things to what most low end Access points have: dhcp, dns forwarder, WPA PSK, firewalling and bridging wired and wireless interfaces. Even at that there are several possibilities with the shipped Slackware packages: you could run bind and full dhcp server but for the sake of simplicity let's combine those 2 in one software that handles both (dnsmasq). If you already run bind or dhcp server you probably know how to get them working for the AP anyway.

Low end AP's also support WEP, but there's an issue with it's implementation that makes it very vulnerable (search wep vulnerability and read more about it or just believe me that even with an extremely low performance machine you can crack WEP keys). Using a WEP protected Wifi is basically the same as not having any protection at all so I'm not going to consider that option. Later on I might talk mention how to go about not having any protection at all but it's something you should really avoid so for the moment let's do things with WPA-PSK security which is the WiFi entry level security.

Another common mistake is to make hidden SSID WiFi networks: in terms of security it's actually less secure because making it harder for the clients to find the AP actually makes it easier for a malicious user to hack into the Wireless Lan. Let's not make our WiFi with a hidden ssid.

3.1 Required Kernel Options

The kernel shipped with Slackware has all you need but just in case you build your own you will need to have these options checked along with whatever is required for your hardware to work properly:

- 802.1d Ethernet Bridging
- Network packet filtering framework (Netfilter) [and a lot of sub options in there] (*)
- cfg80211 - wireless configuration API
- cfg80211 wireless extensions compatibility
- Common routines for IEEE802.11 drivers
- Generic IEEE 802.11 Networking Stack (mac80211)

(*) If you're not familiar with iptables your best bet is to check all sub options that are not experimental

3.1.1 Runtime Kernel Parameters

Here are the runtime kernel parameters I've grouped together in one place, like some other distros, in /etc/sysctl.conf

```
net.ipv6.conf.all.disable_ipv6=1
net.ipv4.ip_forward=1
net.ipv4.conf.all.rp_filter=1
net.ipv4.conf.default.rp_filter=1
#use this if you get odd behavior vaguely resembling failing to find pmtu
#net.ipv4.tcp_sack=0
net.ipv4.ip_dynaddr=1
net.ipv4.conf.all.accept_source_route=0
net.ipv4.conf.default.accept_source_route=0
net.ipv4.conf.all.accept_redirects=0
net.ipv4.conf.default.accept_redirects=0
#use this if icmp black holes become an issue for you
#net.ipv4.tcp_mtu_probing=1
#net.ipv4.tcp_base_mss=1024
```

3.2 Slackware Packages

- dnsmasq
- bridge-utils
- iptables
- wireless-tools
- iw
- libnl
- openssl
- ppp (*)
- rp-pppoe (*)

(*) only if you want your AP to actually manage your internet connection trough some sort of point PPP modem.

3.3 Other Software

- hostapd
- miniupnpd (only required if you have appliances that require upnp)

Neither of these are included in the slackare installation packages, only hostap is mandatory to gae basic AP functionality. To obtain hostapd you could either download a binary version from some reliable source or compile it from sources. You might be slightly better off compiling from sources, so that you get e fairly recent version, but it's up to you. Just don't try using really old hostapd versions like 0.6.7 on 2.6+ kernels: hostapd has followed the kernel wireless stack drivers and transitioned from only supporting FullMAC devices to supporting SoftMAC on almost all devices trough nl80211. Nowadays the combination of both hardware and driver supporting FullMAC is hard to come by (Prism2/2.5/3, and Atheros ar521x) all the others need to go the SoftMAC way so I suggest ignoring the 2 that could still go the FullMAC way and just go for SoftMAC for all, allowing you to use almost any WiFi card for creating an AP. If you like you can read more about mac80211 here.

Hostapd sources can be downloaded from here, you should be looking at the most recent stable version (2.9 last time this article was edited) and avoid the development/old branches. Compiling hostapd is really simple:

1. extract the sources
2. enter the source tree (you should see something like this: CONTRIBUTIONS COPYING README hostapd/ patches/ src/)
3. enter the hostapd sub-directory (we shall refer to this folder as HOSTAPD_SOURCE_TREE from here on)
4. edit the defconfig file and enable any optional features you need (default is fine for a simple setup)
5. copy the defconfig file to .config
6. make a softlink for netlink includes (cd /usr/include; ln -s libnl3/netlink .)
7. make
8. make install (or optionally just put in /usr/local/bin just the hostapd binary)

If you are using a cross compiler to target non x86 hardware make sure you have set it up to right: if you have trouble with that you can try a native build as building hostapd is really simple and not demanding on the hardware.

3.3.1 File Integrity

If your AP also acts a router it's probably going to be exposed to malware and even if you do your best to keep malicious users out of your work they might still find a way in. If your router is exposed to internet you might want to consider some sort of file integrity tool that would alert you if files have been tampered with. File integrity checking could be a whole article so I'm not going to go into any detail beyond advising to read more about it on security oriented communities like security focus or maybe just do a google search on “file integrity tool”.

4 Configuring

Now let's have a look at how to configure everything so that it will work right.

4.1 Hostapd

Create hostapd folder inside /etc and copy into it a few files:

```
mkdir /etc/hostapd
cp HOSTAPD_SOURCE_TREE/hostapd.conf /etc/hostapd/hostapd.template
cp HOSTAPD_SOURCE_TREE/hostapd.wpa_psk /etc/hostapd/
```

Now supposing that wlan0 is the interface you will be using for creating the AP I suggest that you copy hostapd.template to wlan0.conf or br0.conf if it's bridged.

```
cd /etc/hostapd
cp hostapd.template wlan0.conf
```

alternatively just copy it to hostapd.conf … but if you like to run several AP you might prefer to differentiate their config files.

These are the things you have-to edit in wlan0.conf

- interface=wlan0
- driver=nl80211
- ssid=your_ssid
- hw_mode=g #to keep it simple don't attempt n mode right away even if your hardware supports it
- channel=6 #or whatever other channel you prefer
- ieee80211d=1
- country_code=IT
- ieee80211n=1
- auth_algs=1
- macaddr_acl=1 # see notes below
- # 0 = accept unless in deny list (iptables mac filtering and optionally have a ban list)
- # 1 = deny unless in accept list (mandatory to have a mac address ACL)
- accept_mac_file=/etc/hostapd/wlan0.accept
- # deny_mac_file=/etc/hostapd/wlan0.deny
- wpa=2
- wpa_psk_file=/etc/hostapd/wlan0.wpa_psk
- wpa_key_mgmt=WPA-PSK
- rsn_pairwise=CCMP
- wpa_group_rekey=600
- wpa_gmk_rekey=86400
- wpa_ptk_rekey=600

This will use WPA-PSK/WPA2-PSK authentication. Have a look here in the section “A good starting

point for a wpa & wpa2 enabled access point is:" for more details on authentication.

There are pros and cons for dealing with mac address filtering either with iptables or with mac ACL, to put it in really short terms the main difference is that doing it with iptables allows you to keep all filtering stuff in one place but has a tendency to get lengthy if your network has many clients while doing it with mac ACL will require editing of separate file (containing just a newline separated list of mac addresses) but can really simplify things if you have several wired and wireless clinets.

The above hostapd configuration has mac ACL so if you followed it closely you should now create your /etc/hostapd/wlan0.accept containing a new line separated list of macs you wish to allow into your wireless network. Failing to do so will not allow any of your wireless clients to authenticate even with the correct credentials.

There is also an option for telling hostapd about bridging but the comment in hostapd.conf states: "If the bridge parameter is not set, the drivers will automatically figure out the bridge interface (assuming sysfs is enabled and mounted to /sys) and this parameter may not be needed". I followed that to the letter and ignored altogether

```
#bridge=br0
```

and besides that we'll be brigding the interfaces after having setup the AP. I've not played with this option but I suspect that enabling it and/or brigding before starting up hostapd may result in having the wired clients mandated to using WPA too.

Now it's time to deal with the passphrase for associating: there is more then one way of dealing with the passwords but I like to use wpa_psk_file as it allows per NIC passwords, and even have different passwords for your different Wifi staions. If you want to keep things simple just use the special MAC "00:00:00:00:00:00" that will match with every MAC thus having the same passphrase for all the clients. Here's an example:

```
00:00:00:00:00:00 your_psk_for_all_stations
```

Now you should be ready to fire up hostapd

```
/usr/local/bin/hostapd -B /etc/hostapd/wlan0.conf
iwconfig  wlan0
wlan0     IEEE 802.11bg  Mode:Master  Tx-Power=20 dBm
        Retry  long limit:7   RTS thr:off   Fragment thr:off
        Power Management:on
```

and iwconfig should report that the interface is in master mode. If this fails you either got an old version of hostapd that does not support nl80211 of your kernel lacks the necessary options for supporting nl80211.

If you prefer to use iw instead this is the sort of output you should have:

```
iw dev wlan0 info
Interface wlan0
      ifindex 4
      type AP
```

```
      wiphy 0
      channel 6 (2437 MHz) NO HT
```

4.2 Bridging

Now it's time to bridge together wired and Wireless NICs. It's possible to use the native bridge setup from rc.inet1.conf but you may need to restart bridge after starting hostapd so here are the commands necessary to set it up manually.

```
/usr/sbin/brctl addbr br0
/usr/sbin/brctl stp br0 off #turn off spanning tree unless you know you're
going to make loops
/usr/sbin/brctl addif br0 wlan0 #add the wireless nic in the bridge
/usr/sbin/brctl addif br0 eth0  #add the wired nic in the bridge
/sbin/ifconfig wlan0 0.0.0.0 up #bring up the interfaces
/sbin/ifconfig eth0 0.0.0.0 up
/sbin/ifconfig br0 192.168.0.1 up #give the bridge an ip address or dnsmasq
will not work right
```

4.3 DNS and DHCP Servers

Now it's time to start dnsmasq. You can actually leave it running from boot if you like or even run specific servers at your choice. I now run separate dnsmasq instances for each AP so I've abandoned Slackware's rc.dnsmasq and start it from my custom network scripts which requires separate config files for each interface (like /etc/dnsmasq/br0.conf and /etc/dnsmasw/wlan1.conf). Configuring it is something you should look into to suit best your networking needs … let's just look at some of the most common things. Supposing that you want to assign ip addresses belonging to 192.168.0.0/24 and this are the options you will need:

- interface=br0
- bogus-priv
- local=/local/
- domain=local
- except-interface=lo
- listen-address=192.168.0.1
- dhcp-range=192.168.0.2,192.168.0.254,24h
- dhcp-leasefile=/run/dnsmasq/dnsmasq.leases
- conf-dir=/etc/dnsmasq.d

The options local and domain allow dnsmasq to tell clients they belong to a domain and in return serve as authoritative for that domain. Some distributions that use NetworkManager will not get local resolution to work unless you define these 2 options.

The options except-interface bind-interfaces and listen-address are particularly useful if you want to run more then one instance of dnsmasq. For the bridge to work correctly don't forget to allow forwarding either by using rc.ip_forward or by executing

```
/bin/echo 1 > /proc/sys/net/ipv4/ip_forward
```

See section 3.1.1 for the other useful run-time kernel parameters.

4.4 Firewalling

Now is a good time to configure your firwall protection. Supposing that the box will be routing packages thought it I'll show some rules that you might find helpfull.

Newer kernels nswitched to nftables but you can still use iptables to manipulate the kernel netfilter tables. Actually if you want to move to using nftables but you are not familiar with the syntax you can save the kernel netfilter tables in netfilter format like this:

```
nft list ruleset > nftables.conf
```

Personally I'm still struggling with nft syntax so I will keep the rest of this chapter in iptabls syntax: you can convert, like I showed above, after you are done.

This is the output of iptables-save, you can edit it to make the changes you require and then pipe your edited file to iptables-restore. The iptables-save/iptables-restore is a handy way of keeping configuration for easy firewall activation and editing.

```
*mangle
:PREROUTING ACCEPT [16570:1916193]
:INPUT ACCEPT [16553:1912438]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]
-A POSTROUTING -p tcp --tcp-flags SYN,RST SYN -j TCPMSS --clamp-mss-to-pmtu
-m comment --comment "fix packet size for stuff that's being routed through
this box (SEE NOTE *)"
COMMIT
*nat
:PREROUTING ACCEPT [1906:207422]
:POSTROUTING ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
-A POSTROUTING -s 192.168.0.0/24 -j MASQUERADE
COMMIT
*filter
:INPUT DROP [1906:207422]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
-A INPUT -p all -m conntrack --ctstate ESTABLISHED,RELATED  -j ACCEPT
-A INPUT -p icmp -s 192.168.1.0/24 -j ACCEPT
-A INPUT -p tcp --dport 53 -s 192.168.0.0/24 -j ACCEPT -m comment --comment
"allow dns from LAN"
-A INPUT -p udp --dport 53 -s 192.168.0.0/24 -j ACCEPT -m comment --comment
```

```
"allow dns from LAN"
-A INPUT -p tcp --dport 67 -i br0 -j ACCEPT -m comment --comment "allow dhcp
from LAN"
-A INPUT -p udp --dport 67 -i br0 -j ACCEPT -m comment --comment "allow dhcp
from LAN"
-A INPUT -m mac --mac-source 0a:0b:0c:0d:0e:0f -j ACCEPT -m comment --
comment "this one can acces the AP "
-A FORWARD -p all -m conntrack --ctstate ESTABLISHED,RELATED  -j ACCEPT
-A FORWARD -m mac --mac-source 00:01:02:03:04:05 -j ACCEPT -m comment --
comment "this one can route trough the AP"
COMMIT
```

NOTE *: Clamping MSS to PMTU can get internet browsing from your LAN working but can break VPN packets. The proposed workaround has been made necessary by the increasing tendency of failing to find PMTU. It is not always necessary for you to activate the workaround but be warned that it can equally inadvertently stop working leaving you with an intermittent problem that is difficult to debug. It can be intermittent because the path your packets take to arrive to any destination on internet is not always the same due to the necessity of fault tolerance. This note could become a whole article on it's own so here's an intresting read on the issue.

I generally put the content above in /ect/firewall.cf and add “/usr/sbin/iptables-restore < /etc/firewall.cf” to rc.local.

If you want to manage your AP via ssh now is a good time to restart sshd that did not appreciate taking down the interfaces it was binding on (back in section 4.2).

```
/etc/rc.d/rc.sshd restart
```

If you're going to share internet connection you might want to stop ssh access from internet by adding a firewall rule to regulate it or making sshd bind only to the address assigned to br0. The config shown above will not allow incoming ssh traffic from the internet link (by the connection tracking rule) but you may want to back that up with further safety. Longer iptables chains generally have bad impact on firewall performance so you might want to add something like this to /etc/ssh/sshd_config:

```
ListenAddress 192.168.0.1
```

At this point you should be able to associate clients to the AP. With the above iptables rules client with MAC 0a:0b:0c:0d:0e:0f can associate and access the AP itself but not route trough the AP (with the exception of dns querys that get forwarded by dnsmasq), client with MAC 00:01:02:03:04:05 can route thought the AP but not access the AP itself. The example was just to show clearly the difference of having packets go into the box and routing packets through the box as this behavior was radically different in the ip_chains. Another way to implement even more complex MAC ACL to decide who manages AP, who can only route through and who can do both can be done by using ebtables (need to add that to the software list not present in Slackware) and fiddle with the broute chain. In all cases you might find interesting the netfilter flow diagram.

If you don't want MAC address ACL or you only implement MAC ACL for the AP you could replace

```
-A INPUT -p icmp -s 192.168.1.0/24 -j ACCEPT
-A INPUT -p tcp --dport 53 -s 192.168.0.0/24 -j ACCEPT -m comment --comment
```

```
"allow dns from LAN"
-A INPUT -p udp --dport 53 -s 192.168.0.0/24 -j ACCEPT -m comment --comment
"allow dns from LAN"
-A INPUT -p tcp --dport 67 -i br0 -j ACCEPT -m comment --comment "allow dhcp
from LAN"
-A INPUT -p udp --dport 67 -i br0 -j ACCEPT -m comment --comment "allow dhcp
from LAN"
-A INPUT -m mac --mac-source 00:01:02:03:04:05 -j ACCEPT -m comment --
comment "some comment"
-A FORWARD -m mac --mac-source 00:01:02:03:04:05 -j ACCEPT -m comment --
comment "some other comment"
```

With something like this

```
-A INPUT -p all -i br0 -j ACCEPT
-A FORWARD -p all -i br0 -j ACCEPT
```

Most off the shelf AP also let you do a number of port forwarding, this is also an iptables job. Remember that we are masquerading all outgoing traffic to look like it's coming from the AP itself so we only need to create rules of the incoming traffic. Supposing you want to run a web server on 192.168.0.2 you might want to add a rule like this in the FORWARD chain of the filter table:

```
-A FORWARD -p tcp -d 192.168.0.2 -m multiport --dports 80,443 -j ACCEPT -m
comment --comment "allow http traffic to be routed thought the box only to
the correct server"
```

and a rule like this in the PREROUTING chain of the nat table:

```
-A PREROUTING -p tcp -m multiport --dports 80,443 -j DNAT --to-destination
192.168.0.2 -m comment --comment "nat incomming http requests to local
destination before routing"
```

To understand how this works you need to look at the netfilter flow diagram: As a http request arrives to the AP from the internet link it will first have the DNAT changed in the nat prerouting stage, but it would get dropped by the filter policy if we don't find a way to let it in, that's where the forward filter rule comes in to play.

If you start going crazy over transferring large files over fast networks for a problem that seems like mtu related but is not you might want to consider turning net.ipv4.tcp_sack off.

If your ISP gives you some sort of traffic quota you may want to add some quotas to your firewall configuration. You may fully understand the consequences of streaming on your ISP quota but maybe the rest of the family may not: giving them a quota might save you a fit when you need to do an urgent job that requires internet connection. There are various ways you could go about putting quotas on specific clients on your LAN just keep in mind a few things:

- rules with quotas stop matching once quota is exceeded
- flushing your tables will reset all quota counters
- quota counters do not reset themselves whenever your ISP resets your internet quota

Here's an example of how you could go about putting a quota on the FORWARD chain to stop a client using more then 300Mb daily:

```
-A FORWARD -p all -m conntrack --ctstate ESTABLISHED,RELATED ! -d
192.168.0.200 -j ACCEPT -m comment --comment "allow related traffic but not
for 192.168.0.200 that has a quota"
-A FORWARD -p all -m conntrack --ctstate ESTABLISHED,RELATED -d
192.168.0.200 -j ACCEPT -m quota --quota 314572800 -m comment --comment
"allow quota related traffic for 192.168.0.200"
-A FORWARD -s 192.168.0.200 -j ACCEPT -m quota --quota 314572800 -m comment
--comment "outgoing quota for 192.168.0.200"
-A FORWARD -d 192.168.0.200 -j ACCEPT -m quota --quota 314572800 -m comment
--comment "incoming quotafor 192.168.0.200"
-A FORWARD -s 192.168.0.200 -d 192.168.0.0/24 -j ACCEPT -m quota --quota
314572800 -m comment --comment "allow LAN traffic anyway for 192.168.0.200"
```

Or you could use a user-defined chain to group all your quoted traffic into a single quota like this:

```
-A FORWARD -d 192.168.1.0/24 -m conntrack --ctstate RELATED,ESTABLISHED -m
comment --comment "allow quota related traff. to quoted LAN" -j QUOTA
-A FORWARD -s 192.168.1.0/24 -m comment --comment "allow quoted traff. from
quoted LAN" -j QUOTA
-A QUOTA -m quota --quota 314572800 -m comment --comment "accept traffic
within quota in this userdefined chain" -j ACCEPT
-A QUOTA -m comment --comment "when quota is exceeded start rejecting" -j
REJECT --reject-with icmp-port-unreachable
```

Along with this you need to flush the iptables counters every day (or every how often you wnat the quota counters to reset) with something like this

```
iptables -Z
```

I generally do this with an AT job because AT has less adverse effects then CRON on readonly systems. Here's one possible way of making an AT job re schedule itself every day at 00:30:

```
# cat /usr/local/bin/flush_iptables_counters
/usr/sbin/iptables -Z
/usr/bin/at -f /usr/local/bin/flush_iptables_counters "0030 tomorrow"
#
```

Just run it once and it should then re schedule itself. On a readonly system you will need to have atjobs directory on tmpfs and run it the first time from rc.local.

4.5 PPP

Tectonically you're not going to need this on a pure access point (AP) but it's common that the AP also acts as router for your internet access, in this case you will need to configure your PPP link from your AP. Depending on how your ISP provides your internet access you might or might not have an external modem which may or may not understand pppoe protocol. Nowadays it seems to have

become less common to have internal *DSL modem wile external modems have become more popular. Most of them use pppoe protocol. Technically there's not much difference between configuring an internal modem with just the use of pppd on an external modem that uses pppoe, the former will require an extra package (rp-pppoe) and just a few extra options in the configuration file. I'll show how to configure with the use of pppoe, if that's not your case just strip off a few options.

If you have installed both ppp and rp-pppoe you will find a that most of the configuration has already been done with most commonly usefull options in /etc/ppp, still there is a limited amount of work that needs to be done and I also advise to detour from the use of pppoe-start to initiate the connection to stick to something that remains the same regardless of whether you're using pppoe or not. To do this I configure a peer for my ISP and add in there a few options instructing pppd to use pppoe plugin.

To do this you need to create a file in /etc/ppp/peers, call it something that will make it obvious to you what the peer will connect to: for instance if your ISM is called “Telco” you might want to create a file called something like /etc/ppp/peers/telco and here's what needs to go in there typically:

```
plugin rp-pppoe.so
eth1 #or whatever nic your modem is connected to
unit 0
user "username your ISM gave you"
noipdefault
noauth
default-asyncmap
defaultroute
hide-password
nodetach
usepeerdns
mtu 1492
mru 1492
noaccomp
nodeflate
nopcomp
novj
novjccomp
lcp-echo-interval 20
lcp-echo-failure 3
persist
maxfail 0           #these 3 options are particularly important if your
connection is prone to braking as the default is just 10
lcp-max-failure 0   #and after that the peer will not be able to persists
any more
lcp-max-configure 0 #so your internet connection will fail with no apparent
reason
holdoff 20
#idle 900
#demand
updetach
```

If your ISM charges you based on time rather than band usage you might want to uncomment the idle and demand options so that the PPP connection will not stay alive when it's not actually in use. Pot as

much as you can here in the peer and as little as possible in the /etc/ppp/options file so that you can possibly have one or more backup peres that are functional without editing any configuration. If these options do not work for you I suggest you read the /usr/doc/rp-pppoe-3.11/HOW-TO-CONNECT documentation and use pppoe-setup for a guided configuration, fire up the connection with pppoe-start and take not of the options used, then edit accordingly your peer and revert back to using pppd call <peer>.

We're not done yet we need to edit pap-secrets or chap-secrets so that pppd can complete authentication. Which file needs to be addressed may depend on your isp but if you put the same content in both then you should be ok whatever your ISP requires.

```
# Secrets for authentication using PAP
# client        server  secret                  IP addresses
"username your ISM gave you"  *       "password your ISM gave you"
```

You can now fire up your internet connection with

```
pppd call telco
```

4.6 Dynamic DNS

If you are going to do the port forwarding thing and if you ISP is not giving you a static IP you might want to use some sort of dynamic DNS service so that people can reach whatever service you are forwarding. There are many free dynamic DNS service providers ... google is your friend if you don't like any of the ones I'm suggesting:

- http://www.gnutomorrow.com/r/namecheap
- http://www.dnsexit.com/
- http://www.dynu.com/
- http://freedns.afraid.org/

Most give you a client for managing the update ... all you need do is arrange for that client to run either from the router or from any other pc in your LAN with internet access.

4.7 Configuring Clients

For the client side configurations there are many ways too, to keep things simple let's just go about it by using wpa_supplicant. Remember that regardless of whether it's a wired or wireless client: if you're using managing MAC ACL from iptables your client's MAC address needs to be added to the firewall configuration while ff you're doing MAC ACL just with hostapd then you only need to make sure your wireless clients are in the ACL.

4.7.1 Wired Clients

In most cases the modules for your wireless NIC get automatically loaded so that should not be an issue. Keep an eye out on kernel ring buffer and messages to see if your device is requesting

firmware that is not available. If all is fine and the configuration is OK it's really simple: just uplink the client to the AP and start your favorite dhcp client.

4.7.2 Slackware Wireless Clients

Configuring a client to access the newly crated AP is best done with a tool like wpa_gui, as it can create the correct settings in wpa_supplicant.conf based on what's detected. So fire up wpa_gui and activate scanning until you find the newly created AP then double click it. On the window that pops up just add the password without changing any of the detected settings and click add.

If, for any reason, you don't want to use a tool like wpa_gui, and if you followed all the AP side setup to the letter you could just simply put this in /etc/wpa_supplicant.conf

```
ctrl_interface=/var/run/wpa_supplicant
ctrl_interface_group=root

network={
      ssid="your_ssid"
      psk="your_psk_for_all_AP"
      key_mgmt=WPA-PSK
      pairwise=CCMP
      auth_alg=OPEN
}
```

Make sure rc.inet1.conf is configured for using wpa_supplicant

```
IFNAME[4]="wlan0"
USE_DHCP[4]="yes"
WLAN_MODE[4]=Managed
WLAN_WPA[4]="wpa_supplicant"
WLAN_WPADRIVER[4]="wext"
```

It should then be possible to restart rc.inet1 (or just rc.inet1 wlan0_down and rc.inet1 wlan0_up) and the client should associate.

There's also the command line alternative with wpa_cli in this example we'll assume that your client is totally unconfigured and wpa_supplicant is not running. We're going to do everything on the command line:

```
# wpa_supplicant -B -W -Dwext -i wlan0 -c /etc/wpa_supplicant.conf
# wpa_cli
wpa_cli v2.4
Copyright (c) 2004-2015, Jouni Malinen <j@w1.fi> and contributors

This software may be distributed under the terms of the BSD license.
See README for more details.

Selected interface 'wlan0'
```

```
Interactive mode

> scan
OK
<3>WPS-AP-AVAILABLE
> scan_results
bssid / frequency / signal level / flags / ssid
02:0c:42:f9:73:23       2412    -58     [WPA-PSK-CCMP][WPA2-PSK-CCMP][ESS]
a4:51:6f:95:37:b6       2462    -58     [WPA2-PSK-CCMP][WPS][ESS]
Windows Phone0377
00:0c:42:f9:73:23       2412    -62     [ESS]   Insecure-WiFi
>  add_network
0
> set_network 0 ssid "Windows Phone0377"
OK
> set_network 0 psk "passwordforcrappywindowsphone"
OK
> enable_network 0OK
OK
<2>Trying to authenticate with a4:51:6f:95:37:b6 (SSID='Windows Phone0377'
freq=2437 MHz)
<2>Trying to associate with a4:51:6f:95:37:b6 (SSID='Windows Phone0377'
freq=2437 MHz)
<2>Associated with a4:51:6f:95:37:b6
<2>WPA: Key negotiation completed with a4:51:6f:95:37:b6 [PTK=CCMP GTK=CCMP]
<2>CTRL-EVENT-CONNECTED - Connection to a4:51:6f:95:37:b6 completed (reauth)
[id=0 id_str=]
> save_config
OK
> quit
#
```

If all went right and your wpa_supplicant.conf file had

```
update_config=1
```

in it the above snippet would have saved the new network to wpa_supplicant.conf and associated you with it.

Remember that if you're associating with an non secured AP you need to use this:

```
> set_network 0 ssid "Insecure-WiFi"
OK
> set_network 0 key_mgmt NONE
OK
>
```

4.7.3 Other Linux Distributions Wireless Clients

I've tried various other flavor distributions ... most don't use wpa_gui for associating to AP but some

sort of other tool that generally pops up when you click on the icon that notifies the presence of an Access Point. After a few headaches I found that best association success is achieved by forcing setup for hidden AP even if the AP I'm configuring has not the hidden essid. You can always use wpa_cli on the command line if it's shipped with whatever distro you prefer.

5 The AP's Components Status

It might be interesting to query the AP status at any time to see how it's doing. There are several ways to go about this, I'll try to show the simplest way that possibly works on any Linux system.

5.1 Associated Stations

The associated stations can be shown by using either hostapd_cli or iw:

```
root@router:~# hostapd_cli -p /run/hostapd all_sta
Selected interface 'wlan0'
00:01:02:03:04:05
flags=[AUTH][ASSOC][AUTHORIZED][SHORT_PREAMBLE][WMM]
aid=2
capability=0x431
listen_interval=5
supported_rates=02 04 0b 16 0c 12 18 24 30 48 60 6c
timeout_next=NULLFUNC POLL
dot11RSNAStatsSTAAddress=00:01:02:03:04:05
dot11RSNAStatsVersion=1
dot11RSNAStatsSelectedPairwiseCipher=00-0f-ac-4
dot11RSNAStatsTKIPLocalMICFailures=0
dot11RSNAStatsTKIPRemoteMICFailures=0
hostapdWPAPTKState=11
hostapdWPAPTKGroupState=0
rx_packets=16176
tx_packets=20464
rx_bytes=1974499
tx_bytes=23121726
connected_time=439
root@router:~#
```

```
root@router:~# iw wlan0 station dump
Station 00:01:02:03:04:05 (on wlan0)
      inactive time:  0 ms
      rx bytes:       1956931
      rx packets:     16009
      tx bytes:       23105532
      tx packets:     20352
      tx retries:     1484
```

```
      tx failed:      1
      signal:         -63 dBm
      signal avg:     -62 dBm
      tx bitrate:     48.0 MBit/s
      authorized:     yes
      authenticated:  yes
      preamble:       short
      WMM/WME:        yes
      MFP:            no
      TDLS peer:              no
root@router:~#
```

5.2 DHCP Leases

You can dump dnsmasq's lease file to see the dhcp leases

```
root@router:~# cat /run/dnsmasq/br0.leases
1411875361 00:01:02:03:04:05 192.168.0.3 b3bo *
1411874427 0a:0b:0c:0d:0e:0f 192.168.0.4 printsrv *
root@router:~#
```

5.3 ARP Table

The system arp cache cache can be inspected by using the arp command

```
root@router:~# arp -an
? (192.168.0.3) at 00:01:02:03:03:05 [ether] on br0
? (192.168.0.4) at 0a:0b:0c:0d:0e:0f [ether] on br0
root@router:~#
```

5.4 Socket Status

To get a list of the open sockets on your system you can use socklist if you have it installed. If you don't have it installed you can use a somewhat simpler script like the one shown blow (a little less information rich and careless formatting but still useful and much faster then doing it manually):

```
#!/bin/bash

ahex2i ()
{ echo "0x$1" |awk '{printf("%i",strtonum($1))}'
}

hex2ip ()
{ ahex2i ${1:6:2}
  echo -n "."
```

```
  ahex2i ${1:4:2}
  echo -n "."
  ahex2i ${1:2:2}
  echo -n "."
  ahex2i ${1:0:2}
}

getpidofsocket ()
{ find /proc/*/fd -type l -printf "%h %l\n" 2>/dev/null| grep
"socket:\[$1\]" |awk -F/ '{print $3}'
}

echo "          local                   forein  pid/cmd"
grep -v "^ *sl " /proc/net/tcp |awk '{printf("%s %s %s\n",$2,$3,$10)}' |sed
-e "s/:/ /g" | while read LINE
do
  LA=(${LINE})
  PID=$(getpidofsocket ${LA[4]})
  echo "tcp     $(hex2ip ${LA[0]}):$(ahex2i ${LA[1]})   $(hex2ip
${LA[2]}):$(ahex2i ${LA[3]})  ${PID}/$(cat /proc/$PID/comm)"
done

grep -v "^ *sl " /proc/net/udp |awk '{printf("%s %s %s\n",$2,$3,$10)}' |sed
-e "s/:/ /g" | while read LINE
do
  LA=(${LINE})
  PID=$(getpidofsocket ${LA[4]})
  echo "udp     $(hex2ip ${LA[0]}):$(ahex2i ${LA[1]})   $(hex2ip
${LA[2]}):$(ahex2i ${LA[3]})  ${PID}/$(cat /proc/$PID/comm)"
done
```

Please note the above script only works on 2.6 kernels or above. This probably also applies to socklist.

5.5 Virtual AP

If your wireless NIC supports it you might like to run multiple virtual APs. As mentioned in Chapter 2 to be able to do this you will need the “valid interface combinations” sections to have #AP to be greater than one and if you optionally want them to live on separate channels you will need #channels to be greater than 1. If this is the case then you can add virtual AP like this

```
iw phy0 interface add vap0 type __ap
```

or

```
iw wlan0 interface add vap0 type __ap
```

this will create a new virtual AP, arbitrarily called vap0 on which you will need to activate a separate

instance of hostapd as explained in Chapter 4. Incidentally the type can be any of of these (as long as your wireless NIC supports them):

- monitor
- managed
- wds
- mesh
- ibss
- __ap

if you subsequentlt want to remove the virtual AP toy can terminate the hostapd running on it, optionally put the nick in down state and then tell iw you want to delete the virtual device:

```
iw vap0 del
```

6 Remote Administration

I'm not advocating that allowing remote administration from your WAN connection is a good thing but there are times where it may be necessary so here are some tips for minimizing the risk of having your router suffering brute force attacks or other bad things happen to it.

I'm an old fashioned system administrator so for me remote administration is done via ssh, if you've added a nice web administration tool to your AP/Router keep in mind that running apache just for the sake of having remote web administration will expose you to a whole lot of security issues that need to be addressed and maintained over time.

1. use non standard ports
2. disallow password authentication
3. minimize your attack surfaces

Let me give you a little reasoning for the list.

6.1 Use Non Standard Ports

Whatever is your remote administration tool of choice it's a good idea not to leave access to it from WAN on it's well known port, making it less obvious that you run such a service. If you do this there's a good chance that your AP/Router will never get unwanted attention.

6.2 Disallow Password Authentication

Allowing password authentication is a welcome for brute force attacks so avoid it wherever possible (ie for ssh administrations only allow authentication with keys). If you're doing web based remote administration you could send in a key via get and then set a cookie or something like that along with password protected htaccess.

6.3 Minimize Your Attack Surfaces

Your AP/Router should expose to the WAN connection nothing more then what is really needed. Scanning your own AP/Router and closing or disabling unnecessary services to WAN is something you should allays do so that you minimize the attack surfaces should you ever get unwanted attention.

6.3.1 Avoid Running Remote Administration 24x7

If you can have remote administration active only when you need it you're not leaving the attack surface available all the time but then you need an easy way to turn it on when you're away from home. I've two means of doing so:

- If any family member is home it can be temporarily activated by pressing a specific button on the router itself (it's the second button under the blue led in the images below).
- If nobody is home I've modified a 200 line minimal web server (nweb) to listen to requests on a non standard port and temporarily allow remote administration if a specific url is requested.

Whichever way the temporary remote admin is enabled it also get's automatically turned off after some time (should you ever forget to turn it off once you're done).

Nweb is a really basic webserver that only servers static html images and a few archive formats, it does not even allow directory listing. Besides that I have it parse and enable before serving the page … so if you don't physically have the page that enables the remote administration a 403 is returned anyway leaving no clue as to waht was done in response to that request.

If you're interested in nweb you can get it by googling “nweb tiny web server”. You should hit github with something like nweb23.c with some 204 lines of C code. It should be easy for you to modify the source to match your needs.

7 Wrapping It Up

Now that you've done the configuration maybe next time you want to start the AP you want to do it surely more efficiently.

7.1 Simple Starter Script

Just put a few commands in a script to start it up really quickly:

```
/bin/echo 1 > /proc/sys/net/ipv4/ip_forward #you don't need this if you use
rc.ip_forward
#. /etc/rc.d/rc.dnsmasq restart #only if it's not started at boot time
/usr/local/bin/hostapd -B /etc/hostapd/wlan0.conf
/usr/sbin/brctl addbr br0
```

```
/usr/sbin/brctl stp br0 off
/usr/sbin/brctl addif br0 wlan0
/usr/sbin/brctl addif br0 eth0
/sbin/ifconfig wlan0 0.0.0.0 up
/sbin/ifconfig eth0 0.0.0.0 up
/sbin/ifconfig br0 192.168.0.1 up
/usr/sbin/iptables-restore < /etc/firewall.cf
/etc/rc.d/rc.sshd start
```

If you want it to come up at boot time you could run the script from rc.local or even just put those commands directly in there. If you want a neater solution totally integrated in the init scripts read on.

7.2 Modifying Slackware Init Scripts

I've not yet done this but I can suggest a possible way of doing it.

Edit rc.inet1.conf and put in the following things: (that will later be managed by rc.wireless)

```
IPADDR[0]=""
NETMASK[0]=""
USE_DHCP[0]=""
DHCP_HOSTNAME[0]=""

IFNAME[0]="br0"
BRNICS[0]="wlan0 eth0"
IPADDR[0]="192.168.0.1"
NETMASK[0]="255.255.255.0"

IFNAME[4]="wlan0"
WLAN_ESSID[4]=your_ssid
WLAN_MODE[4]=Master
WLAN_RATE[4]="g"
WLAN_CHANNEL[4]="6"
WLAN_IWPRIV[4]="set AuthMode=WPA-PSK | set EncrypType=TKIP | set
WPAPSK=your_psk_for_all_AP"
```

Some of this will be picked up by rc.inet1 (the bridge stuff) and some by rc.wireless but the stock one knows nothing about Master mode. We need to modify rc.wireless to do some extra stuff: pick up values from WLAN_ESSID[4], WLAN_MODE[4], WLAN_CHANNEL[4], WLAN_IWPRIV[4] and sed inline the values in /etc/hostapd/wlan0.conf and /etc/hostapd/hostapd.wpa_psk. Also WLAN_RATE[4]=“g” will be indigestible for the stock rc.wireless if you like you could leave that as 54 and only convert it for Master mode. A special attention needs to go to WLAN_MODE[4]=Master as the stock rc.wireless knows nothing about master mode. Alternatively one could just have IFNAME[4]=“wlan0” and WLAN_MODE[4]=Master and just manually put all the other stuff in etc/hostapd/wlan0.conf.

I had a quick look at rc.inet1 and apparently it starts bridge after wireless so it should be ok without having to restart br0 after AP is initiated.

The sshd server is started after networking so no need to restart that now.

You can write your own rc.firewall or just leave

```
usr/sbin/iptables-restore < /etc/firewall.cf
```

in rc.local.

7.3 Automating AP Startup For USB WiFi Dongles

Udev is very powerful and can do a variety of actions upon detecting certain events, like the appearance of a NIC. In fact it already does that and renames interfaces according to MAC address (have a look at /etc/udev/rules.d/70-persistent-net.rules and see how your interfaces get the same name even if you remove the modules and reinsert them in the wrong order). Apart from renaming NICs and creating device files it can also execute commands or external helper scripts ... this is particularly handy if, for example, you wish that upon plugging a USB Ethernet dongle it automatically assigns an address via DHCP. A few years ago I wrote a nethelper.sh script for my ClashNG that upon detecting LAN devices would run rc.inet1 to start the NIC according to what was configured in rc.inet1.conf. Ok ClashNG is a minimalistic thing that uses busybox to replace most of the binaries but still it might be a useful example. Ignore the part that writes stuff to debug what was going on ... I'm not udev expert.

```
#!/bin/sh
DEVNAME="$1"
COMMAND="$2"

case $DEVNAME in
  eth*|ath*|wlan*|ra*|sta*|ctc*|lcs*|hsi*)
    case $COMMAND in
      'start')
        if [ -x /etc/rc.d/rc.inet1 ]; then
          if ! /sbin/ifconfig | /bin/grep -q "^${DEVNAME} "; then
            /etc/rc.d/rc.inet1 ${DEVNAME}_start
          fi
        fi
      ;;
      'stop')
         if [ -x /etc/rc.d/rc.inet1 ]; then
           if /sbin/ifconfig | /bin/grep -q "^${DEVNAME} "; then
             /etc/rc.d/rc.inet1 ${DEVNAME}_stop
           fi
         fi
         # Does dhcpcd appear to still be running on the
         # interface?  If so, try to stop it.
         if [ -r /etc/dhcpc/dhcpcd-$DEVNAME.pid -o -r /var/run/dhcpcd-
$DEVNAME.pid ]; then
           /sbin/dhcpcd -k $DEVNAME
           # Force garbage removal, if needed:
           if [ -r /etc/dhcpc/dhcpcd-$DEVNAME.pid ]; then
```

```
            /bin/rm -f /etc/dhcpc/dhcpcd-$DEVNAME.pid
          elif [ -r /var/run/dhcpcd-$DEVNAME.pid ]; then
            /bin/rm -f /var/run/dhcpcd-$DEVNAME.pid
          fi
        fi
        # If the interface is now down, exit with a status of 0:
        if /sbin/ifconfig | /bin/grep -q "^${DEVNAME} " ; then
          exit 0
        fi
      ;;
      *)
        logger "usage $0 interface start|stop"
        exit 1
      ;;
    esac
  ;;
  *)
    logger "Interface $DEVNAME not supported."
    exit 1
  ;;
esac
exit 0
```

Although this was written for ClashNG in 2011 it still works right with Slackware 14.1, just modify udev rules to correctly use the helper script like shown below and it's done.

```
root@r2d2:/tmp/clashng/lib/udev/rules.d# grep  "nethelper\.sh" *
90-network.rules:SUBSYSTEM=="net", NAME=="?*", ACTION=="add",
RUN+="nethelper.sh $env{INTERFACE} start"
90-network.rules:SUBSYSTEM=="net", NAME=="?*", ACTION=="remove",
RUN+="nethelper.sh $env{INTERFACE} stop"
root@r2d2:/tmp/clashng/lib/udev/rules.d#
```

These udev rules were in 90-network.rules which no longer exists so I suggest you put these rules in 90-local.rules in current Slackware versions. Without further modification (as long as the rc.inet1.conf has USE_DHCP[?]="yes" for the interface that will be assigned to your dongle) this will allow for USB Ethernat dongles to be configured via DHCP automatically when you plug them in to your box. Remember to do a "/etc/rc.d.rc.udev reload" after changing udev rules.

At the time I also had a modified rc.wireless that would also start up AP … but I was using a different configuration scheme dew to busybox ash limitations (arrays were a problem) and other requirements I had in mind at the time … but never the less here's the section relevant to AP and it was heavily based on Slackware's rc.wireless anyway. Have a look at the code script fragments below to get ideas non how you might want to go about it:

```
# If we stop a wireless interface using wpa_supplicant,
# we'll kill its wpa_supplicant daemon too and exit this script:
if [ "$2" = "stop" ]
then
  if [ "${WLAN_MODE}" = "Master" ]
  then
```

```
    echo "$0: $2 $1 AP mode" |$LOGGER
    /bin/kill $(/usr/bin/fuser /var/run/hostapd/$1 2>&1 |/usr/bin/awk
'{print $NF}')
  fi
  WPAPID=`echo \`ps axww|grep wpa_supplicant |grep i${INTERFACE}\` |cut -f1
-d' '`
  [ ${WPAPID} ] &&  kill ${WPAPID}
  return 0
fi
...
...
...
  if [ "$MODE" = "Master" ]
  then
    echo "$0: $2 $1 AP mode" |$LOGGER
    if [ -r /etc/hostapd/${1}.conf -a "$2" = "start" ]
    then
      [ $DEBUG -eq 1 ] && \
        /usr/bin/hostapd -B -d /etc/hostapd/${1}.conf >/tmp/hostapd.log 2>&1
|| \
        /usr/bin/hostapd -B /etc/hostapd/${1}.conf
    else
      /bin/kill $(/usr/bin/fuser /var/run/hostapd/$1 2>&1 |/usr/bin/tr -d
[a-z])
    fi
  else
  echo "$0:  $IWCOMMAND mode $MODE" | $LOGGER
# if $IWCOMMAND fails, try taking the interface down to run it.
# Some drivers require this.
    if ! $IWCOMMAND mode $MODE 2> /dev/null
    then
      $IFCOMMAND down
      $IWCOMMAND mode $MODE
      $IFCOMMAND up
      sleep 3
    fi
  fi
fi

# This is a bit hackish, but should do the job right...
[ ! -n "$NICKNAME" ] && NICKNAME=`/bin/hostname`
if [ -n "$ESSID" -a "$MODE" != "Master"]
then
  echo "$0:  $IWCOMMAND nick $NICKNAME" | $LOGGER
  $IWCOMMAND nick $NICKNAME
fi
```

If you do all this stuff right you can get your AP to be automatically initiated when you plug in the USB gongle devoted to doing that.

I used to devote 2 usb dongle for this: one that would be left inserted most of the time and has random generated WAP-PSK (only for family use) and one that would get temporarily plugged in for guests with a much simple WPA-PSK to aid them accessing my home network.

7.3.1 Automation With Custom Scripts

Over the years trying to maintain modified rc scripts functional across updates that involved the rc scripts themselves became cumbersome so I started moving away from modifying the stock scripts and started developing my own stuff. Don't get me wrong I still use and appreciate the stock stuff for my desktop systems. The idea behind my own scripts is based on udev detecting the interfaces (even at boot time). The basic idea is still the same: upon detection udev executes nethelper.sh script that looks for and executes /etc/rc.d/network/<NIC> start. This is not for everyone because it requires manually writing the /etc/rc.d/network/<NIC> script but I think most have the basic knowledge and maybe with a little help most can manage.

Here's is what my latest nethelper.sh looks like:

```
#!/bin/bash
PATH=/usr/local/sbin:/usr/sbin:/sbin:/usr/local/bin:/usr/bin:/bin
DEVNAME="$1"
COMMAND="$2"

case $DEVNAME in
  eth*|ath*|wlan*|ra*|sta*|ctc*|lcs*|hsi*)
    case $COMMAND in
      'start') logger -p local3.info "start: $DEVNAME $COMMAND"
        [ -x /etc/rc.d/network/$DEVNAME ] && /etc/rc.d/network/$DEVNAME
start
      ;;
      'stop') logger -p local3.info "stop: $DEVNAME $COMMAND"
        [ -x /etc/rc.d/network/$DEVNAME ] && /etc/rc.d/network/$DEVNAME stop
      ;;
       *) logger -p local3.info "unsupported command: $DEVNAME $COMMAND" ;;
    esac
  ;;
  *) logger -p local3.info "unsupported interface: $DEVNAME $COMMAND" ;;
esac
exit 0
```

Be warned that the logger facility is not yet functional in the early boot stages so it may not log the interface startup at boot time.

Here's an example of /etc/rc.d/network/wlan0 configured in master mode:

```
#!/bin/bash
set -o pipefail
PATH=/usr/local/sbin:/usr/sbin:/sbin:/usr/local/bin:/usr/bin:/bin

INAME=$(basename $0 |awk -F. '{print $NF}')
```

```
do_start ()
{ echo "Start $INAME : $(date "+%F %H:%M:%S") $$"
  ifconfig ${INAME} up
  sleep 1
  iwconfig ${INAME}
  sleep 1
  ifconfig ${INAME} down
  sleep 1
  hostapd -B /etc/hostapd/${INAME}.conf -e /dev/hwrng
  sleep 1
  ifconfig ${INAME} 0.0.0.0 up
}

do_stop ()
{ echo "Stop $INAME : $(date "+%F %H:%M:%S") $$"
  ifconfig ${INAME} 0.0.0.0 down
  PID=$(ps -eo pid,cmd |grep hostapd |grep  -w "${INAME}\.conf" |awk '{print
$1}')
  [ "$PID" != "" ] && kill -9 $PID
}

case $1 in
  start) [ $(grep -wc "${INAME}:" /proc/net/dev) -gt 0 -a $(ps -ef |grep -v
grep |grep -c "/etc/hostapd/${INAME}.conf") -lt 1 ] && do_start |tee -a
/run/${INAME}.log || exit 1 ;;
  stop) [ $(grep -wc "${INAME}:" /proc/net/dev) -gt 0 ] && do_stop |tee -a
/run/${INAME}.log || exit 1 ;;
  restart)
    if [ $(grep -wc "${INAME}:" /proc/net/dev) -gt 0 ]
    then
      do_stop |tee -a /run/${INAME}.log
      do_start |tee -a /run/${INAME}.log
    else
      exit 1
    fi
  ;;
  status)
    if [ $(grep -wc "${INAME}:" /proc/net/dev) -gt 0 ]
    then
      ip link list $INAME |sed -e "s/^[0-9]*: *//"
      iw dev $INAME info
      echo "Associated stations:"
      iw $INAME station dump | grep -w "^Station"
    else
    else
      echo "$INAME: no sich interface found on system"
      exit 1
    fi
  ;;
```

```
esac
```

Logical devices like bridges would need to be initiated either by telling all the physical device starter scripts which is the parent logical device or by adding them to rc.local. In fact wlan0 is bridged with eth0 … for simplicity I chose not to have daughter devices call parent device init scripts so I start br0 from rc.local but care has to be taken not to start daughter devices if they have already been brought up by udev (note how nothing is done in wlan0 if hostapd is already started).

Here's my br0 script:

```
#!/bin/bash
set -o pipefail
PATH=/usr/local/sbin:/usr/sbin:/sbin:/usr/local/bin:/usr/bin:/bin

INAME=$(basename $0 |awk -F. '{print $NF}')
BR_NICS="eth0 wlan0"

do_start ()
{ echo "Start $INAME : $(date "+%F %H:%M:%S") $$"
  brctl addbr $INAME
  brctl stp $INAME off #turn off spanning tree
  for NIC in $BR_NICS
  do
    [ -x /etc/rc.d/network/$NIC ] && /etc/rc.d/network/$NIC start 2>&1
    sleep 1
    brctl addif $INAME $NIC
  done
  ifconfig $INAME 10.1.0.1 netmask 255.255.254.0 up
  dnsmasq -C /etc/dnsmasq/${INAME}.conf
}

do_stop ()
{ echo "Stop $INAME : $(date "+%F %H:%M:%S") $$"
  PID=$(ps -eo pid,cmd |grep dnsmasq | grep  -w "${INAME}\.conf" |awk
'{print $1}')
  [ "$PID" != "" ] && kill -9 $PID
  ifconfig $INAME 10.1.0.1 down
  for NIC in $BR_NICS
  do
    brctl delif $INAME $NIC
    [ -x /etc/rc.d/network/$NIC ] && /etc/rc.d/network/$NIC stop
  done
  brctl delbr $INAME
}

case $1 in
  start) do_start |tee -a /run/${INAME}.log ;;
  stop) do_stop |tee -a /run/${INAME}.log ;;
  restart) do_stop |tee -a /run/${INAME}.log ; do_start |tee -a
/run/${INAME}.log ;;
  status) brctl show
```

```
    for NIC in $BR_NICS
    do
      [ $(grep -wc "${NIC}:" /proc/net/dev) -gt 0 ] && ifconfig $NIC |grep -
Ew "^${NIC}|ether" || echo "${NIC}: not found"
    done
    ifconfig $INAME
    ps -ef | grep -v grep | grep "dnsmasq/${INAME}.conf"
  ;;
esac
```

If you've gone so far you might want a neater way to deal with unplugging dongles as anything udev could possibly do would be too late. One possible approach is to use gpio pins with buttons attached to them and a script that monitors the button status and takes appropriate action when each button is pressed. I've done this on mi RPi2 with 5 buttons and 5 leds indicating whether the button press has been caught by the monitor script: one shuts down the RPI2 while the other 4 are for deactivating whatever is in the respective usb port to prepare it for unplugging. Here's a picture of such a setup while the script for handling it is here.

8 Setting up on an Embedded Device

Supposing you want all this but you don't want to leave a laptop or desktop on all the time you might want to put in on an embedded ARM system that will only use a fraction of the power required to run an X86 laptop/desktop of any sort. (intel Curie might kick in with a 2.2W x86 SOC when the segfault bug is sorted out). Well the official ARM Slackware port userland runs on almost any ARM device out in the market today. There is a number of machines that are officially supported from the ARM port and a god deal of community supported efforts for the platforms that are not officially supported.

In terms of space you can fit all that's required in 151 Mb with a little careful stripping of unnecessary locales and documents. The best way to go about that is to add to the miniroot all the packages you need and then strip unnecessary stuff. Starting from a full install and the stripping unnecessary stuff will take a bigger effort.

Al the configuration so far will be exactly the same even if you're running the Slackware ARM port (actually since I did most of my testing on an ARM device it's more likely that there be an issue on the x86 platform).

On an embedded device there are however some issues that you need to address:

1. preserving the life of the flash mass storage
2. secure the system against unexpected reboots

1) By default the access time of any file gets automatically updated upon access of any file, this is a bad thing to do on flash that only has a limited number of rewrite cycles (typically 10,000 for MLC and 100,000 for SLC devices). Make sure you mount any filesystems on flash devices with the “noatime” option to prevent the access time modification. This is what the root entry in fstab could look like if you mount it with the noatime option (I use labels buy you can also use UUID or just plainly the device file)

```
LABEL=root      /               ext4        defaults,noatime         1   1
```

There are some generic filesystem type that may do better then other at handling flash devices and some that were specifically designed for that purpose but only run on native flash layer (not an option on usb flash sticks) like cffs2 and ubifs. There are also other options I've just shown 2 common options that allow mounting rw if required.

2) Being on all the time your embedded device will crash every time uncontrollable power failures occur (whether it be a general powe failure, wife using too much power and causing the general switch to trip, your kids play unplug dad's stuff or a hard reset required because your AP crashed). Although journal-ed filesystems are robust with regards to that on the long run bad things may occour. What you really want to eliminate the problem altogether is leave your filesystems on flash read only (ro): Slackware was not conceived to run with root mounted ro but with a little tinkering in the init scripts you can work around that problem. Also having the filesystems ro will be even better then avoiding access time updating. You can remount temporarily with Read Write (rw) any time you need to make permanent changes then remount ro once you've finished. With the system running ro and only temporary files in ram filesystems all that can get lost across reboots are logs and temporary files ... but your system will never suffer catastrophic filesystem failures or prompt for interactive fsck during boot.

Here are some changes I frequently make to /etc/rc.d/rc.S to leave root mounter ro:

- reconfigure all I can to use /run instead of /var
- move /etc/{mtab,resolv.conf,ld.so.cache} in /run/etc/ and make links that point to where I moved them
- move /var/log/{messages,syslog,lastlog,wtmp,btmp,secure,dmesg} in /run/log/ and make links that point to where I moved them
- change the line that mounts root rw “/sbin/mount -w -v -n -o remount /” to something like this “/sbin/mount -v -n -o remount /”
- save the random-seed to unused sector on flash when shutting down (rc.0 ... well it's a link to rc.6) and load it back to /dev/urandom from rc.S when system comes up again

You could optionally have fstab specify that root should be mounted ro like this: (example below is relative to when I was using ubifs on the DocStar)

```
/dev/ubi0_0     /               ubifs   ro                  0       1
```

You will also have to make some links in various places so that they end up writing in /run that has been mounted tmpfs. I do that manually just once with / mounted rw and then remount it ro.

If you're interested in actually making such changes to the init scripts I've shared how I go about it on Linux Questions.

9 Unexpected Trouble Sources

When you start doing your own stuff you may run into problems that are hard to debug because the cause is pretty much unexpected.

Here's an example of one that recently gave me a headache over a long weekend after migrating

from the seagate dockstar to RPi2. I had previously tested rigorously the functionality of everything I had changed in the setup and even back-ported some stuff to the dockstar to make sure it was stable. All seemed fine but when I finally switched to the Pi I started getting plagued with unstable internet connection. I fiddled endlessly with the peer setup without any improvement. I called a friend that uses the same ISP and he told me he had no issues so whatever I was experiencing I was the cause of my own problems. During all my tests on both RPi and RPi2 I had never run into issues and nothing had changed that had not been previously tested, what on earth was going on ? It then dawned on me that I had done all my testing on the new hardware running from a USB power bank and that maybe there was some sort of parasitic coupling between the power supply I was now using and the modem itself. Sure enough the problems dissipated as I powered the PRi2 off the power bank again. I then pulled out a better quality power supply for the Pi and everything still worked fine.

It was that lousy wall charger, that I took off a really low end tablet my son broke, that was making enough noise to bother my modem. Had it been a FCC or CE certified power supply that would probably not have happened.

Obviously there may be innumerable other trouble source you may run into (some of which you have no control … like when your ISP makes clumsy NAT for you) but if you're doing your own stuff be prepared to put up and debug them.

10 Conclusions

You have set up an AP with entry level security and the most common options available in the low end appliances but you have gained:

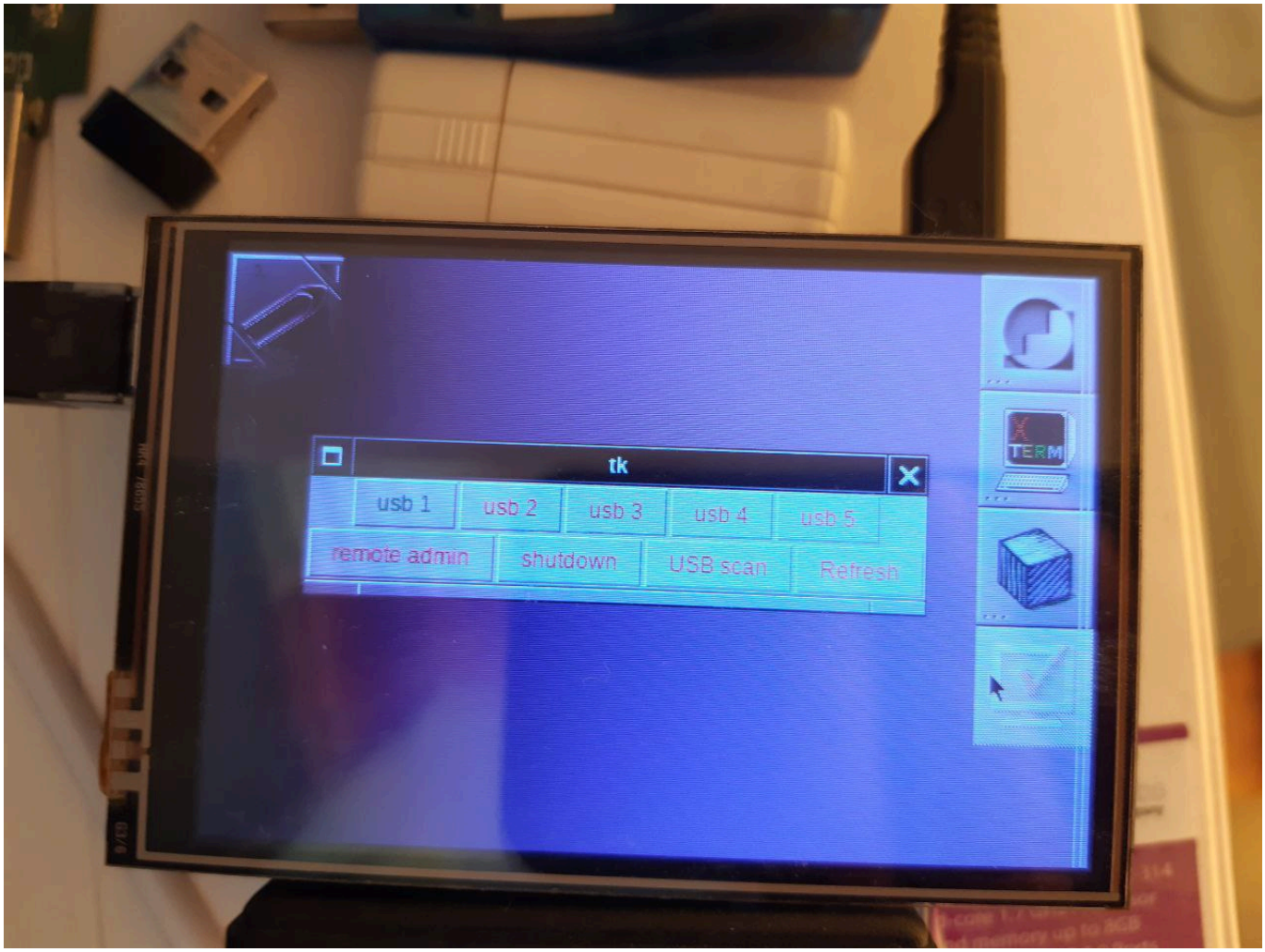
- ability to easily keep your AP software up-to date
- ability to run several AP at the same time (and even mesh into other WiFis … not covered in this howto)
- plug and network capability for the usb dongles
- full control over the way interfaces are bridged and/or the way traffic is routed trough the AP
- full control on the way dhcp works (limitless subnets and address reservation)
- full control over fire-waling (no limits in port forwarding, MAC acl, DMZ ot subnets with specific set of permissions and quotas)
- ability to re-share ANY type of connection the AP has
- ability to choose what and how to backup in the AP
- ssh access for management with a standard Linux environment for easy debugging issues
- scale up/down the hardware that runs the AP without having to change the way you setup things
- update components for which new vulnerabilities have been found without having to wait the the manufacturer does something about it
- whatever the flexibility of a standard Linux environment can give you that a custom crippled embedded environment cannot.

I'm not going to miss my old low end AP, are you ? I replaced it with home-brew stuff from which this article is derived. Over the years I've done several remakes adding some new features. The last upgrade was using a Friendlyelec NanoPi R1S with integrated wifi and 2 ethernet nics requiring no extra usb dongles.

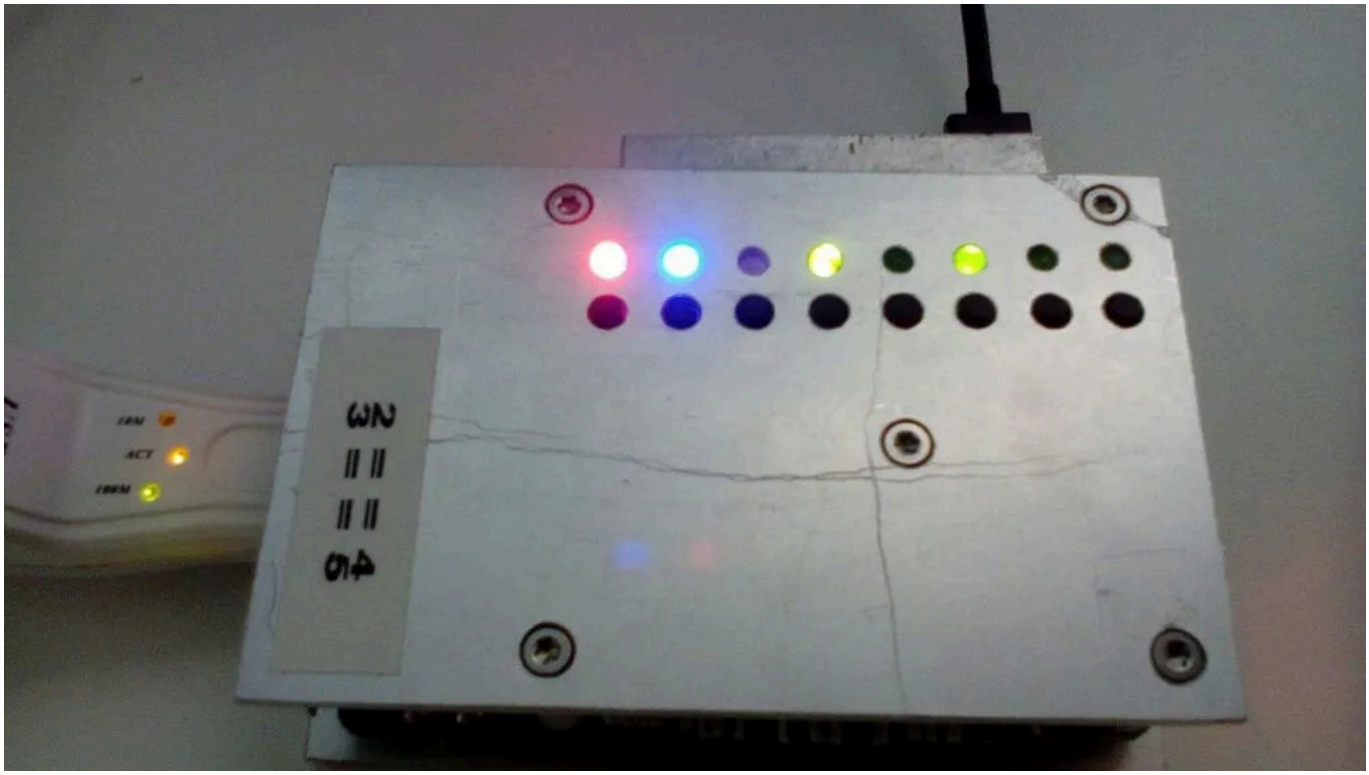
Gen 5 (Friendlyelec NanoPI R1S)

The original case has 2 issues: cappy internal antenna and tendency to overheat ... so I 3d-printed my own case.

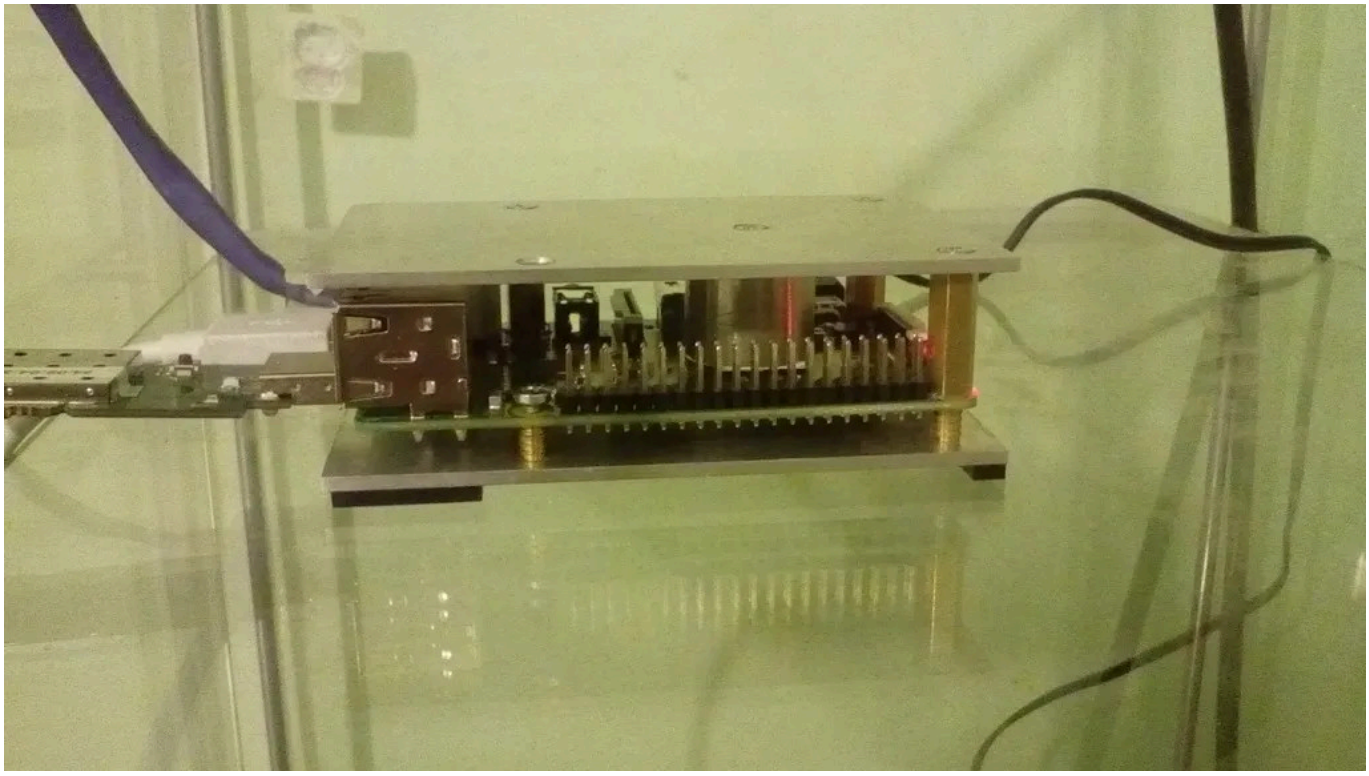
Gen 4 (RPi2 with display)



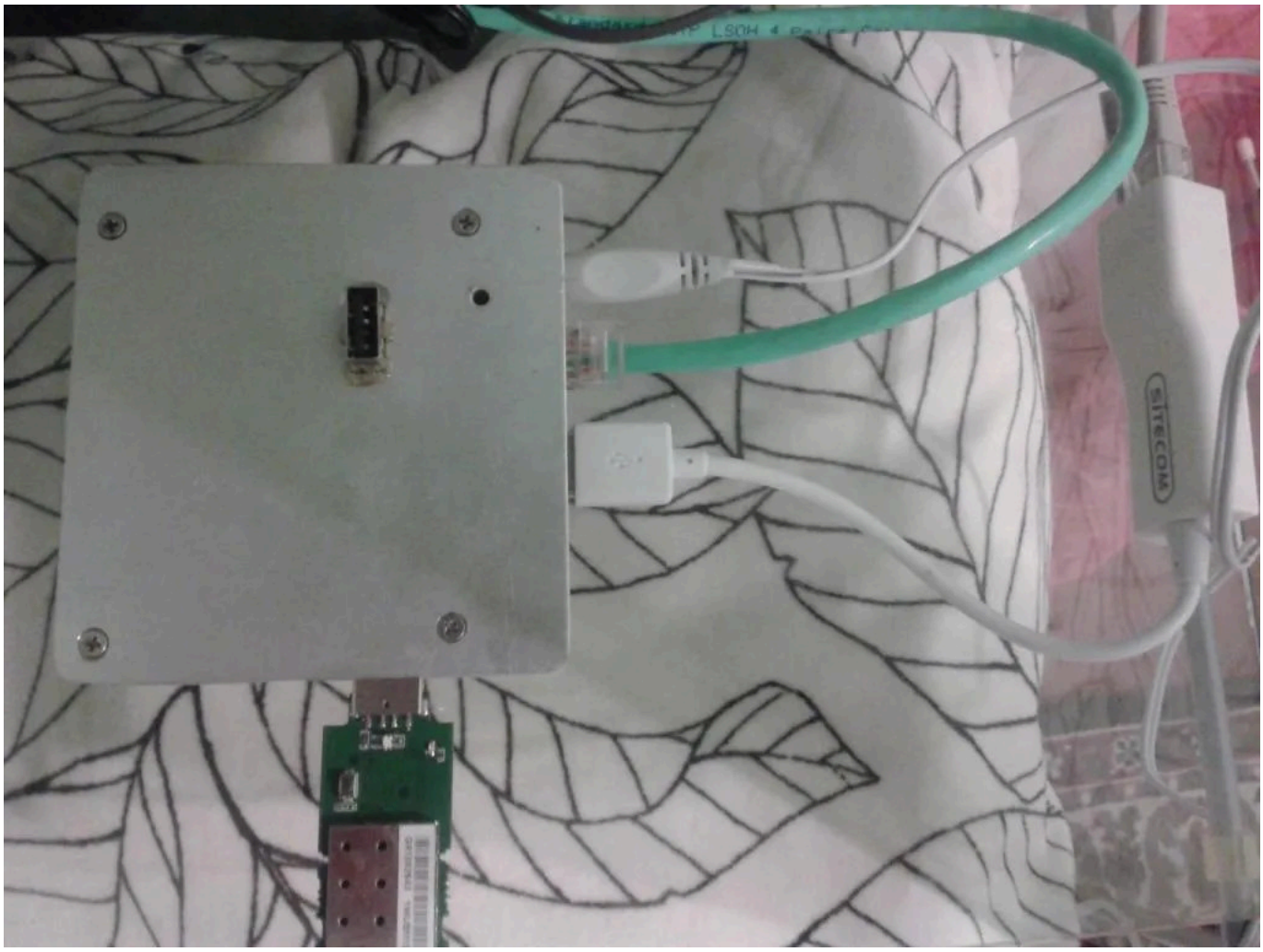
Gen 3 (RPi2 with buttons and leds)



Gen 2 (RPi2 in aluminum Sandwich)



Gen 1 (Seagate DocStar in aluminum Sandwich)



If anyone is interested in a nearly ready to go Raspberry Pi 1/2 image of what has been discussed in this article you cab get if from here. Just extract tar and do “cat rpi_wrap_512mb.img > <your SD device>” into an SD at least 512Mb, boot it and longin via the console, read readme.txt and complete the setup manually. The readme.txt is also in the tarball so you can read it before even logging into the Pi.

Sources

- Originally written by louigi600

howtos, louigi600