UNIVERSITY OF TRENTO
Dipartimento di Ingegneria e Scienza dell'Informazione
Laboratory of Nomadic Communications

A glimpse of the real time 802.11 firmware

1. Tutorial goals

After this tutorial students should have acquired enough skills to

1) understand the code-flow of the real time firmware
2) count, filter, and jam unicast packets

2. Tutorial steps

1) **Code path** The firmware code path is relatively *complex* and a schematic view is illustrated in Figure 1. Given the complexity we will approach it slowly and will start with the reception part, which is easier to understand as the underlying state machine is fairly simple.
   Firmware source The source of the firmware code is located in the folder `/lib/firmware/b43`. The source file is named "`ucode5.asm`" and you can edit it directly using the `vi` "unfriendly" editor. Once you edit the source, simply run "`make`" in the firmware folder and it will be compiled. Any error will be reported.
   Please try to maintain copies of the firmware source inside the /home/wireless folder or, preferably, do backups by copying the firmware source to your laptop. If for any reason the firmware gets corrupted, run the command "`b43_reset.sh`". This will erase the firmware from `/lib/firmware/b43` and replace it with a fresh vanilla copy.

2) **Understanding the rx code path** When a valid packet preamble is detected on the channel, the firmware executes the `rx_plcp` handler to analyze the header and decide how to manage the frame. This is done "as early as possible" when the packet is still being received. The `rx_complete` handler will be executed later, when the packet is received completely. Going back to `rx_plcp`, the firmware first waits for at least 6 bytes of the preamble (PLCP) plus 32 of the MPDU to be received. To see how this is implemented, let us look at the loop `wait_for_header_to_be_received`

```
wait_for_header_to_be_received:
        jext    COND_RX_COMPLETE, header_received
        jl      SPR_RXE_FRAMELEN, 0x026, wait_for_header_to_be_received
header_received:
```

   Instruction `jl` is a conditional jump (jump if less) that loops until register `SPR_RXE_FRAMELEN` is less than 0x26 (6 + 32): that register reports how many bytes of the *current* reception have been already decoded. It is worth noting that to avoid the firmware to stall in case the packet is shorter than 38 bytes, the loop keeps checking if the reception is finished (`jext` is a conditional jump that verifies if the operand-condition is true). Keep in mind that this type of loop is used many times in the firmware!
   The initial part of the packet is also copied in the shared memory starting from the address 0xA08. This is exploited by the firmware for checking the received packet type (`RX_TYPE`) and subtype (`RX_TYPE_SUBTYPE`) and taking further decisions. For instance if `rx_plcp` detects a frame is of management or control type, then it waits for complete reception

(check the loop `rx_plcp_not_data_frame`). If an acknowledgment is detected, then it jumps to the handler `rx_ack`, which, among other actions, clears the transmission timeout. If instead it detects a data packet, it jumps to `rx_data_plus`, where it checks if the packet contains at least 22 bytes (0x1c in hexadecimal):

```
rx_data_plus:
        jext    COND_RX_COMPLETE, end_rx_data_plus
        jl      SPR_RXE_FRAMELEN, 0x01C, rx_data_plus
end_rx_data_plus:
        jl      SPR_RXE_FRAMELEN, 0x01C, rx_check_promisc
        jnext   COND_RX_RAMATCH, rx_ra_dont_match
        jext    COND_TRUE, send_response
```

Questions:

a. Why did we say 22 bytes if the firmware checks for at least 28 (0x1c)?
b. Why is it important to have 22 bytes? How many fields can we find inside?
c. Do you have any ideas about why a frame is considered "suspect" and is handled by the "`rx_check_promisc`" handler (hinting at a sniffer-only receiver), if the frame is short?

If the destination address matches that of the station executing the firmware, then the firmware jumps to `send_response`, where an acknowledgment frame is prepared WITHOUT actually being scheduled. The scheduling decision must be taken afterwards, when the frame is completely received. Besides, `send_response` "remembers" through the state machine register (`SPR_BRC` represents the state of the MAC algorithm) that the frame needs a response (`NEED_RESPONSEFR`), while it cleans the condition that could trigger the transmission of a beacon or a probe response (`NEED_BEACON`, `NEED_PROBE_RESP`)

```
        orxh    NEED_RESPONSEFR,
                SPR_BRC & ~ (NEED_BEACON|NEED_RESPONSEFR|NEED_PROBE_RESP),
                SPR_BRC
```

Here the instruction `orxh` performs a special logical "or" between operand 1 and 2 and stores the result in operand 3. Pay attention, as this is a special instruction, meaning that only some combinations of the bits to clean/set are allowed.

Question:

d. Why must the decision of whether to schedule an ACK or not should be made when the frame is completely received?

For all the cases we considered above (management, control and data), the reception process needs a final handler `rx_complete`, that is executed when the time reserved for the MPDU expires. Based on previous decisions, this handler can schedule the transmission of the ACK, e.g.

```
        jnext   COND_NEED_RESPONSEFR, check_frame_subtype
need_regular_ack:
```

The condition `COND_NEED_RESPONSEFR` is true if the state machine (`SPR_BRC`) was previously programmed by `send_response`. In this case `need_regular_ack` sets up the modulation type and MCS of the ACK frames that will be scheduled according to the incoming frame properties. If instead no ACK frame was prepared (e.g. upon the reception of a multicast frame) then it jumps to `check_frame_subtype` and no ACK is scheduled. ACK scheduling happens below, when the transmission engine is loaded with one of the possible schedule control keys, in this case:

```
or      NEXT_TXE0_CTL, 0x000, SPR_TXE0_CTL
```

where the keyword `NEXT_TXE0_CTL` was previously set up by `send_response`.

3) **Receiving packets** In this exercise we will start practicing with the firmware and we will count incoming packets that satisfy some rules. Mastering this is important to be able to program custom features on the firmware, as the incremental changes required for implementing new MAC algorithms should be executed only for *some traffic* and not for all frames. Thus we will start with counting UDP packets sent to a given port.
The initial part of each packet (configurable length) is copied to the shared memory starting from the address 0xA08 (configurable address). To better understand how an UDP packet appears inside the shared memory, use iperf to generate greedy traffic from the AP to port 3000 of a STA (for selecting such port add "`-p 3000`" to the command lines on both instances of iperf, i.e. on the client and on the server). Then, as long as the traffic is flowing, run this command on the receiver

```
$: readshm -s
```

This will display the entire contents of the shared memory. Scroll down and start the analysis from the address `0xA00`. You should clearly see a `0x45` byte somewhere, preceded by the LLC header `0xAA 0xAA 0x03 0x00 0x00 0x00 0x08 0x00`. If you do not see this, reissue the command above.

   a. Where is it more appropriate to filter *data frames*? One good point is in `rx_data_plus` if we are interested in taking some special decision for packets *as they are being received*. In this case, the minimum number of bytes to wait must be increased to include the initial 6 bytes PLCP, the entire MAC header, IP and UDP ones (at least the destination port). Another good point is in `rx_complete`, e.g. before/after checking that the packet was received without errors.

In the following we will use a couple of nodes, an AP and a STA. The counter will be set up at the receiving node, so let's start with `rx_data_plus`.

4) **CPU and memory access basics** The shared memory is accessed as a 16-bit memory. For instance to copy the content from (per byte) address `0xFF0`, inside register `r63`, use

```
mov     [SHM(0xFF0)], r63
```

Here the brackets mean "access memory directly", while `SHM()` is just a macro that divides the argument by two. The assignment with `mov` is left to right, i.e. it will copy the 16-bit value at byte addresses `0xFF0` and `0xFF1` into register `r63`. As the CPU is little endian, byte at `0xFF0` will go into the Least Significan Byte of `r63`, while byte at `0xFF1` will go into the Most Significant Byte. Remember that all registers from `r0` to `r63` are 16-bit, and that only upper registers (i.e. from `r46` to `r63`) are free. All the others should not be changed, as the MAC state machine relies heavily on their values.
While the `mov` instruction allows direct assignment of 16-bit constants into a register or memory, operations including arithmetic, logic and conditional jumps with direct operands allow only constants in the range `[0, 0x1ff]`. In particular, these are valid

```
mov     0xdead, r62                 // r62 <= 0xdead
mov     0xbeef, r61                 // r61 <= 0xbeef
add     r61, 0x1f0, [SHM(0xFF0])    // [SHM(0xFF0)] <= value(r61) + 0x1f0
and     r59, 0xff, r63              // r63 <= value(r59) & 0xff
je      r60, 0x45, action1          // if value(r60) == 0x45, jump to action1
sr      [SHM(0xFF0)], 8, r63        // r63 <= [SHM(0xFF0)] >> 8
sub     r60, 1, r60                 // r60 <= value(r60) - 1
```

while the following are not:

```
add     r61, 0x4000, r63
je      r60, 0x1234, action2
```

To achieve the intended results with the correct code, use the following:

```
mov     0x4000, r63
add     r61, r63, r63           // r63 <= value(r61) + 0x4000

mov     0x1234, r63
je      r60, r63, action2       // if value(r60) == 0x1234, jump to action2
```

5) **Counting packets** Examine again the `rx_data_plus` handler:

```
rx_data_plus:
        jext    COND_RX_COMPLETE, end_rx_data_plus
        jl      SPR_RXE_FRAMELEN, 0x01C, rx_data_plus
end_rx_data_plus:
        jl      SPR_RXE_FRAMELEN, 0x01C, rx_check_promisc
        jnext   COND_RX_RAMATCH, rx_ra_dont_match
        jext    COND_TRUE, send_response
```

Place your filtering instructions before the last two statements. The rules should be: packet is IP (first byte of the MPDU payload is 0x45), IP type should be the UDP protocol (check the "proto" field in the IP header), and the destination UDP port should be 3000 (check the UDP header for the destination port field). As the filtering code should verify three conditions, if any of them is not verified then jump directly to send_response, e.g. to verify that the first byte of the MAC payload is the first byte of an IP packet (`0x45`):

```
        and     [SHM(0xA2E)], 0xFF, r63   // see the question below
```

```
jne     r63, 0x45, send_response // jump if not equal
…other filters…
…increment some register…
jext    COND_TRUE, send_response
```

a. Why do we compare the LSB of a 16-bit value at address [SHM(0xA2E)] to 0x45? (This is done by the logical *and* operation) Why not comparing the MSB instead? Remember that the CPU is little endian.
b. How could we count packets that match all the filtering? Try incrementing a free register that is not in use, for instance `r60`.

Now start sending traffic from the AP to the STA, to port 3000. You can see the value of `r60` increasing by issuing the command "`readshm`" and checking that the register's content is increasing! (Running the command without "`-s`" avoids displaying the entire content of the shared memory).

6) **Counting more than 65535 packets** As registers (or single 16-bit values in the shared memory) are *only* 16-bit wide, they allow for counting up to 2^16-1=65535 *events*. To count more than that, we should use a couple of registers/values in the shared memory and the carry when we perform summation, e.g. if we plan to use `r61` and `r60` respectively for the MSW and LSW of a 32-bit counter, we should implement

```
add.    r60, 1, r60
addc    r61, 0, r61
```

where the first addition uses "`.`", which indicates to remember the carry (if any), while the second addition "`addc`" adds the two operands and the carry and stores the result into the third operand (pay attention that the second operand is *zero*).

7) **Counting the (likely) total number of packets vs correct packets** If we count packets in `rx_data_plus`, we count *all* packets that satisfy the rules, including also those that might collide afterwards. If instead we want to count only correctly decoded packets, we should place the same filtering instructions in `rx_complete` after evaluating the condition on *successful packet*:

```
frame_successfully_received:
        jext    COND_RX_FIFOFULL, rx_fifo_overflow
        jnext   COND_NEED_RESPONSEFR, check_frame_subtype
```

Filtering instructions should be placed between the last two statements. Try adding again the same filters, this time of course using different register(s) to store the results, and then run iperf.

a. Why is the number of packets counted in `rx_data_plus` greater than that counted in `rx_complete`?
b. What kind of information can you obtain by computing their ratio?
c. Try finding out a relation between their ratio and the values in the `rc_stats` file on the `debugfs` of the sender.

8) **Jamming packets** Jamming some traffic means disturbing the communication between a couple of peers that are actually exchanging traffic. An easy way to perform jamming is to set up a third node to reply with an acknowledgment as if it were the recipient of the traffic. This will cause a collision between the correct acknowledgment and that generated by the jammer, and consequently the transmitter will not receive and ACK frame, which will lead to contention with longer backoffs. To implement such jamming, we need to change the `rx_data_plus` code of the jammer so that the firmware will prepare a valid acknowledgment, then we need to also change the `rx_complete` code so that it also *schedules* the transmission of the ack frame. In particular, we should jump to `send_response` even if the receiver address does not match (of course it cannot, as the jammer has a MAC address that is different to that of the intended receiver).

```
end_rx_data_plus:
        jl      SPR_RXE_FRAMELEN, 0x01C, rx_check_promisc
        // put your filter here: at first non match, jump to skip_filter
        jext    COND_TRUE, send_response
skip_filter:
        jnext   COND_RX_RAMATCH, rx_ra_dont_match
        jext    COND_TRUE, send_response
```

We should then change `rx_complete`, by adding a similar filter, e.g.

```
frame_successfully_received:
        jext    COND_RX_FIFOFULL, rx_fifo_overflow
        // put your filter here: at first non match, jump to skip_filter2
        jext    COND_TRUE, need_regular_ack
skip_filter2:
        jnext   COND_NEED_RESPONSEFR, check_frame_subtype
need_regular_ack:
```

Now try to run an `iperf` session between the AP and the STA and check the throughput. Then bring the jammer up and connect it to the same AP.

a. Is there any difference in performance, with or without the jammer running?

9) **Improving the jammer** As the ACK frames generated by the jammer and the intended recipient are equal, this would not tamper with the traffic session too seriously. To perturb the network significantly, we have two options:

1. Slightly change the content of the ACK sent by the jammer on the fly;
2. Start transmitting the fake ACK immediately, instead of after a SIFS.

With regard to the first possibility we should take a look at the handler `send_response`. This is using the *Transmission and Modify Engine* (TXME) to compose the ACK frame on the fly, by copying the transmitter address from the received frame into the receiver address of the ACK. There are two interesting points here:

a. For retrieving the transmitter address from the received packet, a sort of indirect memory access is used

```
or    [RX_FRAME_ADDR2_1,off1], 0x000, SPR_TME_VAL10
```

This involves using the offset register `off1`, always initialized to `SHM(0xA08)`. This addressing type allows to also specify an offset with respect to the base value stored in the offset register, in this case `RX_FRAME_ADDR2_1`

i. Check in the include files (.inc) if this offset actually corresponds to where you expect to find the first two bytes of the transmitter address

b. For storing the value into the destination address of the ACK, the TXME is used. This allows changing the first 64 bytes of any outgoing packet on the fly, by simply referring to registers such as `SPR_TME_VALXY` where XY can be an even number in the range [0, 62]. In this case `SPR_TME_VAL10` means the bytes 10 and 11 of the outgoing ACK

i. Why not overwriting bytes 4 and 5 (and the following)? Remember the first 6 bytes of every packet. The conclusion holds for both received and transmitted packets.

For the purpose of jamming, it is enough to replace the value in `SPR_TME_VAL10` with something like 0xdead or 0xbeef. Try this, recompile and check if the approach enhances the impact of jamming.

With regard to the second possibility we have to change the scheduling keyword in `rx_complete`, that is

```
or    NEXT_TXE0_CTL, 0x000, SPR_TXE0_CTL
```

Here the keyword is that chosen inside `send_response` and stored inside `NEXT_TXE0_CTL`, that is 0x4021, which means "schedule 10us after the completion of the current reception, if it is a valid packet". To schedule an immediate transmission, replace the `NEXT_TXE0_CTL` variable (this is a register) with 0x4007, which means "schedule immediately". This will effectively force the radio to start transmitting immediately.

a. Try the new jammer and check if it improves. Pay attention, it could crash ☺

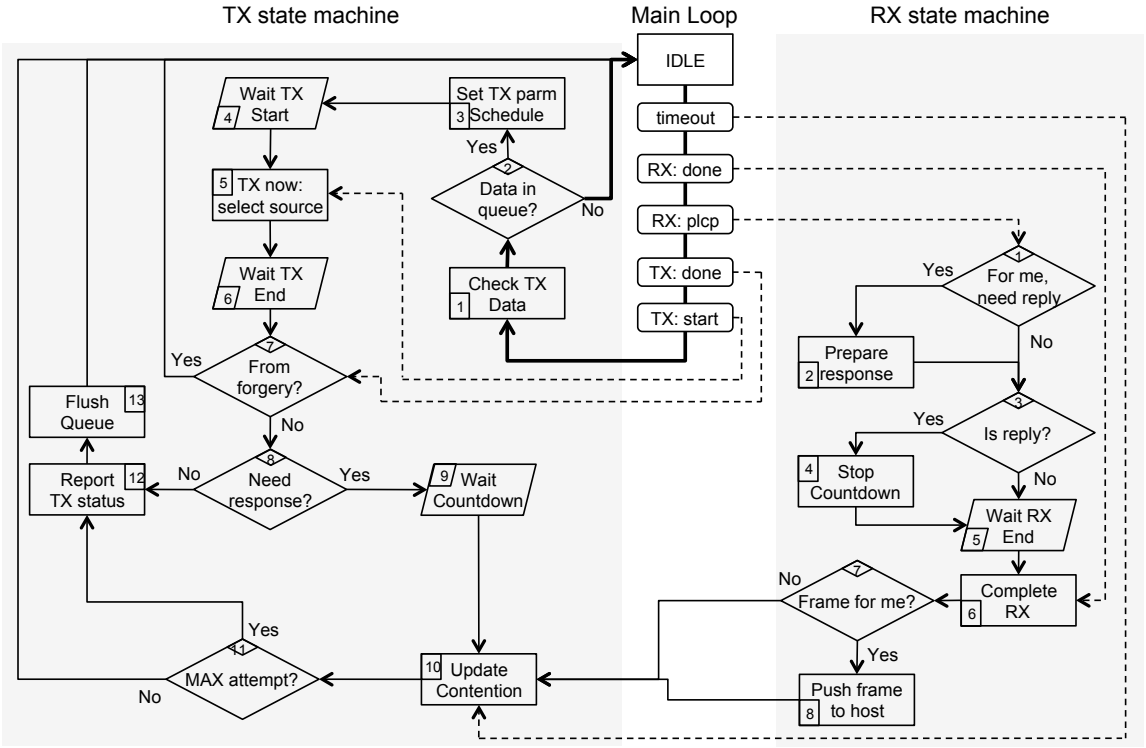

***Figure 1** - Main code blocks of the real time firmware*