A glimpse of the real time 802.11 firmware

1. Tutorial goals

After this tutorial students should have acquired enough skills to

1) understand 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 "complex" and it is reported in Figure 1. Given this complexity we have to approach it slowly and we will start with the reception part, which is easier to understand as the underlying state machine is simple.

2) **Understanding the rx code path** When a valid packet preamble is detected on the channel, the firmware executes handler `rx_plcp` 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: independently of the decision, in fact, handler `rx_complete` will be executed later, when the packet is received completely. Going back to `rx_plcp`, the firmware first waits at least 6 bytes of the preamble (plcp) plus 32 of the MPDU are received: check the loop in `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 `SPR_RXE_FRAMELEN` is less than 0x26 (6 + 32). 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 verify if the operand-condition is true). Remember 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 location 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 frame is management or control, then it waits for completely receiving them (check the loop `rx_plcp_not_data_frame`): if an acknowledgment is detected, then it jumps to handler `rx_ack`, which, among others actions, cleans the transmission timeout. If instead it detects a data packet, it jumps to `rx_data_plus`, that checks if the packet contains at least 22 bytes

```
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 we said 22 bytes if it checks for at least 28 (0x1c)?
b. Why it is important having 22 bytes? How many fields we can find inside?
c. Ideas about why if the frame is shorter then it's a "suspect one" and it should be handled by this "`rx_check_promisc`" handler whose name recalls that of a sniffer-only receiver?

If the receiver address matches the one of the station that is executing the firmware, then the firmware jumps to `send_response` that prepares the acknowledgment frame WITHOUT actually scheduling it, as this decision must be taken afterwards, when the frame is completely received. Besides, `send_response` "remembers" in the state machine register (`SPR_BRC` represents the state of the MAC algorithm) that the frame needs a response (`NEED_RESPONSEFR`), while cleaning the condition which 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 instruction `orxh` performs a special logical "or" between operand 1 and 2 and put the result in operand 3. Pay attention this is special which means that only some combination of the bits to clean/set are allowed.

Question:

d. Why whether scheduling the ack or not must be decided when the frame is completely received?

For all the cases we considered (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., on reception of a multicast frame) then it jumps to `check_frame_subtype` and no ack is scheduled. Ack schedule 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. This ability is important for programming the firmware as the incremental changes for implementing new MAC algorithms should be executed only for *some traffic* and not for all frames. We will start with counting UDP packet at some port.
The initial part of each packet (configurable length) is copied in shared memory starting from address 0xA08 (configurable address). To better understand how a 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 of both instances of iperf on client and on server). Then, as long as traffic is flowing, run this command on the receiver

```
shmread -s
```

This will display the entire content of the shared memory: scroll down and start the analysis from 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 it, reissue the command again.

a. Where is it better 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 in the receiving node, let's start with `rx_data_plus`.

4) **CPU and memory access basics** Shared memory is accessed as a 16-bit memory, e.g., to copy the content at (per byte) address `0xFF0` inside register `r63`, use

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

Here brackets mean "access memory directly", while `SHM()` is just a macro that divides the argument by two. Assignment with `mov` is left to right: it will copy the 16-bit value at byte addresses `0xFF0` and `0xFF1` into register `r63`. As the CPU is little endian, byte in `0xFF0` will go into the LSB of `r63`, while byte in `0xFF1` will go into MSB. Remember that all registers from r0 to r63 are 16-bit, and that only upper registers (e.g., from r46 to r63) are free: all the others should not be changed as the MAC state machine deeply relies on their values.

While `mov` instruction allows direct assignment of 16-bit constant into a register or memory, operations including arithmetic, logic and conditional jumps with direct operands allows 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 these are not:

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

To achieve the same effects with correct code do this:

```
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) **Count 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 between the last two statements. Rules should be: packet is IP (first byte of the MPDU payload is 0x45), IP type should be protocol UDP (check the IP header for field "proto"), and destination UDP port should be 3000 (check UDP header for 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 we compare the LSB of 16-bit value at address [SHM(0xA2E)] to 0x45? (This is done by the logical *and* operation) Why instead not comparing the MSB? Remember that CPU is little endian.
b. How to count packets that pass all the filtering? Try incrementing a free register you do not use like `r60`.

Start now sending traffic from AP to STA to port 3000. You can see the value of `r60` increasing by issuing command "`shmread`" and check that register is increasing! (Running the command without "`-s`" avoid displaying the entire content of the shared memory).

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

```
add.    r60, 1, r60
```

```
        addc    r61, 0, r61
```

where the first addition use "`.`" that means to remember the carry (if any), while second addition "`addc`" adds the two operands and the carry and store the results into the third operand (pay attention second operand is *zero*).

7) **Counting (likely) total packet 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 correct packet we should place the same filter 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 use different register(s) to store the results, then run iperf.

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

8) **Jamming packets** Jamming some traffic means that we are going to disturb the communication between the couple of peers that are actually exchanging such traffic. An easy way to do this 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 of the correct acknowledgment with that generated by the jammer, so that the transmitter will not receive and ack frame and will start deferring with longer backoffs. To this end we need to change the `rx_data_plus` code of the jammer so that the firmware of the jammer 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 can not, as the jammer has a different mac address than 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:
```

Try running now 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. Does make any difference whether or not the jammer is running?

9) **Improve the jammer** As the ack frames generated by the jammer and the intended recipient are equals, this could not puzzle up the traffic session too much. To make a mess instead we have two options:

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

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

a. For picking up 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 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 correspond to where you expect to find the first two bytes of the transmitter address

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

i. Why not overwriting bytes 4 and 5 (and following)? Remember about 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, recompile and check if this improves 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 after 10us since the conclusion of the current reception if it is a good packet". To schedule an immediate transmission replace `NEXT_TXE0_CTL` variable (it's a register) with 0x4007, which means "schedule immediately": this will force the radio to start transmitting immediately.

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

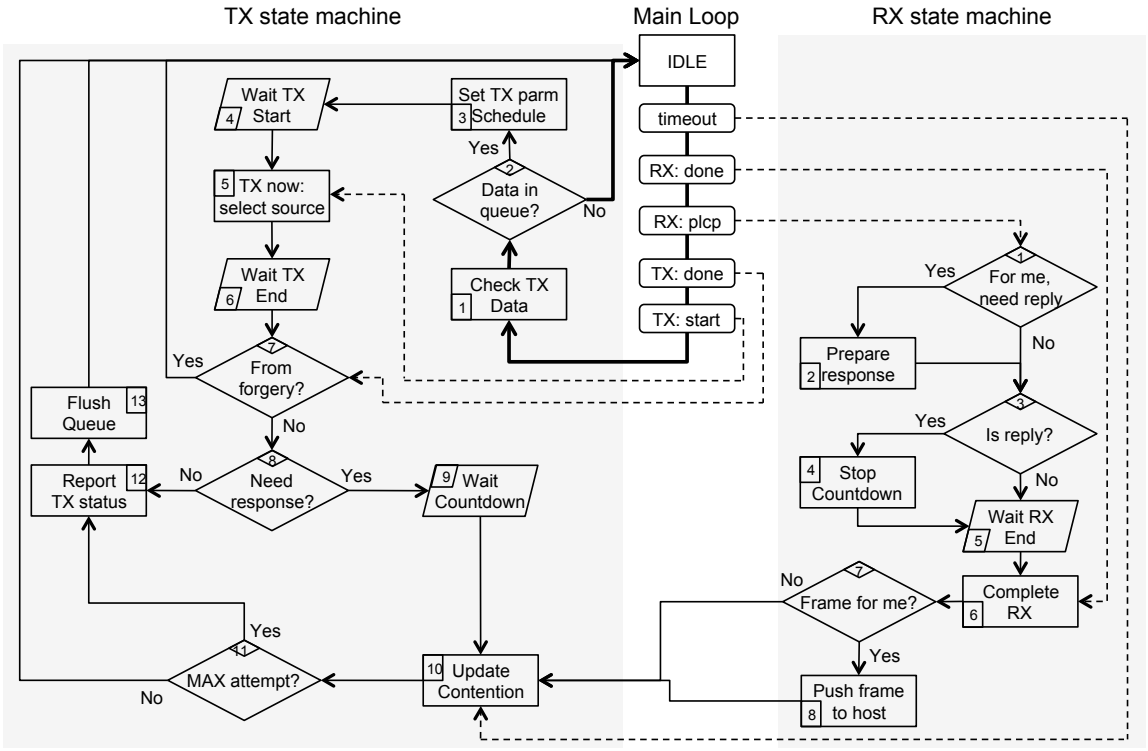

Figure 1 - *Main code blocks of the real time firmware*