Exercise: playing with transmission engine, Contention Windows (CW) and Transmit & Modify Engine

1. Exercise goals

After this exercise students should understand the role of the Contention Window parameters in channel access and should be able to forge specific packets.

2. Tutorial steps

1) **Contention Window Parameters** In the open firmware only one queue can be used at the same time, there is not yet support for Quality of Service. For this reason it is really easy to change the firmware behavior in a few steps. First of all there are two boundaries (min-max) for the CW, namely

   a. `MIN_CONTENTION_WIN:` value assigned to CW on the first transmission attempt;
   b. `MAX_CONTENTION_WIN:` the maximum after which CW is kept constant.

   As the kernel keeps resetting these parameters when the interface is working, it is mandatory to reload them in the main loop to override the kernel assignments, e.g., by adding the following two instructions right after the `state_machine_start` label for fixing CW boundaries to specific values that we store in the shared memory:

```
state_machine_start:
       mov     [SHM(0xFF0)], MIN_CONTENTION_WIN
       mov     [SHM(0xFF2)], MAX_CONTENTION_WIN
```

   Pay attention however that free locations in shared memory (like those used here above) are initially set to zero, this could lead to problems if the two lines are used *as is*: it is better to check if the two locations are different than zero before loading the values, e.g.:

```
state_machine_start:
       je      [SHM(0xFF0)], 0, skip_min_win
       mov     [SHM(0xFF0)], MIN_CONTENTION_WIN
skip_min_win:
       je      [SHM(0xFF2)], 0, skip_max_win
       mov     [SHM(0xFF2)], MAX_CONTENTION_WIN
skip_max_win:
```

   It is now possible to play with the tool `writeshm` to change the two locations but remember that the values written in shared memory must be nibble-swapped. Try writing a new value to `[SHM(0xFF0)]` and verify that it is correctly loaded in register `MIN_CONTENTION_WIN(=r3)` using `readshm`. Remember also that the window parameters must be power-of-two minus `1`, so `0x1F` is ok, while `0x23` is not. Try now the following:

```
$: writeshm /sys/kernel/debug/b43/phyN 0xFF0 0x7f00
```


Copyright © 2017 F. Gringoli <francesco.gringoli@unibs.it>, all rights reserved

and verify it worked with `readshm`.

2) **Playing with contention windows parameters** To test how these parameters affect the behavior of the transmission try running two iperf sessions from two stations to the AP. Fix the same data rate so that the rate controller could not affect the experiment: to this end use `iwconfig` tool:

```
$: iwconfig wlan0 rate 6M
```

After verifying that the two sessions almost share the same throughput, try lowering the minimum contention window of only one station: what happens? And what if instead only the maximum value is lowered, keeping the same minimums?

Finally, reset the contention window parameters to the initial values (`0x1F` and `0x3FF`) and fix with `iwconfig` two different data-rate (e.g., 6M and 12M). Run two iperf sessions again: what happens? Is it possible to play with the contention window parameter to *force* the same throughput share?

3) **Transmission & Modify Engine (TME)** There are two main ways for transmitting a frame: i) from one of the FIFO queues, in this case the kernel code provides all the necessary information to the firmware for preparing the transmission (this is managed by handler `check_tx_data_with_disabled_engine`); or ii) the firmware can transmit frames whose content is drawn from a special memory called `TEMPLATE RAM` like in the case of the acknowledgments and beacons. We will see now the basic steps for scheduling a packet:

   a. Choose the encoding, output antenna(s), power level (does not work on all devices) by writing the specific values into `SPR_TXE0_PHY_CTL` (meaning of all bits at http://bcm-v4.sipsolutions.net/802.11/TX). For this tutorial it is enough to write `0xFC00` for DSSS/CCK, `0xFC01` for OFDM;

   b. Load the value of the backoff counter in register `SPR_IFS_BKOFFDELAY`;

   c. Modify on the fly the first 64 bytes if needed using the Transmission & Modify Engine (see below);

   d. Scheduling the packet for transmission after a given delay by writing a proper value in register `SPR_TXE0_CTL`; there are many combinations for this register (details at http://bcm-v4.sipsolutions.net/802.11/Registers) but for the rest of the tutorial we will consider the scheduling of an ack frame so that we can easily understand how to compose packet payload by taking a look to the `send_response` handler;

   e. Once the packet is scheduled we have to wait for the transmission to start, that is signaled by the activation of the condition `COND_TX_NOW`, that makes the firmware jump to handler `tx_frame_now` where the transmission is finalized: in case we want

Copyright © 2017 F. Gringoli <francesco.gringoli@unibs.it>, all rights reserved

to transmit a packet from the `TEMPLATE RAM` we have to use the same syntax used for the transmission of an acknowledgment that is right below the label `dont_update_preamble`.

We will now use what we learnt from the previous tutorial for implementing a system that transmits an arbitrary frame with chosen payload and MCS when we receive a specific packet (e.g., UDP to port `0xbeef`). Add the filtering instruction in `rx_data_plus` and when the content is detected instead of jumping to `send_response` jump to our own `send_fake_response` where we will compose the frame. In the following we imagine that the packet is `SIG_LENGTH` bytes long. Then to define this new handler we have to:

a. choose the encoding, e.g., OFDM, then write `0xFC01` to `SPR_TXE0_PHY_CTL`

b. remember in the state machine that we are going to transmit a (fake) response at the end of the current reception:

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

c. create a valid PLCP, for the OFDM case this requires to specify in the first 16-bit word of the PLCP the logical or of the number of *octets* shift left five times with the rate code type, where the rate code is (`0xB, 0xF, 0xA, 0xE, 0x9, 0xD, 0x8, 0xC`) for the corresponding (6, 9, 12, 18, 24, 36, 48, 54)Mb/s rate; the resulting value should be loaded in the first register of the TME, e.g.,

```
mov     LENGTH_RATE_KEYWORD, SPR_TME_VAL0
mov     0xFFFF, SPR_TME_MASK0
mov     0, SPR_TME_VAL2
mov     0xFFFF, SPR_TME_MASK2
```

Here the first assignment overrides the first two bytes of the outgoing packet, independently it is being transmitted from a FIFO or from the `TEMPLATE RAM`. To finally modify only *specific* bit, however, we can use the corresponding mask, in this case we want to change all 16 bits so we write `0xFFFF`. The second assignment is necessary to avoid firmware crashes: remember this rule for the next tutorial!!

For the DSSS/CCK case we have to load into the first two bytes of the PLCP the DSSS/CCK rate code (e.g., for 1Mb/s it is `0x040A`), then load into bytes 2 and 3 the number of microseconds taken by the transmission of the frame payload, in this case it is `SIG_LENGTH * 8` (1 bit per microsecond) hence

```
mov     RATE_CODE, SPR_TME_VAL0
mov     0xFFFF, SPR_TME_MASK0
mov     MICROSECONDS, SPR_TME_VAL2
mov     0xFFFF, SPR_TME_MASK2
```


Copyright © 2017 F. Gringoli <francesco.gringoli@unibs.it>, all rights reserved

Note that while for bytes 0 and 1 we use `VAL0/MASK0`, for bytes 2 and 3 we use `VAL2/MASK2`.

Question:

i. What should we use to change only byte 7? (and not byte 6?)

d. remember somewhere, e.g., into a new variable `tx_fake_response` that we will transmit our *fake* response and not the legacy acknowledgment: this will be important in `tx_frame_now` for finalizing the transmission.

e. compose the packet: take into account that with this approach we can specify (on the fly) only the first 64 bytes (using `SPR_TME_VAL0` to `SPR_TME_VAL62` and `SPR_TME_MASK0` to `SPR_TME_MASK62`), but as the first 6 bytes are for the PLCP we can change only the first 58 bytes of the frame payload.

Decide the frame control and write it using the correct `TME` registers: pay attention to avoid using invalid values otherwise the sniffer will not show such frames.

Exercise:

i. Try to send a short data frame to a valid active station, we will see the ack coming back from it!

f. finally remember to load into `NEXT_TXE0_CTL` the value that `rx_complete` will use for scheduling the fake response, to to conclude the handler, e.g., copying what done by `send_response` do

```
mov     0x4021, NEXT_TXE0_CTL
jext    COND_RX_COMPLETE, rx_complete
jext    COND_TRUE, state_machine_idle
```

g. now we customize the `tx_frame_now` handler. As we are changing basically the way we send an ack, we can simply focus on the code right after label `dont_update_preamble`, where we can check if we are transmitting a real ack or the *fake* response by controlling the value of variable `tx_fake_response`, e.g.,

```
dont_update_preamble:
    jne     tx_fake_response, 0, finalize_fake_response

[    cut ack code    ]

finalize_fake_response:
    mov     0xB, SPR_TXE0_WM0
    mov     0, SPR_TXE0_SELECT
    mov     0, SPR_TXE0_Template_TX_Pointer
    add     SIG_LENGTH, 2, SPR_TXE0_TX_COUNT
    mov     0x826, SPR_TXE0_SELECT
    mov     0, tx_fake_response
    add     TX_COUNTER, 1, TX_COUNTER
    jext    COND_TRUE, complete_tx
```


Copyright © 2017 F. Gringoli <francesco.gringoli@unibs.it>, all rights reserved

The first line is really important: to modify bytes on the fly with the `TME` it is not enough what we did so far, we have also to specify in the couple of registers `SPR_TXE0_WM0` and `SPR_TXE0_WM1` which words of 16 bit should be actually modified: consider the two registers as bitmasks. In this case we want to change byte 0-1, byte 2-3 (PLCP) and byte 6-7 (Frame Control), so the bitmask will be (last four bits of `SPR_TXE0_WM0`) `1011` which is exactly `0xB`.

Second line specifies the transmission will happens from `TEMPLATE RAM` (0 in `SPR_TXE0_SELECT`), third line sets the start address in the `TEMPLATE RAM`, fourth requires `SIG_LENGTH + 2` bytes are transmitted (this is because in total we should have `SIG_LENGTH + 6` considering the first six bytes of the PLCP but the TX Engine will add the FCS automatically at the end, so we should ask for `SIG_LENGTH + 6 – 4` that is exactly what written ☺). Then we reset the `tx_fake_response` variable to zero and we store somewhere in the shared memory how many fake response we sent so far (choose a shared memory address for `TX_COUNTER`).

f. REMEMBER also to reset the `tx_fake_response` variable to zero in rx_plcp !! This is really important to avoid unexpected behaviors.

4) **Experiment** Compile the firmware, associate to an AP, then from the AP use iperf to send a session to the chosen port, use a sniffer to confirm that the packet we forged is actually transmitted.

Copyright © 2017 F. Gringoli <francesco.gringoli@unibs.it>, all rights reserved