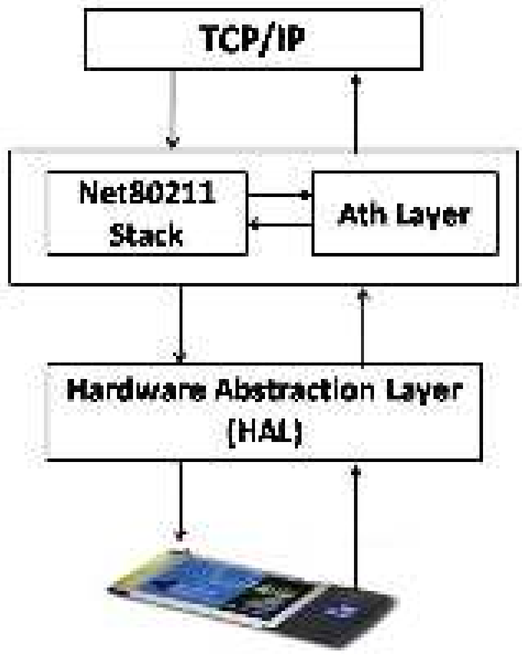
(Atheros) Wireless in FreeBSD

Adrian Chadd
<adrian@freebsd.org>

Overview

- Basic wireless infrastructure overview
- How QoS is handled
- How the hardware is setup (in general, to do QoS)
- How the hardware implements 802.11
- Example: TDMA
- Transmission: Overheads, Bursting, Aggregation
- Transmitting and Receiving Frames
- TX Rate Control
- 11n, Rate Control

Wireless Infrastructure



Wireless Infrastructure

- Net80211
  - Handles 802.11 negotiation, protocol/session handling
- Driver (eg ath(4))
  - Handles TX/RX, frame completion, DMA, buffer management, interface management
- HAL (eg ath_hal(4))
  - Handles radio interfacing – register programming, calibration etc
- Rate control (eg ath_rate, net80211_ratectl)
  - Handles TX selection based on feedback from driver

QoS handling

- Net80211
  - QoS/WME parameters negotiated via beacon frames
- ieee80211_classify()
  - Determine the WME AC based on IPv4/IPv6 diffserv
- Each mbuf has a WME AC (ether_vtag)
- The driver then queues the frame to the relevant hardware queue.
  - .. the driver has to put it in the right queue!
  - .. the 802.11e settings for the hardware queue have to be correct!

Hardware Organisation

- Host Interface
  - PCI, PCIe, USB, etc
- Radio
  - TX and RX of differential signals, handles 2 and 5 GHz conversions
- PHY
  - Handles frame encoding/decoding, signal level determination, “RX busy” for clear channel assessment
- MAC – Medium Access Controller
  - Implements TX/RX DMA, encryption/decryption
  - Implements the 802.11 protocol handling
  - .. more to come

Media Access Controller

- Takes care of the 802.11 frames themselves
-

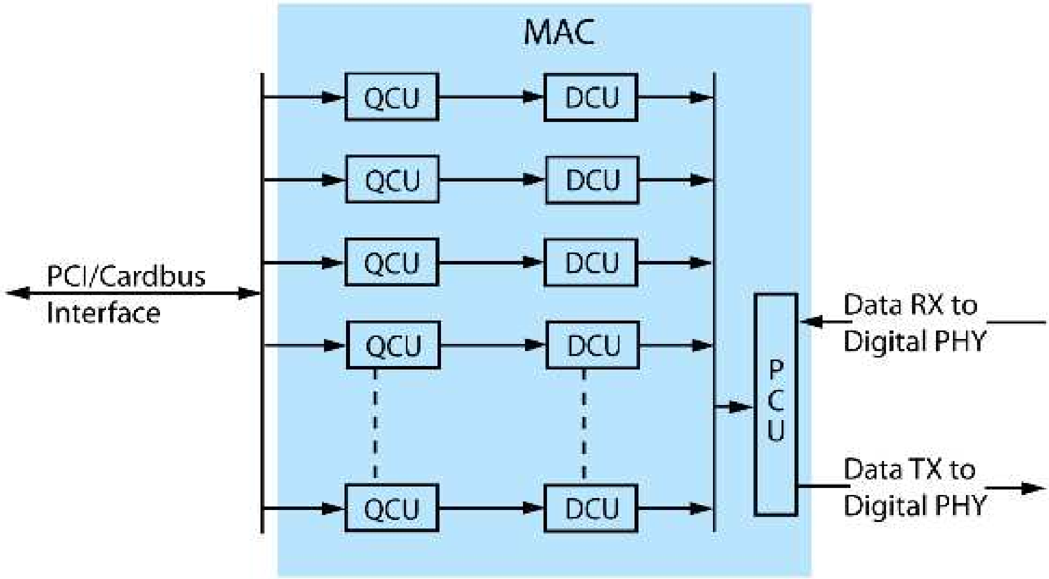


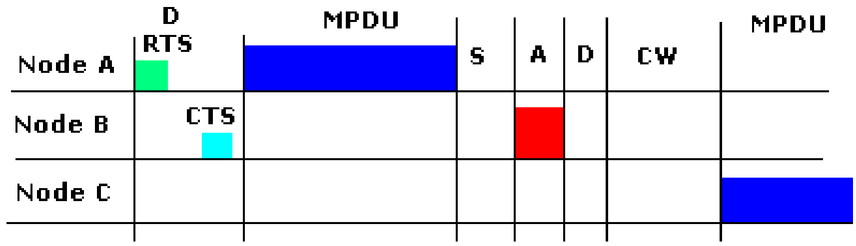
Figure 2: Atheros AR5212 MAC

Media Access Controller

- PCU
  - TX/RX scheduling, ACK/RTS/CTS handling, TSF (Timing synchronisation function – 1.024ms) handling, TU (beacon interval) handling, encryption/decryption, DCF/PCF (coordination function), 802.11n aggregation, Block-Ack handling
- QCU
  - Handles TX DMA from the host memory to the PCU
- DCU
  - Handles the distributed coordination function
- Each DCU “controls” a QCU, allowing it to TX
- This implements 802.11e priority queues

Implement 802.11

- The hardware implements 802.11 !
  - PCU: global settings such as SIFS, EIFS
  - RTS/CTS/ACK duration
  - QCU: burst duration, AIFS, contention window min/max, TX retry limits, back-off handling
- This implements a series of timers and state engines which implement part of 802.11 itself



Advantages!

- The hardware provides a lot of fine grained control over 802.11 “air” timing
  - .. making it easy to do experiments with arbitration, frame spacing, bursting, etc
- The hardware has some very nice QCU/DCU gating features
  - .. which allow traffic to be transmitted at certain times (based on beacon TSF timestamps)
  - .. this is how the TDMA solution works
- A lot is coded up in the HAL, but almost never actually used by the driver!

Example: TDMA

- TDMA allows..
  - a TX/RX pair to own a timeslice of air time
  - .. so they don't need to negotiate, do contention backoff, etc.
- It uses..
  - .. a QCU with a burst time set, where the unit can TX as much as it can fit in the time window..
  - .. a timer which fires after a number of microseconds following the beacon interval..
  - .. a gating method which pauses the TX queue until the timer fires, then automatically opens the QCU up for transmit.

Example: TDMA

- The host simply sets it up!
  - The hardware handles all of the timing, burst time handling, contention handling (ie, none), etc
- .. all the driver does is keep the clocks in sync.
- It's likely broken in -HEAD for now, sorry!

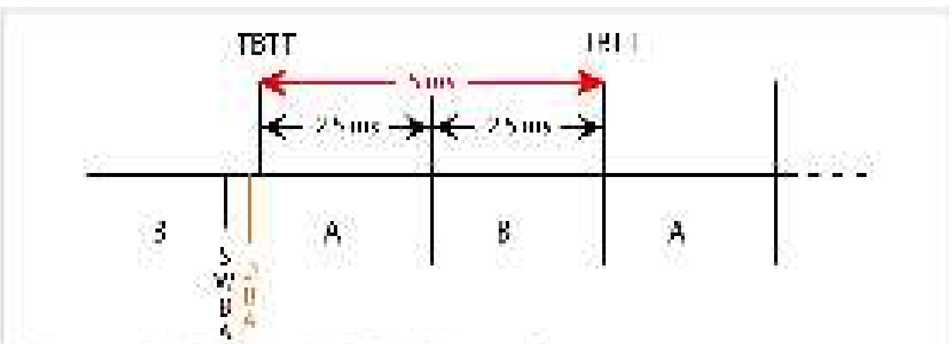
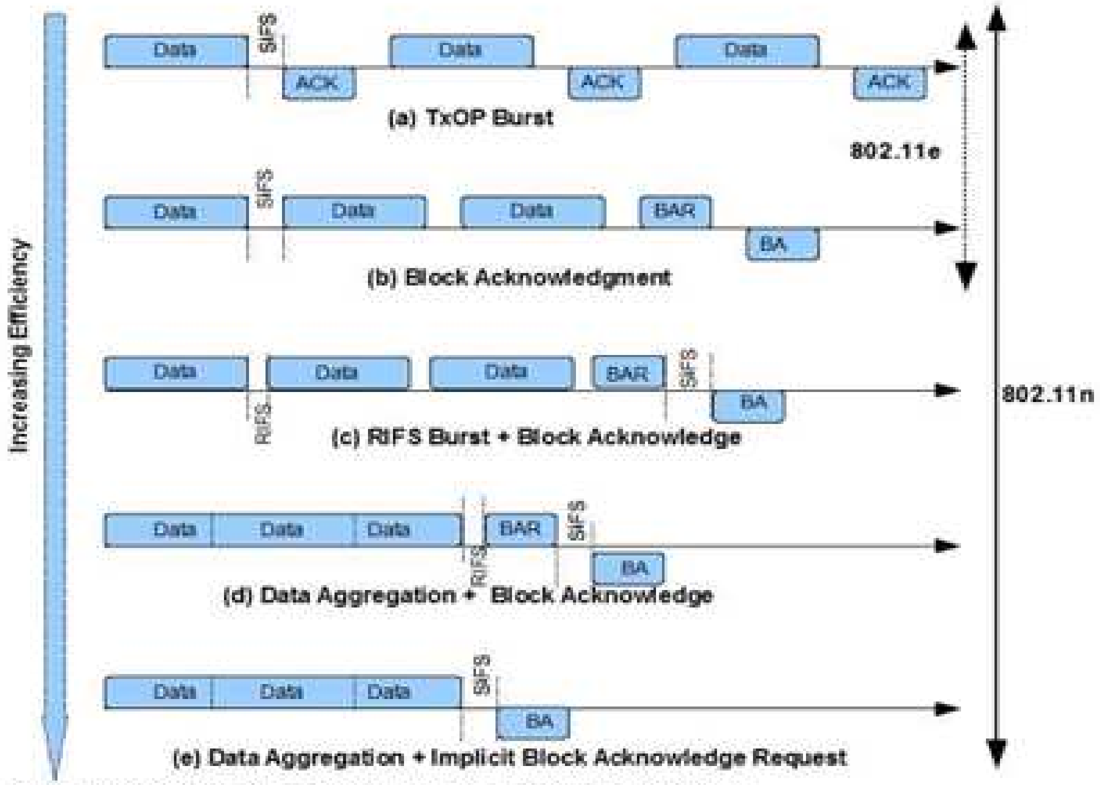


Figure 1: Example TDMA configuration.

Transmission Overheads

- 802.11 frames have fixed overheads
  - Contention Avoidance/Backoff
  - Preamble length
  - Initial frame header is done as legacy
  - RTS/CTS rates are done as legacy
  - ACK frames are done as legacy
- Higher density TX encodings only affect the data portion, not the rest of the frame
- So as you increase the TX symbol rate, the amount of fixed overhead doesn't change
- So your real world throughput doesn't increase!

Transmission Overheads



These improvements are shown in the first two rows of Figure (3).

Figure (3) MAC Improvements

Improving Transmission

- 802.11e
  - Either negotiated between station/AP, or part of the WME negotiation during association
  - If the station wishes to transmit on a higher priority queue, it uses different contention window parameters (Cwmin, Cwmax, AIFS, burst time)
- 802.11e + Burst
  - Again, can be negotiated as needed
  - The transmitter can TX during this period without waiting for the medium to be “idle” - it is assumed to be 100% available for it
  - Useful for VoIP, etc where latency is to be minimised

Improving Transmission

- A-MPDU – MPDU aggregation
  - Part of 802.11e, but is only implemented (these days) in 802.11n
  - The transmitter “bursts” many MPDU frames without contention or waiting for SIFS / ACK
  - A “block ACK” is sent at the end, indicating which sub-frames were successfully received
  - The software then retransmits whichever frames weren't successfully transmitted
  - The maximum burst length is 4ms! (due to legacy restrictions)

Improving Transmission

- However, A-MPDU has some issues
  - Tracking the retransmission window is complicated
    - But luckily, not a part of this discussion!
  - 4ms is a long time, but a lot of data can be squeezed (close to theoretical maximum throughput)
  - Highly noisy environments result in many, many retransmissions
  - .. so keeping the “air fair” whilst doing high throughput aggregation can be quite difficult
    - Where do you slot your VoIP traffic in when the NIC has been handed a 4ms frame?

Transmitting frames

- In FreeBSD-HEAD:
  - .. a software queue is maintained per-node and per-TID
  - .. 16 TIDs for each node;
  - .. WME AC's map to a single TID.
  - This is required for handling A-MPDU aggregation sessions, which is based on TID.
- The hardware then:
  - .. is handed a set of frames from the software queue
  - The hardware then does its own QoS, based on register settings
- Software retransmission is done as needed

Frame Receive

- A lot of useful information is available!
- Per-frame information:
  - Signal strength, received rate, CRC errors
  - 11n parameters (guard interval, STBC, whether an aggregate/burst, EVM)
  - PHY errors – helps to debug noisy/busy environments
- Global information:
  - Amount of time spent TX'ing, RX'ing
  - Amount of time the air was “busy”, so the hardware couldn't try TX'ing
  - Useful for determining how congested the air is!

Frame Transmission

- Each frame has a lot of parameters:
  - RTS/CTS, plus RTS/CTS duration and rate;
  - Whether an ACK is required;
  - Per-frame TX power level (TODO: not working yet!)
  - Overriding the duration field, for forcing NAV updates
  - Multi-rate retry: 4 attempts of ..
    - Which TX rate to use
    - How many times to try TX'ing
    - 11n parameters – guard interval, STBC, 20/40MHz mode
    - RTS/CTS enable
  - 11n:
    - How big the aggregate is; delimiters

Frame transmission completion

- Again, a lot of parameters are available:
  - Per frame:
    - Which TX rate succeeded
    - How many attempts for RTS/CTS negotiation
    - How many attempts at TX'ing the data (no ACK)
    - "Virtual collision" with 802.11e (eg going over burst duration)
    - ACK signal strength
    - 11n block-ack contents, TID
    - DMA status
    - Encryption engine status

TX rate control

- TX rate control allows for:
  - .. adapting to changing conditions
  - .. which may be different for each node
  - .. and may change unpredictably
- The API allows:
  - Rate selection
    - Choose a rate for each frame, based on the node and current conditions
  - Traffic completion
    - Analyse the completion data from each frame and update current conditions

TX rate control

- A few exist:
  - ath_rate: implements onoe, amrr, sample
  - Sample is the only one which supports 11n – and only in a basic way
    - It chooses a TX rate which minimises the average amount of time a frame takes to TX, given retransmission and backoff
  - ieee80211_ratectl: implements rssiadapt, amrr
    - ath(4) currently doesn't use this
    - .. the aim is to teach all wifi modules to use this!
    - It isn't 802.11n aware!
- .. it doesn't factor into TX queue or QoS parameters
  - .. it only controls TX rate selection!

11n and rate control?

- 802.11n has a large number of variables:
  - MCS encoding type (BPSK, QPSK, QAM)
  - Number of spatial streams
  - STBC (space-time block encoding)
  - TX power level
  - Short or long GI (guard interval)
  - Maximum aggregate length.
- FreeBSD/Linux only take into account the first two.
- It may be worthwhile adding in QoS awareness and queue management to the rate control API
  - .. since queue management influences TX performance!

What could be done?

- The hardware is powerful..
    - .. but nothing (FOSS) really goes in and tries to intelligently manage per-node frame queuing
    - .. based on current air conditions, rather than just traditional queue management techniques (eg RED, WRED, tail-drop, etc)
- Extend the rate control API to include the above?
    - .. allow rate control code to tune per-node, per-TID queue parameters
    - .. have the software queue code enforce this behaviour
    - .. perhaps export this to userland and allow userland TX classifiers (eg in python) to dynamically control these TX parameters?
- .. but the big one is this:

The worst case: too much TX?

- The hardware does frame retransmission for you
    - .. but the question is: how long is the hardware spending trying to transmit your frame?
        - .. whilst it's doing this, all other TX to nodes is on hold
        - .. and whilst it's TX'ing, it isn't RX'ing anything.
- .. so is one badly behaving node potentially messing up the entire airtime, for all potential nodes?
    - FreeBSD/Linux doesn't attempt to address this particular issue
    - .. either by logging useful data to establish if this is happening..
    - .. or dynamically limiting it from occurring
    - eg by reducing frame TX retries and doing it in software, allowing other nodes to TX.

Summary

- The Atheros NICs handle a lot for you:
  - 802.11 frame timing, transmission, retransmission
  - 802.11e parameters
  - Fine-grain control over when to TX frames
- Almost everything for 802.11 frame timing is a register somewhere..
  - .. and happily documented in the existing HAL code, so I don't have to break NDA to tell you this.
- But only a small part of this is really leveraged in FreeBSD/Linux!
  - .. but I bet commercial AP vendors are using it! :)

Questions?

References

- http://archives.ece.iastate.edu/archive/00000497/01/Thesis_PrateekGangwal.pdf
- http://www.zytrax.com/tech/wireless/802_mac.htm
- http://www.eetimes.com/design/communications-design/4206282/How-throughput-enhancements-dramatically-boost-802-11n-MA
- http://people.freebsd.org/~sam/FreeBSD_TDMA-20090921.pdf