****
25812 Wed Jun 29 14:08:27 2016
new/usr/src/uts/common/io/arn/arn_ani.c
7154 arn(7D) walks out of bounds when byteswapping the 4K eeprom
7152 weird condition in arn(7D) needs clarification
7153 delete unused code in arn(7D)
7155 arn(7D) should include the mac fields in the eeprom enumeration
****
_____***unchanged_portion_omitted_***

```
 49 static boolean_t
 50 ath9k_hw_ani_control(struct ath_hal *ah, enum ath9k_ani_cmd cmd, int param)
 51 {
 52         struct ath_hal_5416 *ahp = AH5416(ah);
 53         struct ar5416AniState *aniState = ahp->ah_curani;

 55         switch (cmd & ahp->ah_ani_function) {
 56         case ATH9K_ANI_NOISE_IMMUNITY_LEVEL: {
 57                 uint32_t level = param;

 59                 if (level >= ARRAY_SIZE(ahp->ah_totalSizeDesired)) {
 60                         ARN_DBG((ARN_DBG_ANI, "arn: "
 61                             "ah->ah_sc, ATH_DBG_ANI",
 62                             "%s: level out of range (%u > %u)\n",
 63                             __func__, level,
 64                             (unsigned)ARRAY_SIZE(ahp->ah_totalSizeDesired)));

 66                         return (B_FALSE);
 67                 }

 69                 REG_RMW_FIELD(ah, AR_PHY_DESIRED_SZ,
 70                     AR_PHY_DESIRED_SZ_TOT_DES,
 71                     ahp->ah_totalSizeDesired[level]);
 72                 REG_RMW_FIELD(ah, AR_PHY_AGC_CTL1,
 73                     AR_PHY_AGC_CTL1_COARSE_LOW,
 74                     ahp->ah_coarseLow[level]);
 75                 REG_RMW_FIELD(ah, AR_PHY_AGC_CTL1,
 76                     AR_PHY_AGC_CTL1_COARSE_HIGH,
 77                     ahp->ah_coarseHigh[level]);
 78                 REG_RMW_FIELD(ah, AR_PHY_FIND_SIG,
 79                     AR_PHY_FIND_SIG_FIRPWR,
 80                     ahp->ah_firpwr[level]);

 82                 if (level > aniState->noiseImmunityLevel)
 83                         ahp->ah_stats.ast_ani_niup++;
 84                 else if (level < aniState->noiseImmunityLevel)
 85                         ahp->ah_stats.ast_ani_nidown++;
 86                 aniState->noiseImmunityLevel = (uint8_t)level; /* LINT */
 87                 break;
 88         }
 89         case ATH9K_ANI_OFDM_WEAK_SIGNAL_DETECTION: {
 90                 const int m1ThreshLow[] = { 127, 50 };
 91                 const int m2ThreshLow[] = { 127, 40 };
 92                 const int m1Thresh[] = { 127, 0x4d };
 93                 const int m2Thresh[] = { 127, 0x40 };
 94                 const int m2CountThr[] = { 31, 16 };
 95                 const int m2CountThrLow[] = { 63, 48 };
 96                 uint32_t on = param ? 1 : 0;

 98                 REG_RMW_FIELD(ah, AR_PHY_SFCORR_LOW,
 99                     AR_PHY_SFCORR_LOW_M1_THRESH_LOW,
100                     m1ThreshLow[on]);
101                 REG_RMW_FIELD(ah, AR_PHY_SFCORR_LOW,
102                     AR_PHY_SFCORR_LOW_M2_THRESH_LOW,
103                     m2ThreshLow[on]);
104                 REG_RMW_FIELD(ah, AR_PHY_SFCORR,
```



```
105                     AR_PHY_SFCORR_M1_THRESH,
106                     m1Thresh[on]);
107                 REG_RMW_FIELD(ah, AR_PHY_SFCORR,
108                     AR_PHY_SFCORR_M2_THRESH,
109                     m2Thresh[on]);
110                 REG_RMW_FIELD(ah, AR_PHY_SFCORR,
111                     AR_PHY_SFCORR_M2COUNT_THR,
112                     m2CountThr[on]);
113                 REG_RMW_FIELD(ah, AR_PHY_SFCORR_LOW,
114                     AR_PHY_SFCORR_LOW_M2COUNT_THR_LOW,
115                     m2CountThrLow[on]);

117                 REG_RMW_FIELD(ah, AR_PHY_SFCORR_EXT,
118                     AR_PHY_SFCORR_EXT_M1_THRESH_LOW,
119                     m1ThreshLow[on]);
120                 REG_RMW_FIELD(ah, AR_PHY_SFCORR_EXT,
121                     AR_PHY_SFCORR_EXT_M2_THRESH_LOW,
122                     m2ThreshLow[on]);
123                 REG_RMW_FIELD(ah, AR_PHY_SFCORR_EXT,
124                     AR_PHY_SFCORR_EXT_M1_THRESH,
125                     m1Thresh[on]);
126                 REG_RMW_FIELD(ah, AR_PHY_SFCORR_EXT,
127                     AR_PHY_SFCORR_EXT_M2_THRESH,
128                     m2Thresh[on]);

130                 if (on)
131                         REG_SET_BIT(ah, AR_PHY_SFCORR_LOW,
132                             AR_PHY_SFCORR_LOW_USE_SELF_CORR_LOW);
133                 else
134                         REG_CLR_BIT(ah, AR_PHY_SFCORR_LOW,
135                             AR_PHY_SFCORR_LOW_USE_SELF_CORR_LOW);

137                 if ((!on) != aniState->ofdmWeakSigDetectOff) {
137                 if (!on != aniState->ofdmWeakSigDetectOff) {
138                         if (on)
139                                 ahp->ah_stats.ast_ani_ofdmon++;
140                         else
141                                 ahp->ah_stats.ast_ani_ofdmoff++;
142                         aniState->ofdmWeakSigDetectOff = !on;
143                 }
144                 break;
145         }
146         case ATH9K_ANI_CCK_WEAK_SIGNAL_THR: {
147                 const int weakSigThrCck[] = { 8, 6 };
148                 uint32_t high = param ? 1 : 0;

150                 REG_RMW_FIELD(ah, AR_PHY_CCK_DETECT,
151                     AR_PHY_CCK_DETECT_WEAK_SIG_THR_CCK,
152                     weakSigThrCck[high]);
153                 if (high != aniState->cckWeakSigThreshold) {
154                         if (high)
155                                 ahp->ah_stats.ast_ani_cckhigh++;
156                         else
157                                 ahp->ah_stats.ast_ani_ccklow++;
158                         /* LINT */
159                         aniState->cckWeakSigThreshold = (uint8_t)high;
160                 }
161                 break;
162         }
163         case ATH9K_ANI_FIRSTEP_LEVEL: {
164                 const int firstep[] = { 0, 4, 8 };
165                 uint32_t level = param;

167                 if (level >= ARRAY_SIZE(firstep)) {
168                         ARN_DBG((ARN_DBG_ANI, "arn: "
169                             "%s: level out of range (%u > %u)\n",
```

```
170                             __func__, level,
171                             (unsigned)ARRAY_SIZE(firstep)));

173                     return (B_FALSE);
174             }
175             REG_RMW_FIELD(ah, AR_PHY_FIND_SIG,
176                 AR_PHY_FIND_SIG_FIRSTEP, firstep[level]);
177             if (level > aniState->firstepLevel)
178                     ahp->ah_stats.ast_ani_stepup++;
179             else if (level < aniState->firstepLevel)
180                     ahp->ah_stats.ast_ani_stepdown++;
181             aniState->firstepLevel = (uint8_t)level; /* LINT */
182             break;
183     }
184     case ATH9K_ANI_SPUR_IMMUNITY_LEVEL: {
185             const int cycpwrThr1[] =
186                     { 2, 4, 6, 8, 10, 12, 14, 16 };
187             uint32_t level = param;

189             if (level >= ARRAY_SIZE(cycpwrThr1)) {
190                     ARN_DBG((ARN_DBG_ANI, "arn: "
191                         "%s: level out of range (%u > %u)\n",
192                         __func__, level,
193                         (unsigned)ARRAY_SIZE(cycpwrThr1)));

195                     return (B_FALSE);
196             }
197             REG_RMW_FIELD(ah, AR_PHY_TIMING5,
198                 AR_PHY_TIMING5_CYCPWR_THR1, cycpwrThr1[level]);
199             if (level > aniState->spurImmunityLevel)
200                     ahp->ah_stats.ast_ani_spurup++;
201             else if (level < aniState->spurImmunityLevel)
202                     ahp->ah_stats.ast_ani_spurdown++;
203             aniState->spurImmunityLevel = (uint8_t)level; /* LINT */
204             break;
205     }
206     case ATH9K_ANI_PRESENT:
207             break;
208     default:
209             ARN_DBG((ARN_DBG_ANI, "arn: "
210                 "%s: invalid cmd %u\n", __func__, cmd));
211             return (B_FALSE);
212     }

214     ARN_DBG((ARN_DBG_ANI, "arn: "
215         "%s: ANI parameters:\n", __func__));
216     ARN_DBG((ARN_DBG_ANI, "arn: "
217         "noiseImmunityLevel=%d, spurImmunityLevel=%d, "
218         "ofdmWeakSigDetectOff=%d\n",
219         aniState->noiseImmunityLevel, aniState->spurImmunityLevel,
220         !aniState->ofdmWeakSigDetectOff));
221     ARN_DBG((ARN_DBG_ANI, "arn: "
222         "cckWeakSigThreshold=%d, "
223         "firstepLevel=%d, listenTime=%d\n",
224         aniState->cckWeakSigThreshold, aniState->firstepLevel,
225         aniState->listenTime));
226     ARN_DBG((ARN_DBG_ANI, "arn: "
227         "cycleCount=%d, ofdmPhyErrCount=%d, cckPhyErrCount=%d\n\n",
228         aniState->cycleCount, aniState->ofdmPhyErrCount,
229         aniState->cckPhyErrCount));

231     return (B_TRUE);
232 }
```

unchanged_portion_omitted_

```
**********************************************************
   10951 Wed Jun 29 14:08:28 2016
new/usr/src/uts/common/io/arn/arn_beacon.c
7154 arn(7D) walks out of bounds when byteswapping the 4K eeprom
7152 weird condition in arn(7D) needs clarification
7153 delete unused code in arn(7D)
7155 arn(7D) should include the mac fields in the eeprom enumeration
**********************************************************
```

```
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2008 Atheros Communications Inc.
 *
 * Permission to use, copy, modify, and/or distribute this software for any
 * purpose with or without fee is hereby granted, provided that the above
 * copyright notice and this permission notice appear in all copies.
 *
 * THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
 * WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
 * ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
 * ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 * OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 */

#include <sys/param.h>
#include <sys/strsun.h>
#include <inet/common.h>
#include <inet/nd.h>
#include <inet/mi.h>
#include <inet/wifi_ioctl.h>

#include "arn_core.h"

/*
 * This function will modify certain transmit queue properties depending on
 * the operating mode of the station (AP or AdHoc).  Parameters are AIFS
 * settings and channel width min/max
 */

static int
/* LINTED E_STATIC_UNUSED */
arn_beaconq_config(struct arn_softc *sc)
{
        struct ath_hal *ah = sc->sc_ah;
        struct ath9k_tx_queue_info qi;

        (void) ath9k_hw_get_txq_props(ah, sc->sc_beaconq, &qi);
        if (sc->sc_ah->ah_opmode == ATH9K_M_HOSTAP) {
                /* Always burst out beacon and CAB traffic. */
                qi.tqi_aifs = 1;
                qi.tqi_cwmin = 0;
                qi.tqi_cwmax = 0;
        } else {
                /* Adhoc mode; important thing is to use 2x cwmin. */
                qi.tqi_aifs = sc->sc_beacon_qi.tqi_aifs;
                qi.tqi_cwmin = 2*sc->sc_beacon_qi.tqi_cwmin;
                qi.tqi_cwmax = sc->sc_beacon_qi.tqi_cwmax;
        }

        if (!ath9k_hw_set_txq_props(ah, sc->sc_beaconq, &qi)) {
                arn_problem("unable to update h/w beacon queue parameters\n");
```



```
                return (0);
        } else {
                /* push to h/w */
                (void) ath9k_hw_resettxqueue(ah, sc->sc_beaconq);
                return (1);
        }
}

/*
 * Associates the beacon frame buffer with a transmit descriptor.  Will set
 * up all required antenna switch parameters, rate codes, and channel flags.
 * Beacons are always sent out at the lowest rate, and are not retried.
 */
#ifdef ARN_IBSS
static void
arn_beacon_setup(struct arn_softc *sc, struct ath_buf *bf)
{
#define USE_SHPREAMBLE(_ic) \
        (((_ic)->ic_flags & (IEEE80211_F_SHPREAMBLE | IEEE80211_F_USEBARKER))\
            == IEEE80211_F_SHPREAMBLE)
        mblk_t *mp = bf->bf_m;
        struct ath_hal *ah = sc->sc_ah;
        struct ath_desc *ds;
        /* LINTED E_FUNC_SET_NOT_USED */
        int flags, antenna = 0;
        struct ath_rate_table *rt;
        uint8_t rix, rate;
        struct ath9k_11n_rate_series series[4];
        int ctsrate = 0;
        int ctsduration = 0;

        /* set up descriptors */
        ds = bf->bf_desc;

        flags = ATH9K_TXDESC_NOACK;
        if (sc->sc_ah->ah_opmode == ATH9K_M_IBSS &&
            (ah->ah_caps.hw_caps & ATH9K_HW_CAP_VEOL)) {
                ds->ds_link = bf->bf_daddr;     /* self-linked */
                flags |= ATH9K_TXDESC_VEOL;
                /*
                 * Let hardware handle antenna switching.
                 */
                antenna = 0;
        } else {
                ds->ds_link = 0;
                /*
                 * Switch antenna every 4 beacons.
                 * NB: assumes two antenna
                 */
                antenna = ((sc->ast_be_xmit / sc->sc_nbcnvaps) & 1 ? 2 : 1);
        }

        ds->ds_data = bf->bf_dma.cookie.dmac_address;
        /*
         * Calculate rate code.
         * XXX everything at min xmit rate
         */
        rix = 0;
        rt = sc->hw_rate_table[sc->sc_curmode];
        rate = rt->info[rix].ratecode;
        if (sc->sc_flags & SC_OP_PREAMBLE_SHORT)
                rate |= rt->info[rix].short_preamble;

        ath9k_hw_set11n_txdesc(ah, ds,
            MBLKL(mp) + IEEE80211_CRC_LEN, /* frame length */
            ATH9K_PKT_TYPE_BEACON,      /* Atheros packet type */
```

```
 89             MAX_RATE_POWER,             /* FIXME */
 90             ATH9K_TXKEYIX_INVALID,      /* no encryption */
 91             ATH9K_KEY_TYPE_CLEAR,       /* no encryption */
 92             flags);                     /* no ack, veol for beacons */

 94         /* NB: beacon's BufLen must be a multiple of 4 bytes */
 95         (void) ath9k_hw_filltxdesc(ah, ds,
 96             roundup(MBLKL(mp), 4),      /* buffer length */
 97             B_TRUE,                     /* first segment */
 98             B_TRUE,                     /* last segment */
 99             ds);                        /* first descriptor */

101         (void) memset(series, 0, sizeof (struct ath9k_11n_rate_series) * 4);
102         series[0].Tries = 1;
103         series[0].Rate = rate;
104         series[0].ChSel = sc->sc_tx_chainmask;
105         series[0].RateFlags = (ctsrate) ? ATH9K_RATESERIES_RTS_CTS : 0;
106         ath9k_hw_set11n_ratescenario(ah, ds, ds, 0,
107             ctsrate, ctsduration, series, 4, 0);
108 #undef  USE_SHPREAMBLE
109 }
110 #endif

112 /*
113  * Startup beacon transmission for adhoc mode when they are sent entirely
114  * by the hardware using the self-linked descriptor + veol trick.
115  */
116 #ifdef ARN_IBSS
117 static void
118 arn_beacon_start_adhoc(struct arn_softc *sc)

119 {
120         struct ath_buf *bf = list_head(&sc->sc_bcbuf_list);
121         struct ieee80211_node *in = bf->bf_in;
122         struct ieee80211com *ic = in->in_ic;
123         struct ath_hal *ah = sc->sc_ah;
124         mblk_t *mp;

126         mp = bf->bf_m;
127         if (ieee80211_beacon_update(ic, bf->bf_in, &sc->asc_boff, mp, 0))
128                 bcopy(mp->b_rptr, bf->bf_dma.mem_va, MBLKL(mp));

130         /* Construct tx descriptor. */
131         arn_beacon_setup(sc, bf);

133         /*
134          * Stop any current dma and put the new frame on the queue.
135          * This should never fail since we check above that no frames
136          * are still pending on the queue.
137          */
138         if (!ath9k_hw_stoptxdma(ah, sc->sc_beaconq)) {
139                 arn_problem("ath: beacon queue %d did not stop?\n",
140                     sc->sc_beaconq);
141         }
142         ARN_DMA_SYNC(bf->bf_dma, DDI_DMA_SYNC_FORDEV);

144         /* NB: caller is known to have already stopped tx dma */
145         (void) ath9k_hw_puttxbuf(ah, sc->sc_beaconq, bf->bf_daddr);
146         (void) ath9k_hw_txstart(ah, sc->sc_beaconq);

148         ARN_DBG((ARN_DBG_BEACON, "arn: arn_bstuck_process(): "
149             "TXDP%u = %llx (%p)\n", sc->sc_beaconq,
150             ito64(bf->bf_daddr), bf->bf_desc));
151 }
```

unchanged_portion_omitted_


```
221 void
222 arn_beacon_config(struct arn_softc *sc)

223 {
224         struct ath_beacon_config conf;
225         ieee80211com_t *ic = (ieee80211com_t *)sc;
226         struct ieee80211_node *in = ic->ic_bss;

228         /* New added */
229         struct ath9k_beacon_state bs;
230         int dtimperiod, dtimcount, sleepduration;
231         int cfpperiod, cfpcount;
232         uint32_t nexttbtt = 0, intval, tsftu;
233         uint64_t tsf;

235         (void) memset(&conf, 0, sizeof (struct ath_beacon_config));

237         /* XXX fix me */
238         conf.beacon_interval = in->in_intval ?
239             in->in_intval : ATH_DEFAULT_BINTVAL;
240         ARN_DBG((ARN_DBG_BEACON, "arn: arn_beacon_config():"
241             "conf.beacon_interval = %d\n", conf.beacon_interval));
242         conf.listen_interval = 1;
243         conf.dtim_period = conf.beacon_interval;
244         conf.dtim_count = 1;
245         conf.bmiss_timeout = ATH_DEFAULT_BMISS_LIMIT * conf.beacon_interval;

247         (void) memset(&bs, 0, sizeof (bs));
248         intval = conf.beacon_interval & ATH9K_BEACON_PERIOD;

250         /*
251          * Setup dtim and cfp parameters according to
252          * last beacon we received (which may be none).
253          */
254         dtimperiod = conf.dtim_period;
255         if (dtimperiod <= 0)            /* NB: 0 if not known */
256                 dtimperiod = 1;
257         dtimcount = conf.dtim_count;
258         if (dtimcount >= dtimperiod)    /* NB: sanity check */
259                 dtimcount = 0;
260         cfpperiod = 1;                  /* NB: no PCF support yet */
261         cfpcount = 0;

263         sleepduration = conf.listen_interval * intval;
264         if (sleepduration <= 0)
265                 sleepduration = intval;

267         /*
268          * Pull nexttbtt forward to reflect the current
269          * TSF and calculate dtim+cfp state for the result.
270          */
271         tsf = ath9k_hw_gettsf64(sc->sc_ah);
272         tsftu = TSF_TO_TU(tsf>>32, tsf) + FUDGE;
273         do {
274                 nexttbtt += intval;
275                 if (--dtimcount < 0) {
276                         dtimcount = dtimperiod - 1;
277                         if (--cfpcount < 0)
278                                 cfpcount = cfpperiod - 1;
279                 }
280         } while (nexttbtt < tsftu);

282         bs.bs_intval = intval;
283         bs.bs_nexttbtt = nexttbtt;
284         bs.bs_dtimperiod = dtimperiod*intval;
285         bs.bs_nextdtim = bs.bs_nexttbtt + dtimcount*intval;
```

```
286         bs.bs_cfpperiod = cfpperiod*bs.bs_dtimperiod;
287         bs.bs_cfpnext = bs.bs_nextdtim + cfpcount*bs.bs_dtimperiod;
288         bs.bs_cfpmaxduration = 0;

290         /*
291          * Calculate the number of consecutive beacons to miss* before taking
292          * a BMISS interrupt. The configuration is specified in TU so we only
293          * need calculate based on the beacon interval.  Note that we clamp the
294          * result to at most 15 beacons.
295          */
296         if (sleepduration > intval) {
297                 bs.bs_bmissthreshold = conf.listen_interval *
298                     ATH_DEFAULT_BMISS_LIMIT / 2;
299         } else {
300                 bs.bs_bmissthreshold = DIV_ROUND_UP(conf.bmiss_timeout, intval);
301                 if (bs.bs_bmissthreshold > 15)
302                         bs.bs_bmissthreshold = 15;
303                 else if (bs.bs_bmissthreshold == 0)
304                         bs.bs_bmissthreshold = 1;
305         }

307         /*
308          * Calculate sleep duration. The configuration is given in ms.
309          * We ensure a multiple of the beacon period is used. Also, if the sleep
310          * duration is greater than the DTIM period then it makes senses
311          * to make it a multiple of that.
312          *
313          * XXX fixed at 100ms
314          */

316         bs.bs_sleepduration = roundup(IEEE80211_MS_TO_TU(100), sleepduration);
317         if (bs.bs_sleepduration > bs.bs_dtimperiod)
318                 bs.bs_sleepduration = bs.bs_dtimperiod;

320         /* TSF out of range threshold fixed at 1 second */
321         bs.bs_tsfoor_threshold = ATH9K_TSFOOR_THRESHOLD;

323         ARN_DBG((ARN_DBG_BEACON, "arn: arn_beacon_config(): "
324             "tsf %llu "
325             "tsf:tu %u "
326             "intval %u "
327             "nexttbtt %u "
328             "dtim %u "
329             "nextdtim %u "
330             "bmiss %u "
331             "sleep %u "
332             "cfp:period %u "
333             "maxdur %u "
334             "next %u "
335             "timoffset %u\n",
336             (unsigned long long)tsf, tsftu,
337             bs.bs_intval,
338             bs.bs_nexttbtt,
339             bs.bs_dtimperiod,
340             bs.bs_nextdtim,
341             bs.bs_bmissthreshold,
342             bs.bs_sleepduration,
343             bs.bs_cfpperiod,
344             bs.bs_cfpmaxduration,
345             bs.bs_cfpnext,
346             bs.bs_timoffset));

348         /* Set the computed STA beacon timers */

350         (void) ath9k_hw_set_interrupts(sc->sc_ah, 0);
351         ath9k_hw_set_sta_beacon_timers(sc->sc_ah, &bs);
```



```
352         sc->sc_imask |= ATH9K_INT_BMISS;
353         (void) ath9k_hw_set_interrupts(sc->sc_ah, sc->sc_imask);
354 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   28761 Wed Jun 29 14:08:28 2016
new/usr/src/uts/common/io/arn/arn_calib.c
7154 arn(7D) walks out of bounds when byteswapping the 4K eeprom
7152 weird condition in arn(7D) needs clarification
7153 delete unused code in arn(7D)
7155 arn(7D) should include the mac fields in the eeprom enumeration
**********************************************************
______unchanged_portion_omitted_

 721 int16_t
 722 ath9k_hw_getnf(struct ath_hal *ah, struct ath9k_channel *chan)
 723 {
 724         int16_t nf, nfThresh;
 725         int16_t nfarray[NUM_NF_READINGS] = { 0 };
 726         struct ath9k_nfcal_hist *h;
 727         /* LINTED E_FUNC_SET_NOT_USED */
 728         uint8_t chainmask;

 730         if (AR_SREV_9280(ah))
 731                 chainmask = 0x1B;
 732         else
 733                 chainmask = 0x3F;

 728         chan->channelFlags &= (~CHANNEL_CW_INT);
 729         if (REG_READ(ah, AR_PHY_AGC_CONTROL) & AR_PHY_AGC_CONTROL_NF) {
 730                 ARN_DBG((ARN_DBG_CALIBRATE, "arn: "
 731                     "%s: NF did not complete in calibration window\n",
 732                     __func__));
 733                 nf = 0;
 734                 chan->rawNoiseFloor = nf;
 735                 return (chan->rawNoiseFloor);
 736         } else {
 737                 ath9k_hw_do_getnf(ah, nfarray);
 738                 nf = nfarray[0];
 739                 if (getNoiseFloorThresh(ah, chan, &nfThresh) &&
 740                     nf > nfThresh) {
 741                         ARN_DBG((ARN_DBG_CALIBRATE, "arn: "
 742                             "%s: noise floor failed detected; "
 743                             "detected %d, threshold %d\n", __func__,
 744                             nf, nfThresh));
 745                         chan->channelFlags |= CHANNEL_CW_INT;
 746                 }
 747         }

 749 #ifdef ARN_NF_PER_CHAN
 750         h = chan->nfCalHist;
 751 #else
 752         h = ah->nfCalHist;
 753 #endif

 755         ath9k_hw_update_nfcal_hist_buffer(h, nfarray);
 756         chan->rawNoiseFloor = h[0].privNF;

 758         return (chan->rawNoiseFloor);
 759 }
______unchanged_portion_omitted_
```

```
**********************************************************
   82985 Wed Jun 29 14:08:29 2016
new/usr/src/uts/common/io/arn/arn_eeprom.c
7154 arn(7D) walks out of bounds when byteswapping the 4K eeprom
7152 weird condition in arn(7D) needs clarification
7153 delete unused code in arn(7D)
7155 arn(7D) should include the mac fields in the eeprom enumeration
**********************************************************
______unchanged_portion_omitted_

 370 static int
 371 ath9k_hw_check_4k_eeprom(struct ath_hal *ah)
 372 {
 373 #define EEPROM_4K_SIZE  (sizeof (struct ar5416_eeprom_4k) / sizeof (uint16_t))
 374         struct ath_hal_5416 *ahp = AH5416(ah);
 375         struct ar5416_eeprom_4k *eep =
 376             (struct ar5416_eeprom_4k *)&ahp->ah_eeprom.map4k;
 377         uint16_t *eepdata, temp, magic, magic2;
 378         uint32_t sum = 0, el;
 379         boolean_t need_swap = B_FALSE;
 380         int i, addr;


 383         if (!ath9k_hw_use_flash(ah)) {

 385                 if (!ath9k_hw_nvram_read(ah, AR5416_EEPROM_MAGIC_OFFSET,
 386                     &magic)) {
 387                         ARN_DBG((ARN_DBG_EEPROM,
 388                             "Reading Magic # failed\n"));
 389                         return (B_FALSE);
 390                 }

 392                 ARN_DBG((ARN_DBG_EEPROM,
 393                     "Read Magic = 0x%04X\n", magic));

 395                 if (magic != AR5416_EEPROM_MAGIC) {
 396                         magic2 = swab16(magic);

 398                         if (magic2 == AR5416_EEPROM_MAGIC) {
 399                                 need_swap = B_TRUE;
 400                                 eepdata = (uint16_t *)(&ahp->ah_eeprom);

 402                                 for (addr = 0; addr < EEPROM_4K_SIZE; addr++) {
 403                                         temp = swab16(*eepdata);
 404                                         *eepdata = temp;
 405                                         eepdata++;

 407                                         ARN_DBG((ARN_DBG_EEPROM,
 408                                             "0x%04X  ", *eepdata));

 410                                         if (((addr + 1) % 6) == 0)
 411                                                 ARN_DBG((ARN_DBG_EEPROM, "\n"));
 412                                 }
 413                         } else {
 414                                 ARN_DBG((ARN_DBG_EEPROM,
 415                                     "Invalid EEPROM Magic. "
 416                                     "endianness mismatch.\n"));
 417                                 return (EINVAL);
 418                         }
 419                 }
 420         }

 422         ARN_DBG((ARN_DBG_EEPROM, "need_swap = %s.\n",
 423             need_swap ? "True" : "False"));

 425         if (need_swap)
```



```
 426                 el = swab16(ahp->ah_eeprom.map4k.baseEepHeader.length);
 427         else
 428                 el = ahp->ah_eeprom.map4k.baseEepHeader.length;

 430         if (el > sizeof (struct ar5416_eeprom_def))
 431                 el = sizeof (struct ar5416_eeprom_4k) / sizeof (uint16_t);
 432         else
 433                 el = el / sizeof (uint16_t);

 435         eepdata = (uint16_t *)(&ahp->ah_eeprom);

 437         for (i = 0; i < el; i++)
 438                 sum ^= *eepdata++;

 440         if (need_swap) {
 441                 uint32_t integer;
 442                 uint16_t word;

 444                 ARN_DBG((ARN_DBG_EEPROM,
 445                     "EEPROM Endianness is not native.. Changing \n"));

 447                 word = swab16(eep->baseEepHeader.length);
 448                 eep->baseEepHeader.length = word;

 450                 word = swab16(eep->baseEepHeader.checksum);
 451                 eep->baseEepHeader.checksum = word;

 453                 word = swab16(eep->baseEepHeader.version);
 454                 eep->baseEepHeader.version = word;

 456                 word = swab16(eep->baseEepHeader.regDmn[0]);
 457                 eep->baseEepHeader.regDmn[0] = word;

 459                 word = swab16(eep->baseEepHeader.regDmn[1]);
 460                 eep->baseEepHeader.regDmn[1] = word;

 462                 word = swab16(eep->baseEepHeader.rfSilent);
 463                 eep->baseEepHeader.rfSilent = word;

 465                 word = swab16(eep->baseEepHeader.blueToothOptions);
 466                 eep->baseEepHeader.blueToothOptions = word;

 468                 word = swab16(eep->baseEepHeader.deviceCap);
 469                 eep->baseEepHeader.deviceCap = word;

 471                 integer = swab32(eep->modalHeader.antCtrlCommon);
 472                 eep->modalHeader.antCtrlCommon = integer;

 474                 for (i = 0; i < AR5416_EEP4K_MAX_CHAINS; i++) {
 474                 for (i = 0; i < AR5416_MAX_CHAINS; i++) {
 475                         integer = swab32(eep->modalHeader.antCtrlChain[i]);
 476                         eep->modalHeader.antCtrlChain[i] = integer;
 477                 }

 479                 for (i = 0; i < AR5416_EEPROM_MODAL_SPURS; i++) {
 480                         word = swab16(eep->modalHeader.spurChans[i].spurChan);
 481                         eep->modalHeader.spurChans[i].spurChan = word;
 482                 }
 483         }

 485         if (sum != 0xffff || ar5416_get_eep4k_ver(ahp) != AR5416_EEP_VER ||
 486             ar5416_get_eep4k_rev(ahp) < AR5416_EEP_NO_BACK_VER) {
 487                 ARN_DBG((ARN_DBG_EEPROM,
 488                     "Bad EEPROM checksum 0x%x or revision 0x%04x\n",
 489                     sum, ar5416_get_eep4k_ver(ahp)));
 490                 return (EINVAL);
```

```
 491         }

 493         return (0);
 494 #undef EEPROM_4K_SIZE
 495 }
```

_____***unchanged_portion_omitted_***

```
*********************************************************
   30616 Wed Jun 29 14:08:30 2016
new/usr/src/uts/common/io/arn/arn_hw.h
7154 arn(7D) walks out of bounds when byteswapping the 4K eeprom
7152 weird condition in arn(7D) needs clarification
7153 delete unused code in arn(7D)
7155 arn(7D) should include the mac fields in the eeprom enumeration
*********************************************************
______unchanged_portion_omitted_

 405 #define AR5416_OPFLAGS_11A              0x01
 406 #define AR5416_OPFLAGS_11G              0x02
 407 #define AR5416_OPFLAGS_N_5G_HT40        0x04
 408 #define AR5416_OPFLAGS_N_2G_HT40        0x08
 409 #define AR5416_OPFLAGS_N_5G_HT20        0x10
 410 #define AR5416_OPFLAGS_N_2G_HT20        0x20

 412 #define EEP_RFSILENT_ENABLED            0x0001
 413 #define EEP_RFSILENT_ENABLED_S          0
 414 #define EEP_RFSILENT_POLARITY           0x0002
 415 #define EEP_RFSILENT_POLARITY_S         1
 416 #define EEP_RFSILENT_GPIO_SEL           0x001c
 417 #define EEP_RFSILENT_GPIO_SEL_S         2

 419 #define AR5416_EEP_NO_BACK_VER          0x1
 420 #define AR5416_EEP_VER                  0xE
 421 #define AR5416_EEP_VER_MINOR_MASK       0x0FFF
 422 #define AR5416_EEP_MINOR_VER_2          0x2
 423 #define AR5416_EEP_MINOR_VER_3          0x3
 424 #define AR5416_EEP_MINOR_VER_7          0x7
 425 #define AR5416_EEP_MINOR_VER_9          0x9
 426 #define AR5416_EEP_MINOR_VER_16         0x10
 427 #define AR5416_EEP_MINOR_VER_17         0x11
 428 #define AR5416_EEP_MINOR_VER_19         0x13
 429 /* 2.6.30 */
 430 #define AR5416_EEP_MINOR_VER_20         0x14
 431 #define AR5416_EEP_MINOR_VER_22         0x16

 433 #define AR5416_NUM_5G_CAL_PIERS         8
 434 #define AR5416_NUM_2G_CAL_PIERS         4
 435 #define AR5416_NUM_5G_20_TARGET_POWERS  8
 436 #define AR5416_NUM_5G_40_TARGET_POWERS  8
 437 #define AR5416_NUM_2G_CCK_TARGET_POWERS 3
 438 #define AR5416_NUM_2G_20_TARGET_POWERS  4
 439 #define AR5416_NUM_2G_40_TARGET_POWERS  4
 440 #define AR5416_NUM_CTLS                 24
 441 #define AR5416_NUM_BAND_EDGES           8
 442 #define AR5416_NUM_PD_GAINS             4
 443 #define AR5416_PD_GAINS_IN_MASK         4
 444 #define AR5416_PD_GAIN_ICEPTS           5
 445 #define AR5416_EEPROM_MODAL_SPURS       5
 446 #define AR5416_MAX_RATE_POWER           63
 447 #define AR5416_NUM_PDADC_VALUES         128
 448 #define AR5416_BCHAN_UNUSED             0xFF
 449 #define AR5416_MAX_PWR_RANGE_IN_HALF_DB 64
 450 #define AR5416_MAX_CHAINS               3
 451 #define AR5416_PWR_TABLE_OFFSET         -5

 453 /* Rx gain type values */
 454 #define AR5416_EEP_RXGAIN_23DB_BACKOFF  0
 455 #define AR5416_EEP_RXGAIN_13DB_BACKOFF  1
 456 #define AR5416_EEP_RXGAIN_ORIG          2

 458 /* Tx gain type values */
 459 #define AR5416_EEP_TXGAIN_ORIGINAL      0
 460 #define AR5416_EEP_TXGAIN_HIGH_POWER    1
```



```
 462 #define AR5416_EEP4K_START_LOC                      64
 463 #define AR5416_EEP4K_NUM_2G_CAL_PIERS               3
 464 #define AR5416_EEP4K_NUM_2G_CCK_TARGET_POWERS       3
 465 #define AR5416_EEP4K_NUM_2G_20_TARGET_POWERS        3
 466 #define AR5416_EEP4K_NUM_2G_40_TARGET_POWERS        3
 467 #define AR5416_EEP4K_NUM_CTLS                       12
 468 #define AR5416_EEP4K_NUM_BAND_EDGES                 4
 469 #define AR5416_EEP4K_NUM_PD_GAINS                   2
 470 #define AR5416_EEP4K_PD_GAINS_IN_MASK               4
 471 #define AR5416_EEP4K_PD_GAIN_ICEPTS                 5
 472 #define AR5416_EEP4K_MAX_CHAINS                     1

 474 enum eeprom_param {
 475         EEP_NFTHRESH_5 = 0,
 475         EEP_NFTHRESH_5,
 476         EEP_NFTHRESH_2,
 477         EEP_MAC_MSW,
 478         EEP_MAC_MID,
 479         EEP_MAC_LSW,
 480         EEP_REG_0,
 481         EEP_REG_1,
 482         EEP_OP_CAP,
 483         EEP_OP_MODE,
 484         EEP_RF_SILENT,
 485         EEP_OB_5,
 486         EEP_DB_5,
 487         EEP_OB_2,
 488         EEP_DB_2,
 489         EEP_MINOR_REV,
 490         EEP_TX_MASK,
 491         EEP_RX_MASK,
 492         EEP_RXGAIN_TYPE,
 493         EEP_TXGAIN_TYPE,
 494         EEP_OL_PWRCTRL,
 495         EEP_RC_CHAIN_MASK,
 496         EEP_DAC_HPWR_5G,
 497         EEP_FRAC_N_5G,
 498         EEP_MAC_0 = AR_EEPROM_MAC(0),
 499         EEP_MAC_1 = AR_EEPROM_MAC(1),
 500         EEP_MAC_2 = AR_EEPROM_MAC(2)
 497         EEP_FRAC_N_5G
 501 };
______unchanged_portion_omitted_
```

```
**********************************************************
   88259 Wed Jun 29 14:08:31 2016
new/usr/src/uts/common/io/arn/arn_main.c
7154 arn(7D) walks out of bounds when byteswapping the 4K eeprom
7152 weird condition in arn(7D) needs clarification
7153 delete unused code in arn(7D)
7155 arn(7D) should include the mac fields in the eeprom enumeration
**********************************************************
______unchanged_portion_omitted_
```

```
505 static struct ath_rate_table *
506 /* LINTED E_STATIC_UNUSED */
507 arn_get_ratetable(struct arn_softc *sc, uint32_t mode)
508 {
509         struct ath_rate_table *rate_table = NULL;

511         switch (mode) {
512         case IEEE80211_MODE_11A:
513                 rate_table = sc->hw_rate_table[ATH9K_MODE_11A];
514                 break;
515         case IEEE80211_MODE_11B:
516                 rate_table = sc->hw_rate_table[ATH9K_MODE_11B];
517                 break;
518         case IEEE80211_MODE_11G:
519                 rate_table = sc->hw_rate_table[ATH9K_MODE_11G];
520                 break;
521 #ifdef ARB_11N
522         case IEEE80211_MODE_11NA_HT20:
523                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NA_HT20];
524                 break;
525         case IEEE80211_MODE_11NG_HT20:
526                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NG_HT20];
527                 break;
528         case IEEE80211_MODE_11NA_HT40PLUS:
529                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NA_HT40PLUS];
530                 break;
531         case IEEE80211_MODE_11NA_HT40MINUS:
532                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NA_HT40MINUS];
533                 break;
534         case IEEE80211_MODE_11NG_HT40PLUS:
535                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NG_HT40PLUS];
536                 break;
537         case IEEE80211_MODE_11NG_HT40MINUS:
538                 rate_table = sc->hw_rate_table[ATH9K_MODE_11NG_HT40MINUS];
539                 break;
540 #endif
541         default:
542                 ARN_DBG((ARN_DBG_FATAL, "arn: arn_get_ratetable(): "
543                     "invalid mode %u\n", mode));
544                 return (NULL);
545         }

547         return (rate_table);

549 }
```

```
505 static void
506 arn_setcurmode(struct arn_softc *sc, enum wireless_mode mode)
507 {
508         struct ath_rate_table *rt;
509         int i;

511         for (i = 0; i < sizeof (sc->asc_rixmap); i++)
512                 sc->asc_rixmap[i] = 0xff;

514         rt = sc->hw_rate_table[mode];
```



```
515         ASSERT(rt != NULL);

517         for (i = 0; i < rt->rate_cnt; i++)
518                 sc->asc_rixmap[rt->info[i].dot11rate &
519                     IEEE80211_RATE_VAL] = (uint8_t)i; /* LINT */

521         sc->sc_currates = rt;
522         sc->sc_curmode = mode;

524         /*
525          * All protection frames are transmited at 2Mb/s for
526          * 11g, otherwise at 1Mb/s.
527          * XXX select protection rate index from rate table.
528          */
529         sc->sc_protrix = (mode == ATH9K_MODE_11G ? 1 : 0);
530 }
```

```
______unchanged_portion_omitted_
```

```
924 /*
925  *  This routine performs the periodic noise floor calibration function
926  *  that is used to adjust and optimize the chip performance.  This
927  *  takes environmental changes (location, temperature) into account.
928  *  When the task is complete, it reschedules itself depending on the
929  *  appropriate interval that was calculated.
930  */
931 static void
932 arn_ani_calibrate(void *arg)

933 {
934         ieee80211com_t *ic = (ieee80211com_t *)arg;
935         struct arn_softc *sc = (struct arn_softc *)ic;
936         struct ath_hal *ah = sc->sc_ah;
937         boolean_t longcal = B_FALSE;
938         boolean_t shortcal = B_FALSE;
939         boolean_t aniflag = B_FALSE;
940         unsigned int timestamp = drv_hztousec(ddi_get_lbolt())/1000;
941         uint32_t cal_interval;

943         /*
944          * don't calibrate when we're scanning.
945          * we are most likely not on our home channel.
946          */
947         if (ic->ic_state != IEEE80211_S_RUN)
948                 goto settimer;

950         /* Long calibration runs independently of short calibration. */
951         if ((timestamp - sc->sc_ani.sc_longcal_timer) >= ATH_LONG_CALINTERVAL) {
952                 longcal = B_TRUE;
953                 ARN_DBG((ARN_DBG_CALIBRATE, "arn: "
954                     "%s: longcal @%lu\n", __func__, drv_hztousec));
955                 sc->sc_ani.sc_longcal_timer = timestamp;
956         }

958         /* Short calibration applies only while sc_caldone is FALSE */
959         if (!sc->sc_ani.sc_caldone) {
960                 if ((timestamp - sc->sc_ani.sc_shortcal_timer) >=
961                     ATH_SHORT_CALINTERVAL) {
962                         shortcal = B_TRUE;
963                         ARN_DBG((ARN_DBG_CALIBRATE, "arn: "
964                             "%s: shortcal @%lu\n",
965                             __func__, drv_hztousec));
966                         sc->sc_ani.sc_shortcal_timer = timestamp;
967                         sc->sc_ani.sc_resetcal_timer = timestamp;
968                 }
969         } else {
970                 if ((timestamp - sc->sc_ani.sc_resetcal_timer) >=
```

```
 971                     ATH_RESTART_CALINTERVAL) {
 972                         ath9k_hw_reset_calvalid(ah, ah->ah_curchan,
 973                             &sc->sc_ani.sc_caldone);
 974                         if (sc->sc_ani.sc_caldone)
 975                                 sc->sc_ani.sc_resetcal_timer = timestamp;
 976                 }
 977         }

 979         /* Verify whether we must check ANI */
 980         if ((timestamp - sc->sc_ani.sc_checkani_timer) >=
 981             ATH_ANI_POLLINTERVAL) {
 982                 aniflag = B_TRUE;
 983                 sc->sc_ani.sc_checkani_timer = timestamp;
 984         }

 986         /* Skip all processing if there's nothing to do. */
 987         if (longcal || shortcal || aniflag) {
 988                 /* Call ANI routine if necessary */
 989                 if (aniflag)
 990                         ath9k_hw_ani_monitor(ah, &sc->sc_halstats,
 991                             ah->ah_curchan);

 993                 /* Perform calibration if necessary */
 994                 if (longcal || shortcal) {
 995                         boolean_t iscaldone = B_FALSE;

 997                         if (ath9k_hw_calibrate(ah, ah->ah_curchan,
 998                             sc->sc_rx_chainmask, longcal, &iscaldone)) {
 999                                 if (longcal)
1000                                         sc->sc_ani.sc_noise_floor =
1001                                             ath9k_hw_getchan_noise(ah,
1002                                             ah->ah_curchan);

1004                                 ARN_DBG((ARN_DBG_CALIBRATE, "arn: "
1005                                     "%s: calibrate chan %u/%x nf: %d\n",
1006                                     __func__,
1007                                     ah->ah_curchan->channel,
1008                                     ah->ah_curchan->channelFlags,
1009                                     sc->sc_ani.sc_noise_floor));
1010                         } else {
1011                                 ARN_DBG((ARN_DBG_CALIBRATE, "arn: "
1012                                     "%s: calibrate chan %u/%x failed\n",
1013                                     __func__,
1014                                     ah->ah_curchan->channel,
1015                                     ah->ah_curchan->channelFlags));
1016                         }
1017                         sc->sc_ani.sc_caldone = iscaldone;
1018                 }
1019         }

1021 settimer:
1022         /*
1023          * Set timer interval based on previous results.
1024          * The interval must be the shortest necessary to satisfy ANI,
1025          * short calibration and long calibration.
1026          */
1027         cal_interval = ATH_LONG_CALINTERVAL;
1028         if (sc->sc_ah->ah_config.enable_ani)
1029                 cal_interval =
1030                     min(cal_interval, (uint32_t)ATH_ANI_POLLINTERVAL);

1032         if (!sc->sc_ani.sc_caldone)
1033                 cal_interval = min(cal_interval,
1034                     (uint32_t)ATH_SHORT_CALINTERVAL);

1036         sc->sc_scan_timer = 0;
```



```
1037         sc->sc_scan_timer = timeout(arn_ani_calibrate, (void *)sc,
1038             drv_usectohz(cal_interval * 1000));
1039 }
______unchanged_portion_omitted_

2578 static void
2579 arn_setup_ht_cap(struct arn_softc *sc)
2580 {
2581 #define ATH9K_HT_CAP_MAXRXAMPDU_65536 0x3       /* 2 ^ 16 */
2582 #define ATH9K_HT_CAP_MPDUDENSITY_8 0x6          /* 8 usec */

2631         /* LINTED E_FUNC_SET_NOT_USED */
2632         uint8_t tx_streams;
2584         uint8_t rx_streams;

2586         arn_ht_conf *ht_info = &sc->sc_ht_conf;

2588         ht_info->ht_supported = B_TRUE;

2590         /* Todo: IEEE80211_HTCAP_SMPS */
2591         ht_info->cap = IEEE80211_HTCAP_CHWIDTH40|
2592             IEEE80211_HTCAP_SHORTGI40 |
2593             IEEE80211_HTCAP_DSSSCCK40;

2595         ht_info->ampdu_factor = ATH9K_HT_CAP_MAXRXAMPDU_65536;
2596         ht_info->ampdu_density = ATH9K_HT_CAP_MPDUDENSITY_8;

2598         /* set up supported mcs set */
2599         (void) memset(&ht_info->rx_mcs_mask, 0, sizeof (ht_info->rx_mcs_mask));
2649         tx_streams = ISP2(sc->sc_ah->ah_caps.tx_chainmask) ? 1 : 2;
2600         rx_streams = ISP2(sc->sc_ah->ah_caps.rx_chainmask) ? 1 : 2;

2602         ht_info->rx_mcs_mask[0] = 0xff;
2603         if (rx_streams >= 2)
2604                 ht_info->rx_mcs_mask[1] = 0xff;
2605 }
______unchanged_portion_omitted_

3449 DDI_DEFINE_STREAM_OPS(arn_dev_ops, nulldev, nulldev, arn_attach, arn_detach,
3450     nodev, NULL, D_MP, NULL, arn_quiesce);

3452 static struct modldrv arn_modldrv = {
3453         &mod_driverops, /* Type of module.  This one is a driver */
3454         "Atheros 9000 series driver", /* short description */
3504         "arn-Atheros 9000 series driver:2.0", /* short description */
3455         &arn_dev_ops /* driver specific ops */
3456 };
______unchanged_portion_omitted_
```

```
**********************************************************
   61994 Wed Jun 29 14:08:33 2016
new/usr/src/uts/common/io/arn/arn_xmit.c
7154 arn(7D) walks out of bounds when byteswapping the 4K eeprom
7152 weird condition in arn(7D) needs clarification
7153 delete unused code in arn(7D)
7155 arn(7D) should include the mac fields in the eeprom enumeration
**********************************************************
______unchanged_portion_omitted_

 282 /*
 283  * TODO: For frame(s) that are in the retry state, we will reuse the
 284  * sequence number(s) without setting the retry bit. The
 285  * alternative is to give up on these and BAR the receiver's window
 286  * forward.
 287  */
 288 static void
 289 arn_tid_drain(struct arn_softc *sc,
 290     struct ath_txq *txq,
 291     struct ath_atx_tid *tid)

 292 {
 293         struct ath_buf *bf;

 295         list_t list;
 296         list_create(&list, sizeof (struct ath_buf),
 297             offsetof(struct ath_buf, bf_node));

 299         for (;;) {
 300                 if (list_empty(&tid->buf_q))
 301                         break;

 303                 bf = list_head(&tid->buf_q);
 304                 list_remove(&tid->buf_q, bf);
 305                 list_insert_tail(&list, bf);

 307                 if (bf_isretried(bf))
 308                         arn_tx_update_baw(sc, tid, bf->bf_seqno);

 310                 mutex_enter(&txq->axq_lock);
 311                 arn_tx_complete_buf(sc, bf, &list, 0, 0);
 312                 mutex_exit(&txq->axq_lock);
 313         }

 315         tid->seq_next = tid->seq_start;
 316         tid->baw_tail = tid->baw_head;
 317 }
______unchanged_portion_omitted_
1147 #endif /* ARN_TX_AGGREGATION */

1150 /*
1151  * ath_pkt_dur - compute packet duration (NB: not NAV)
1152  * rix - rate index
1153  * pktlen - total bytes (delims + data + fcs + pads + pad delims)
1154  * width  - 0 for 20 MHz, 1 for 40 MHz
1155  * half_gi - to use 4us v/s 3.6 us for symbol time
1156  */

1158 static uint32_t
1159 /* LINTED E_STATIC_UNUSED */
1160 arn_pkt_duration(struct arn_softc *sc, uint8_t rix, struct ath_buf *bf,
1161     int width, int half_gi, boolean_t shortPreamble)
1162 {
1163         struct ath_rate_table *rate_table = sc->sc_currates;
1164         uint32_t nbits, nsymbits, duration, nsymbols;
1165         uint8_t rc;
```



```
1166         int streams, pktlen;

1168         pktlen = bf_isaggr(bf) ? bf->bf_al : bf->bf_frmlen;
1169         rc = rate_table->info[rix].ratecode;

1171         /* for legacy rates, use old function to compute packet duration */
1172         if (!IS_HT_RATE(rc))
1173                 return (ath9k_hw_computetxtime(sc->sc_ah, rate_table, pktlen,
1174                     rix, shortPreamble));

1176         /* find number of symbols: PLCP + data */
1177         nbits = (pktlen << 3) + OFDM_PLCP_BITS;
1178         nsymbits = bits_per_symbol[HT_RC_2_MCS(rc)][width];
1179         nsymbols = (nbits + nsymbits - 1) / nsymbits;

1181         if (!half_gi)
1182                 duration = SYMBOL_TIME(nsymbols);
1183         else
1184                 duration = SYMBOL_TIME_HALFGI(nsymbols);

1186         /* addup duration for legacy/ht training and signal fields */
1187         streams = HT_RC_2_STREAMS(rc);
1188         duration += L_STF + L_LTF + L_SIG + HT_SIG + HT_STF + HT_LTF(streams);

1190         return (duration);
1191 }

1149 static struct ath_buf *
1150 arn_tx_get_buffer(struct arn_softc *sc)
1151 {
1152         struct ath_buf *bf = NULL;

1154         mutex_enter(&sc->sc_txbuflock);
1155         bf = list_head(&sc->sc_txbuf_list);
1156         /* Check if a tx buffer is available */
1157         if (bf != NULL)
1158                 list_remove(&sc->sc_txbuf_list, bf);
1159         if (list_empty(&sc->sc_txbuf_list)) {
1160                 ARN_DBG((ARN_DBG_XMIT, "arn: arn_tx(): "
1161                     "stop queue\n"));
1162                 sc->sc_stats.ast_tx_qstop++;
1163         }
1164         mutex_exit(&sc->sc_txbuflock);

1166         return (bf);
1167 }
______unchanged_portion_omitted_

1288 /* Rate module function to set rate related fields in tx descriptor */
1289 static void
1290 ath_buf_set_rate(struct arn_softc *sc,
1291     struct ath_buf *bf,
1292     struct ieee80211_frame *wh)
1335 struct ath_buf *bf,
1336 struct ieee80211_frame *wh)
1293 {
1294         struct ath_hal *ah = sc->sc_ah;
1295         struct ath_rate_table *rt;
1296         struct ath_desc *ds = bf->bf_desc;
1297         struct ath_desc *lastds = bf->bf_desc; /* temp workground */
1298         struct ath9k_11n_rate_series series[4];
1299         struct ath9k_tx_rate *rates;
1300         int i, flags, rtsctsena = 0;
1301         uint32_t ctsduration = 0;
1302         uint8_t rix = 0, cix, ctsrate = 0;
```

```
1304         (void) memset(series, 0, sizeof (struct ath9k_11n_rate_series) * 4);

1306         rates = bf->rates;

1308         if (IEEE80211_HAS_MOREFRAGS(wh) ||
1309             wh->i_seq[0] & IEEE80211_SEQ_FRAG_MASK) {
1310                 rates[1].count = rates[2].count = rates[3].count = 0;
1311                 rates[1].idx = rates[2].idx = rates[3].idx = 0;
1312                 rates[0].count = ATH_TXMAXTRY;
1313         }

1315         /* get the cix for the lowest valid rix */
1316         rt = sc->sc_currates;
1317         for (i = 3; i >= 0; i--) {
1318                 if (rates[i].count && (rates[i].idx >= 0)) {
1319                         rix = rates[i].idx;
1320                         break;
1321                 }
1322         }

1324         flags = (bf->bf_flags & (ATH9K_TXDESC_RTSENA | ATH9K_TXDESC_CTSENA));
1325         cix = rt->info[rix].ctrl_rate;

1327         /*
1328          * If 802.11g protection is enabled, determine whether to use RTS/CTS or
1329          * just CTS.  Note that this is only done for OFDM/HT unicast frames.
1330          */
1331         if (sc->sc_protmode != PROT_M_NONE &&
1332             !(bf->bf_flags & ATH9K_TXDESC_NOACK) &&
1333             (rt->info[rix].phy == WLAN_RC_PHY_OFDM ||
1334             WLAN_RC_PHY_HT(rt->info[rix].phy))) {
1335                 if (sc->sc_protmode == PROT_M_RTSCTS)
1336                         flags = ATH9K_TXDESC_RTSENA;
1337                 else if (sc->sc_protmode == PROT_M_CTSONLY)
1338                         flags = ATH9K_TXDESC_CTSENA;

1340                 cix = rt->info[sc->sc_protrix].ctrl_rate;
1341                 rtsctsena = 1;
1342         }

1344         /*
1345          * For 11n, the default behavior is to enable RTS for hw retried frames.
1346          * We enable the global flag here and let rate series flags determine
1347          * which rates will actually use RTS.
1348          */
1349         if ((ah->ah_caps.hw_caps & ATH9K_HW_CAP_HT) && bf_isdata(bf)) {
1350                 /* 802.11g protection not needed, use our default behavior */
1351                 if (!rtsctsena)
1352                         flags = ATH9K_TXDESC_RTSENA;
1353         }

1355         /* Set protection if aggregate protection on */
1356         if (sc->sc_config.ath_aggr_prot &&
1357             (!bf_isaggr(bf) || (bf_isaggr(bf) && bf->bf_al < 8192))) {
1358                 flags = ATH9K_TXDESC_RTSENA;
1359                 cix = rt->info[sc->sc_protrix].ctrl_rate;
1360                 rtsctsena = 1;
1361         }

1363         /* For AR5416 - RTS cannot be followed by a frame larger than 8K */
1364         if (bf_isaggr(bf) && (bf->bf_al > ah->ah_caps.rts_aggr_limit))
1365                 flags &= ~(ATH9K_TXDESC_RTSENA);

1367         /*
1368          * CTS transmit rate is derived from the transmit rate by looking in the
1369          * h/w rate table.  We must also factor in whether or not a short
```



```
1370          * preamble is to be used. NB: cix is set above where RTS/CTS is enabled
1371          */
1372         ctsrate = rt->info[cix].ratecode |
1373             (bf_isshpreamble(bf) ? rt->info[cix].short_preamble : 0);

1375         for (i = 0; i < 4; i++) {
1376                 if (!rates[i].count || (rates[i].idx < 0))
1377                         continue;

1379                 rix = rates[i].idx;

1381                 series[i].Rate = rt->info[rix].ratecode |
1382                     (bf_isshpreamble(bf) ?
1383                     rt->info[rix].short_preamble : 0);

1385                 series[i].Tries = rates[i].count;

1387                 series[i].RateFlags =
1388                     ((rates[i].flags & ATH9K_TX_RC_USE_RTS_CTS) ?
1389                     ATH9K_RATESERIES_RTS_CTS : 0) |
1390                     ((rates[i].flags & ATH9K_TX_RC_40_MHZ_WIDTH) ?
1391                     ATH9K_RATESERIES_2040 : 0) |
1392                     ((rates[i].flags & ATH9K_TX_RC_SHORT_GI) ?
1393                     ATH9K_RATESERIES_HALFGI : 0);

1395                 series[i].PktDuration = ath_pkt_duration(sc, rix, bf,
1396                     (rates[i].flags & ATH9K_TX_RC_40_MHZ_WIDTH) != 0,
1397                     (rates[i].flags & ATH9K_TX_RC_SHORT_GI),
1398                     bf_isshpreamble(bf));

1400                 series[i].ChSel = sc->sc_tx_chainmask;

1402                 if (rtsctsena)
1403                         series[i].RateFlags |= ATH9K_RATESERIES_RTS_CTS;

1405                 ARN_DBG((ARN_DBG_RATE,
1406                     "series[%d]--flags & ATH9K_TX_RC_USE_RTS_CTS = %08x"
1407                     "--flags & ATH9K_TX_RC_40_MHZ_WIDTH = %08x"
1408                     "--flags & ATH9K_TX_RC_SHORT_GI = %08x\n",
1409                     rates[i].flags & ATH9K_TX_RC_USE_RTS_CTS,
1410                     rates[i].flags & ATH9K_TX_RC_40_MHZ_WIDTH,
1411                     rates[i].flags & ATH9K_TX_RC_SHORT_GI));

1413                 ARN_DBG((ARN_DBG_RATE,
1414                     "series[%d]:"
1415                     "dot11rate:%d"
1416                     "index:%d"
1417                     "retry count:%d\n",
1418                     i,
1419                     (rt->info[rates[i].idx].ratekbps)/1000,
1420                     rates[i].idx,
1421                     rates[i].count));
1422         }

1424         /* set dur_update_en for l-sig computation except for PS-Poll frames */
1425         ath9k_hw_set11n_ratescenario(ah, ds, lastds, !bf_ispspoll(bf),
1426             ctsrate, ctsduration,
1427             series, 4, flags);

1429         if (sc->sc_config.ath_aggr_prot && flags)
1430                 ath9k_hw_set11n_burstduration(ah, ds, 8192);
1431 }
```

unchanged_portion_omitted_

```
**********************************************************
    2155 Wed Jun 29 14:08:34 2016
new/usr/src/uts/intel/arn/Makefile
7154 arn(7D) walks out of bounds when byteswapping the 4K eeprom
7152 weird condition in arn(7D) needs clarification
7153 delete unused code in arn(7D)
7155 arn(7D) should include the mac fields in the eeprom enumeration
**********************************************************
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#


#
#       This file makes the atheros IEEE 802.11n driver for an intel system
#
#       intel architecture dependent
#

#
#       Path to the base of the uts directory tree (usually /usr/src/uts).
#
UTSBASE = ../..
#
#       Define the module and object file sets.
#
MODULE          = arn
OBJECTS         = $(ARN_OBJS:%=$(OBJS_DIR)/%)
LINTS           = $(ARN_OBJS:%.o=$(LINTS_DIR)/%.ln)
ROOTMODULE      = $(ROOT_DRV_DIR)/$(MODULE)

#
#       Include common rules.
#
include $(UTSBASE)/intel/Makefile.intel

#
#       Define targets
#
ALL_TARGET      = $(BINARY)
LINT_TARGET     = $(MODULE).lint
INSTALL_TARGET  = $(BINARY) $(ROOTMODULE)

#
#       Driver depends on GLDv3 & wifi kernel support module.
```



```
#
LDFLAGS         += -dy -Nmisc/mac -Nmisc/net80211

LINTTAGS        += -erroff=E_BAD_PTR_CAST_ALIGN

CERRWARN        += -_gcc=-Wno-unused-variable
CERRWARN        += -_gcc=-Wno-unused-function
CERRWARN        += -_gcc=-Wno-uninitialized
CERRWARN        += -_gcc=-Wno-char-subscripts
CERRWARN        += -_gcc=-Wno-empty-body

#
#       Default build targets.
#
.KEEP_STATE:

def:            $(DEF_DEPS)

all:            $(ALL_DEPS)

clean:          $(CLEAN_DEPS)

clobber:        $(CLOBBER_DEPS)

lint:           $(LINT_DEPS)

modlintlib:     $(MODLINTLIB_DEPS)

clean.lint:     $(CLEAN_LINT_DEPS)

install:        $(INSTALL_DEPS)

#
#       Include common targets.
#
include $(UTSBASE)/intel/Makefile.targ

#
#       If you have any special case that general
#       Makefile rules don't serve for you, just do
#       it yourself.
#
```