```
**********************************************************
   60242 Thu Feb  7 16:20:16 2013
new/usr/src/uts/common/io/igb/igb_82575.c
3534 Disable EEE support in igb for I350
**********************************************************
```

```
/*
 * 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 (c) 2007-2012 Intel Corporation. All rights reserved.
 */

/*
 * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2013, Nexenta Systems, Inc. All rights reserved.
 */

/* IntelVersion: 1.146.2.2 v3_3_14_3_BHSW1 */

/*
 * 82575EB Gigabit Network Connection
 * 82575EB Gigabit Backplane Connection
 * 82575GB Gigabit Network Connection
 * 82576 Gigabit Network Connection
 * 82576 Quad Port Gigabit Mezzanine Adapter
 */

#include "igb_api.h"

static s32 e1000_init_phy_params_82575(struct e1000_hw *hw);
static s32 e1000_init_nvm_params_82575(struct e1000_hw *hw);
static s32 e1000_init_mac_params_82575(struct e1000_hw *hw);
static s32 e1000_acquire_phy_82575(struct e1000_hw *hw);
static void e1000_release_phy_82575(struct e1000_hw *hw);
static s32 e1000_acquire_nvm_82575(struct e1000_hw *hw);
static void e1000_release_nvm_82575(struct e1000_hw *hw);
static s32 e1000_check_for_link_82575(struct e1000_hw *hw);
static s32 e1000_get_cfg_done_82575(struct e1000_hw *hw);
static s32 e1000_get_link_up_info_82575(struct e1000_hw *hw, u16 *speed,
    u16 *duplex);
static s32 e1000_init_hw_82575(struct e1000_hw *hw);
static s32 e1000_phy_hw_reset_sgmii_82575(struct e1000_hw *hw);
static s32 e1000_read_phy_reg_sgmii_82575(struct e1000_hw *hw, u32 offset,
    u16 *data);
static s32 e1000_reset_hw_82575(struct e1000_hw *hw);
static s32 e1000_reset_hw_82580(struct e1000_hw *hw);
static s32 e1000_read_phy_reg_82580(struct e1000_hw *hw, u32 offset,
    u16 *data);
```



```
static s32 e1000_write_phy_reg_82580(struct e1000_hw *hw, u32 offset,
    u16 data);
static s32 e1000_set_d0_lplu_state_82575(struct e1000_hw *hw,
    bool active);
static s32 e1000_setup_copper_link_82575(struct e1000_hw *hw);
static s32 e1000_setup_serdes_link_82575(struct e1000_hw *hw);
static s32 e1000_valid_led_default_82575(struct e1000_hw *hw, u16 *data);
static s32 e1000_write_phy_reg_sgmii_82575(struct e1000_hw *hw,
    u32 offset, u16 data);
static void e1000_clear_hw_cntrs_82575(struct e1000_hw *hw);
static s32 e1000_acquire_swfw_sync_82575(struct e1000_hw *hw, u16 mask);
static s32 e1000_get_pcs_speed_and_duplex_82575(struct e1000_hw *hw,
    u16 *speed, u16 *duplex);
static s32 e1000_get_phy_id_82575(struct e1000_hw *hw);
static void e1000_release_swfw_sync_82575(struct e1000_hw *hw, u16 mask);
static bool e1000_sgmii_active_82575(struct e1000_hw *hw);
static s32 e1000_reset_init_script_82575(struct e1000_hw *hw);
static s32 e1000_read_mac_addr_82575(struct e1000_hw *hw);
static void e1000_power_down_phy_copper_82575(struct e1000_hw *hw);
static void e1000_shutdown_serdes_link_82575(struct e1000_hw *hw);
static s32 e1000_set_pcie_completion_timeout(struct e1000_hw *hw);

static s32 e1000_update_nvm_checksum_with_offset(struct e1000_hw *hw,
    u16 offset);
static s32 e1000_validate_nvm_checksum_with_offset(struct e1000_hw *hw,
    u16 offset);
static s32 e1000_validate_nvm_checksum_i350(struct e1000_hw *hw);
static s32 e1000_update_nvm_checksum_i350(struct e1000_hw *hw);
static void e1000_write_vfta_i350(struct e1000_hw *hw, u32 offset, u32 value);
static void e1000_clear_vfta_i350(struct e1000_hw *hw);

static const u16 e1000_82580_rxpbs_table[] =
        {36, 72, 144, 1, 2, 4, 8, 16, 35, 70, 140};
#define E1000_82580_RXPBS_TABLE_SIZE \
        (sizeof (e1000_82580_rxpbs_table)/sizeof (u16))

/*
 * e1000_init_phy_params_82575 - Init PHY func ptrs.
 * @hw: pointer to the HW structure
 */
static s32
e1000_init_phy_params_82575(struct e1000_hw *hw)
{
        struct e1000_phy_info *phy = &hw->phy;
        s32 ret_val = E1000_SUCCESS;

        DEBUGFUNC("e1000_init_phy_params_82575");

        if (hw->phy.media_type != e1000_media_type_copper) {
                phy->type = e1000_phy_none;
                goto out;
        }

        phy->ops.power_up = e1000_power_up_phy_copper;
        phy->ops.power_down = e1000_power_down_phy_copper_82575;

        phy->autoneg_mask = AUTONEG_ADVERTISE_SPEED_DEFAULT;
        phy->reset_delay_us = 100;

        phy->ops.acquire = e1000_acquire_phy_82575;
        phy->ops.check_reset_block = e1000_check_reset_block_generic;
        phy->ops.commit = e1000_phy_sw_reset_generic;
        phy->ops.get_cfg_done = e1000_get_cfg_done_82575;
        phy->ops.release = e1000_release_phy_82575;

        if (e1000_sgmii_active_82575(hw)) {
```

```
128                 phy->ops.reset = e1000_phy_hw_reset_sgmii_82575;
129                 phy->ops.read_reg = e1000_read_phy_reg_sgmii_82575;
130                 phy->ops.write_reg = e1000_write_phy_reg_sgmii_82575;
131         } else if (hw->mac.type == e1000_82580) {
132                 phy->ops.reset = e1000_phy_hw_reset_generic;
133                 phy->ops.read_reg = e1000_read_phy_reg_82580;
134                 phy->ops.write_reg = e1000_write_phy_reg_82580;
135         } else {
136                 phy->ops.reset = e1000_phy_hw_reset_generic;
137                 phy->ops.read_reg = e1000_read_phy_reg_igp;
138                 phy->ops.write_reg = e1000_write_phy_reg_igp;
139         }

141         /* Set phy->phy_addr and phy->id. */
142         ret_val = e1000_get_phy_id_82575(hw);

144         /* Verify phy id and set remaining function pointers */
145         switch (phy->id) {
146         case M88E1111_I_PHY_ID:
147                 phy->type = e1000_phy_m88;
148                 phy->ops.check_polarity = e1000_check_polarity_m88;
149                 phy->ops.get_info = e1000_get_phy_info_m88;
150                 phy->ops.get_cable_length = e1000_get_cable_length_m88;
151                 phy->ops.force_speed_duplex = e1000_phy_force_speed_duplex_m88;
152                 break;
153         case IGP03E1000_E_PHY_ID:
154         case IGP04E1000_E_PHY_ID:
155                 phy->type = e1000_phy_igp_3;
156                 phy->ops.check_polarity = e1000_check_polarity_igp;
157                 phy->ops.get_info = e1000_get_phy_info_igp;
158                 phy->ops.get_cable_length = e1000_get_cable_length_igp_2;
159                 phy->ops.force_speed_duplex = e1000_phy_force_speed_duplex_igp;
160                 phy->ops.set_d0_lplu_state = e1000_set_d0_lplu_state_82575;
161                 phy->ops.set_d3_lplu_state = e1000_set_d3_lplu_state_generic;
162                 break;
163         case I82580_I_PHY_ID:
164         case I350_I_PHY_ID:
165                 phy->type = e1000_phy_82580;
166                 phy->ops.check_polarity = e1000_check_polarity_82577;
167                 phy->ops.force_speed_duplex =
168                     e1000_phy_force_speed_duplex_82577;
169                 phy->ops.get_cable_length = e1000_get_cable_length_82577;
170                 phy->ops.get_info = e1000_get_phy_info_82577;
171                 break;
172         default:
173                 ret_val = -E1000_ERR_PHY;
174                 goto out;
175         }

177 out:
178         return (ret_val);
179 }
```

*unchanged_portion_omitted_*

```
250 /*
251  * e1000_init_mac_params_82575 - Init MAC func ptrs.
252  * @hw: pointer to the HW structure
253  */
254 static s32
255 e1000_init_mac_params_82575(struct e1000_hw *hw)
256 {
257         struct e1000_mac_info *mac = &hw->mac;
258         struct e1000_dev_spec_82575 *dev_spec = &hw->dev_spec._82575;
259         u32 ctrl_ext = 0;

261         DEBUGFUNC("e1000_init_mac_params_82575");
```



```
263         /* Set media type */
264         /*
265          * The 82575 uses bits 22:23 for link mode. The mode can be changed
266          * based on the EEPROM. We cannot rely upon device ID. There
267          * is no distinguishable difference between fiber and internal
268          * SerDes mode on the 82575. There can be an external PHY attached
269          * on the SGMII interface. For this, we'll set sgmii_active to true.
270          */
271         hw->phy.media_type = e1000_media_type_copper;
272         dev_spec->sgmii_active = false;

274         ctrl_ext = E1000_READ_REG(hw, E1000_CTRL_EXT);
275         switch (ctrl_ext & E1000_CTRL_EXT_LINK_MODE_MASK) {
276         case E1000_CTRL_EXT_LINK_MODE_SGMII:
277                 dev_spec->sgmii_active = true;
278                 ctrl_ext |= E1000_CTRL_I2C_ENA;
279                 break;
280         case E1000_CTRL_EXT_LINK_MODE_1000BASE_KX:
281         case E1000_CTRL_EXT_LINK_MODE_PCIE_SERDES:
282                 hw->phy.media_type = e1000_media_type_internal_serdes;
283                 ctrl_ext |= E1000_CTRL_I2C_ENA;
284                 break;
285         default:
286                 ctrl_ext &= ~E1000_CTRL_I2C_ENA;
287                 break;
288         }

290         E1000_WRITE_REG(hw, E1000_CTRL_EXT, ctrl_ext);

292         /*
293          * if using i2c make certain the MDICNFG register is cleared to prevent
294          * communications from being misrouted to the mdic registers
295          */
296         if ((ctrl_ext & E1000_CTRL_I2C_ENA) && (hw->mac.type == e1000_82580))
297                 E1000_WRITE_REG(hw, E1000_MDICNFG, 0);

299         /* Set mta register count */
300         mac->mta_reg_count = 128;
301         /* Set uta register count */
302         mac->uta_reg_count = (hw->mac.type == e1000_82575) ? 0 : 128;
303         /* Set rar entry count */
304         mac->rar_entry_count = E1000_RAR_ENTRIES_82575;
305         if (mac->type == e1000_82576)
306                 mac->rar_entry_count = E1000_RAR_ENTRIES_82576;
307         if (mac->type == e1000_82580)
308                 mac->rar_entry_count = E1000_RAR_ENTRIES_82580;
309         if (mac->type == e1000_i350) {
310                 mac->rar_entry_count = E1000_RAR_ENTRIES_I350;
311                 /* Disable EEE default settings for i350 */
312                 dev_spec->eee_disable = B_TRUE;
310                 /* Enable EEE default settings for i350 */
311                 dev_spec->eee_disable = B_FALSE;
313         }
314         /* Set if part includes ASF firmware */
315         mac->asf_firmware_present = true;
316         /* Set if manageability features are enabled. */
317         mac->arc_subsystem_valid =
318             (E1000_READ_REG(hw, E1000_FWSM) & E1000_FWSM_MODE_MASK)
319             ? true : false;

321         /* Function pointers */

323         /* bus type/speed/width */
324         mac->ops.get_bus_info = e1000_get_bus_info_pcie_generic;
325         /* reset */
```

```
326         if (mac->type == e1000_82580)
327                 mac->ops.reset_hw = e1000_reset_hw_82580;
328         else
329                 mac->ops.reset_hw = e1000_reset_hw_82575;
330         /* hw initialization */
331         mac->ops.init_hw = e1000_init_hw_82575;
332         /* link setup */
333         mac->ops.setup_link = e1000_setup_link_generic;
334         /* physical interface link setup */
335         mac->ops.setup_physical_interface =
336             (hw->phy.media_type == e1000_media_type_copper)
337             ? e1000_setup_copper_link_82575
338             : e1000_setup_serdes_link_82575;
339         /* physical interface shutdown */
340         mac->ops.shutdown_serdes = e1000_shutdown_serdes_link_82575;
341         /* check for link */
342         mac->ops.check_for_link = e1000_check_for_link_82575;
343         /* receive address register setting */
344         mac->ops.rar_set = e1000_rar_set_generic;
345         /* read mac address */
346         mac->ops.read_mac_addr = e1000_read_mac_addr_82575;
347         /* multicast address update */
348         mac->ops.update_mc_addr_list = e1000_update_mc_addr_list_generic;

350         if (hw->mac.type == e1000_i350) {
351                 /* writing VFTA */
352                 mac->ops.write_vfta = e1000_write_vfta_i350;
353                 /* clearing VFTA */
354                 mac->ops.clear_vfta = e1000_clear_vfta_i350;
355         } else {
356                 /* writing VFTA */
357                 mac->ops.write_vfta = e1000_write_vfta_generic;
358                 /* clearing VFTA */
359                 mac->ops.clear_vfta = e1000_clear_vfta_generic;
360         }
361         /* setting MTA */
362         mac->ops.mta_set = e1000_mta_set_generic;
363         /* ID LED init */
364         mac->ops.id_led_init = e1000_id_led_init_generic;
365         /* blink LED */
366         mac->ops.blink_led = e1000_blink_led_generic;
367         /* setup LED */
368         mac->ops.setup_led = e1000_setup_led_generic;
369         /* cleanup LED */
370         mac->ops.cleanup_led = e1000_cleanup_led_generic;
371         /* turn on/off LED */
372         mac->ops.led_on = e1000_led_on_generic;
373         mac->ops.led_off = e1000_led_off_generic;
374         /* clear hardware counters */
375         mac->ops.clear_hw_cntrs = e1000_clear_hw_cntrs_82575;
376         /* link info */
377         mac->ops.get_link_up_info = e1000_get_link_up_info_82575;

379         /* set lan id for port to determine which phy lock to use */
380         hw->mac.ops.set_lan_id(hw);

382         return (E1000_SUCCESS);
383 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   37829 Thu Feb  7 16:20:17 2013
new/usr/src/uts/common/io/igb/igb_gld.c
3534 Disable EEE support in igb for I350
**********************************************************
```

```
/*
 * 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(c) 2007-2010 Intel Corporation. All rights reserved.
 */

/*
 * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2013, Nexenta Systems, Inc. All rights reserved.
 */

#include "igb_sw.h"

int
igb_m_stat(void *arg, uint_t stat, uint64_t *val)
{
        igb_t *igb = (igb_t *)arg;
        struct e1000_hw *hw = &igb->hw;
        igb_stat_t *igb_ks;
        uint32_t low_val, high_val;

        igb_ks = (igb_stat_t *)igb->igb_ks->ks_data;

        mutex_enter(&igb->gen_lock);

        if (igb->igb_state & IGB_SUSPENDED) {
                mutex_exit(&igb->gen_lock);
                return (ECANCELED);
        }

        switch (stat) {
        case MAC_STAT_IFSPEED:
                *val = igb->link_speed * 1000000ull;
                break;

        case MAC_STAT_MULTIRCV:
                igb_ks->mprc.value.ui64 +=
                    E1000_READ_REG(hw, E1000_MPRC);
                *val = igb_ks->mprc.value.ui64;
                break;

        case MAC_STAT_BRDCSTRCV:
```



```
                igb_ks->bprc.value.ui64 +=
                    E1000_READ_REG(hw, E1000_BPRC);
                *val = igb_ks->bprc.value.ui64;
                break;

        case MAC_STAT_MULTIXMT:
                igb_ks->mptc.value.ui64 +=
                    E1000_READ_REG(hw, E1000_MPTC);
                *val = igb_ks->mptc.value.ui64;
                break;

        case MAC_STAT_BRDCSTXMT:
                igb_ks->bptc.value.ui64 +=
                    E1000_READ_REG(hw, E1000_BPTC);
                *val = igb_ks->bptc.value.ui64;
                break;

        case MAC_STAT_NORCVBUF:
                igb_ks->rnbc.value.ui64 +=
                    E1000_READ_REG(hw, E1000_RNBC);
                *val = igb_ks->rnbc.value.ui64;
                break;

        case MAC_STAT_IERRORS:
                igb_ks->rxerrc.value.ui64 +=
                    E1000_READ_REG(hw, E1000_RXERRC);
                igb_ks->algnerrc.value.ui64 +=
                    E1000_READ_REG(hw, E1000_ALGNERRC);
                igb_ks->rlec.value.ui64 +=
                    E1000_READ_REG(hw, E1000_RLEC);
                igb_ks->crcerrs.value.ui64 +=
                    E1000_READ_REG(hw, E1000_CRCERRS);
                igb_ks->cexterr.value.ui64 +=
                    E1000_READ_REG(hw, E1000_CEXTERR);
                *val = igb_ks->rxerrc.value.ui64 +
                    igb_ks->algnerrc.value.ui64 +
                    igb_ks->rlec.value.ui64 +
                    igb_ks->crcerrs.value.ui64 +
                    igb_ks->cexterr.value.ui64;
                break;

        case MAC_STAT_NOXMTBUF:
                *val = 0;
                break;

        case MAC_STAT_OERRORS:
                igb_ks->ecol.value.ui64 +=
                    E1000_READ_REG(hw, E1000_ECOL);
                *val = igb_ks->ecol.value.ui64;
                break;

        case MAC_STAT_COLLISIONS:
                igb_ks->colc.value.ui64 +=
                    E1000_READ_REG(hw, E1000_COLC);
                *val = igb_ks->colc.value.ui64;
                break;

        case MAC_STAT_RBYTES:
                /*
                 * The 64-bit register will reset whenever the upper
                 * 32 bits are read. So we need to read the lower
                 * 32 bits first, then read the upper 32 bits.
                 */
                low_val = E1000_READ_REG(hw, E1000_TORL);
                high_val = E1000_READ_REG(hw, E1000_TORH);
                igb_ks->tor.value.ui64 +=
```

```
128                     (uint64_t)high_val << 32 | (uint64_t)low_val;
129                 *val = igb_ks->tor.value.ui64;
130                 break;
131
132         case MAC_STAT_IPACKETS:
133                 igb_ks->tpr.value.ui64 +=
134                     E1000_READ_REG(hw, E1000_TPR);
135                 *val = igb_ks->tpr.value.ui64;
136                 break;
137
138         case MAC_STAT_OBYTES:
139                 /*
140                  * The 64-bit register will reset whenever the upper
141                  * 32 bits are read. So we need to read the lower
142                  * 32 bits first, then read the upper 32 bits.
143                  */
144                 low_val = E1000_READ_REG(hw, E1000_TOTL);
145                 high_val = E1000_READ_REG(hw, E1000_TOTH);
146                 igb_ks->tot.value.ui64 +=
147                     (uint64_t)high_val << 32 | (uint64_t)low_val;
148                 *val = igb_ks->tot.value.ui64;
149                 break;
150
151         case MAC_STAT_OPACKETS:
152                 igb_ks->tpt.value.ui64 +=
153                     E1000_READ_REG(hw, E1000_TPT);
154                 *val = igb_ks->tpt.value.ui64;
155                 break;
156
157         /* RFC 1643 stats */
158         case ETHER_STAT_ALIGN_ERRORS:
159                 igb_ks->algnerrc.value.ui64 +=
160                     E1000_READ_REG(hw, E1000_ALGNERRC);
161                 *val = igb_ks->algnerrc.value.ui64;
162                 break;
163
164         case ETHER_STAT_FCS_ERRORS:
165                 igb_ks->crcerrs.value.ui64 +=
166                     E1000_READ_REG(hw, E1000_CRCERRS);
167                 *val = igb_ks->crcerrs.value.ui64;
168                 break;
169
170         case ETHER_STAT_FIRST_COLLISIONS:
171                 igb_ks->scc.value.ui64 +=
172                     E1000_READ_REG(hw, E1000_SCC);
173                 *val = igb_ks->scc.value.ui64;
174                 break;
175
176         case ETHER_STAT_MULTI_COLLISIONS:
177                 igb_ks->mcc.value.ui64 +=
178                     E1000_READ_REG(hw, E1000_MCC);
179                 *val = igb_ks->mcc.value.ui64;
180                 break;
181
182         case ETHER_STAT_SQE_ERRORS:
183                 igb_ks->sec.value.ui64 +=
184                     E1000_READ_REG(hw, E1000_SEC);
185                 *val = igb_ks->sec.value.ui64;
186                 break;
187
188         case ETHER_STAT_DEFER_XMTS:
189                 igb_ks->dc.value.ui64 +=
190                     E1000_READ_REG(hw, E1000_DC);
191                 *val = igb_ks->dc.value.ui64;
192                 break;
```



```
194         case ETHER_STAT_TX_LATE_COLLISIONS:
195                 igb_ks->latecol.value.ui64 +=
196                     E1000_READ_REG(hw, E1000_LATECOL);
197                 *val = igb_ks->latecol.value.ui64;
198                 break;
199
200         case ETHER_STAT_EX_COLLISIONS:
201                 igb_ks->ecol.value.ui64 +=
202                     E1000_READ_REG(hw, E1000_ECOL);
203                 *val = igb_ks->ecol.value.ui64;
204                 break;
205
206         case ETHER_STAT_MACXMT_ERRORS:
207                 igb_ks->ecol.value.ui64 +=
208                     E1000_READ_REG(hw, E1000_ECOL);
209                 *val = igb_ks->ecol.value.ui64;
210                 break;
211
212         case ETHER_STAT_CARRIER_ERRORS:
213                 igb_ks->cexterr.value.ui64 +=
214                     E1000_READ_REG(hw, E1000_CEXTERR);
215                 *val = igb_ks->cexterr.value.ui64;
216                 break;
217
218         case ETHER_STAT_TOOLONG_ERRORS:
219                 igb_ks->roc.value.ui64 +=
220                     E1000_READ_REG(hw, E1000_ROC);
221                 *val = igb_ks->roc.value.ui64;
222                 break;
223
224         case ETHER_STAT_MACRCV_ERRORS:
225                 igb_ks->rxerrc.value.ui64 +=
226                     E1000_READ_REG(hw, E1000_RXERRC);
227                 *val = igb_ks->rxerrc.value.ui64;
228                 break;
229
230         /* MII/GMII stats */
231         case ETHER_STAT_XCVR_ADDR:
232                 /* The Internal PHY's MDI address for each MAC is 1 */
233                 *val = 1;
234                 break;
235
236         case ETHER_STAT_XCVR_ID:
237                 *val = hw->phy.id | hw->phy.revision;
238                 break;
239
240         case ETHER_STAT_XCVR_INUSE:
241                 switch (igb->link_speed) {
242                 case SPEED_1000:
243                         *val =
244                             (hw->phy.media_type == e1000_media_type_copper) ?
245                             XCVR_1000T : XCVR_1000X;
246                         break;
247                 case SPEED_100:
248                         *val =
249                             (hw->phy.media_type == e1000_media_type_copper) ?
250                             (igb->param_100t4_cap == 1) ?
251                             XCVR_100T4 : XCVR_100T2 : XCVR_100X;
252                         break;
253                 case SPEED_10:
254                         *val = XCVR_10;
255                         break;
256                 default:
257                         *val = XCVR_NONE;
258                         break;
259                 }
```

```
260                 break;

262         case ETHER_STAT_CAP_1000FDX:
263                 *val = igb->param_1000fdx_cap;
264                 break;

266         case ETHER_STAT_CAP_1000HDX:
267                 *val = igb->param_1000hdx_cap;
268                 break;

270         case ETHER_STAT_CAP_100FDX:
271                 *val = igb->param_100fdx_cap;
272                 break;

274         case ETHER_STAT_CAP_100HDX:
275                 *val = igb->param_100hdx_cap;
276                 break;

278         case ETHER_STAT_CAP_10FDX:
279                 *val = igb->param_10fdx_cap;
280                 break;

282         case ETHER_STAT_CAP_10HDX:
283                 *val = igb->param_10hdx_cap;
284                 break;

286         case ETHER_STAT_CAP_ASMPAUSE:
287                 *val = igb->param_asym_pause_cap;
288                 break;

290         case ETHER_STAT_CAP_PAUSE:
291                 *val = igb->param_pause_cap;
292                 break;

294         case ETHER_STAT_CAP_AUTONEG:
295                 *val = igb->param_autoneg_cap;
296                 break;

298         case ETHER_STAT_ADV_CAP_1000FDX:
299                 *val = igb->param_adv_1000fdx_cap;
300                 break;

302         case ETHER_STAT_ADV_CAP_1000HDX:
303                 *val = igb->param_adv_1000hdx_cap;
304                 break;

306         case ETHER_STAT_ADV_CAP_100FDX:
307                 *val = igb->param_adv_100fdx_cap;
308                 break;

310         case ETHER_STAT_ADV_CAP_100HDX:
311                 *val = igb->param_adv_100hdx_cap;
312                 break;

314         case ETHER_STAT_ADV_CAP_10FDX:
315                 *val = igb->param_adv_10fdx_cap;
316                 break;

318         case ETHER_STAT_ADV_CAP_10HDX:
319                 *val = igb->param_adv_10hdx_cap;
320                 break;

322         case ETHER_STAT_ADV_CAP_ASMPAUSE:
323                 *val = igb->param_adv_asym_pause_cap;
324                 break;
```



```
326         case ETHER_STAT_ADV_CAP_PAUSE:
327                 *val = igb->param_adv_pause_cap;
328                 break;

330         case ETHER_STAT_ADV_CAP_AUTONEG:
331                 *val = hw->mac.autoneg;
332                 break;

334         case ETHER_STAT_LP_CAP_1000FDX:
335                 *val = igb->param_lp_1000fdx_cap;
336                 break;

338         case ETHER_STAT_LP_CAP_1000HDX:
339                 *val = igb->param_lp_1000hdx_cap;
340                 break;

342         case ETHER_STAT_LP_CAP_100FDX:
343                 *val = igb->param_lp_100fdx_cap;
344                 break;

346         case ETHER_STAT_LP_CAP_100HDX:
347                 *val = igb->param_lp_100hdx_cap;
348                 break;

350         case ETHER_STAT_LP_CAP_10FDX:
351                 *val = igb->param_lp_10fdx_cap;
352                 break;

354         case ETHER_STAT_LP_CAP_10HDX:
355                 *val = igb->param_lp_10hdx_cap;
356                 break;

358         case ETHER_STAT_LP_CAP_ASMPAUSE:
359                 *val = igb->param_lp_asym_pause_cap;
360                 break;

362         case ETHER_STAT_LP_CAP_PAUSE:
363                 *val = igb->param_lp_pause_cap;
364                 break;

366         case ETHER_STAT_LP_CAP_AUTONEG:
367                 *val = igb->param_lp_autoneg_cap;
368                 break;

370         case ETHER_STAT_LINK_ASMPAUSE:
371                 *val = igb->param_asym_pause_cap;
372                 break;

374         case ETHER_STAT_LINK_PAUSE:
375                 *val = igb->param_pause_cap;
376                 break;

378         case ETHER_STAT_LINK_AUTONEG:
379                 *val = hw->mac.autoneg;
380                 break;

382         case ETHER_STAT_LINK_DUPLEX:
383                 *val = (igb->link_duplex == FULL_DUPLEX) ?
384                     LINK_DUPLEX_FULL : LINK_DUPLEX_HALF;
385                 break;

387         case ETHER_STAT_TOOSHORT_ERRORS:
388                 igb_ks->ruc.value.ui64 +=
389                     E1000_READ_REG(hw, E1000_RUC);
390                 *val = igb_ks->ruc.value.ui64;
391                 break;
```

```
393         case ETHER_STAT_CAP_REMFAULT:
394                 *val = igb->param_rem_fault;
395                 break;
396
397         case ETHER_STAT_ADV_REMFAULT:
398                 *val = igb->param_adv_rem_fault;
399                 break;
400
401         case ETHER_STAT_LP_REMFAULT:
402                 *val = igb->param_lp_rem_fault;
403                 break;
404
405         case ETHER_STAT_JABBER_ERRORS:
406                 igb_ks->rjc.value.ui64 +=
407                     E1000_READ_REG(hw, E1000_RJC);
408                 *val = igb_ks->rjc.value.ui64;
409                 break;
410
411         case ETHER_STAT_CAP_100T4:
412                 *val = igb->param_100t4_cap;
413                 break;
414
415         case ETHER_STAT_ADV_CAP_100T4:
416                 *val = igb->param_adv_100t4_cap;
417                 break;
418
419         case ETHER_STAT_LP_CAP_100T4:
420                 *val = igb->param_lp_100t4_cap;
421                 break;
422
423         default:
424                 mutex_exit(&igb->gen_lock);
425                 return (ENOTSUP);
426         }
427
428         mutex_exit(&igb->gen_lock);
429
430         if (igb_check_acc_handle(igb->osdep.reg_handle) != DDI_FM_OK) {
431                 ddi_fm_service_impact(igb->dip, DDI_SERVICE_DEGRADED);
432                 return (EIO);
433         }
434
435         return (0);
436 }
```

unchanged_portion_omitted_

```
1379 /* ARGSUSED */
1380 int
1381 igb_set_priv_prop(igb_t *igb, const char *pr_name,
1382     uint_t pr_valsize, const void *pr_val)
1383 {
1384         int err = 0;
1385         long result;
1386         struct e1000_hw *hw = &igb->hw;
1387         int i;
1388
1389         if (strcmp(pr_name, "_eee_support") == 0) {
1390                 if (pr_val == NULL)
1391                         return (EINVAL);
1392                 (void) ddi_strtol(pr_val, (char **)NULL, 0, &result);
1393                 switch (result) {
1394                 case 0:
1395                 case 1:
1396                         if (hw->mac.type != e1000_i350) {
1397                                 /*
```



```
1398                                  * For now, only supported on I350.
1399                                  * Add new mac.type values (or use < instead)
1400                                  * as new cards offer up EEE.
1401                                  */
1402                                 return (ENXIO);
1403                         }
1404                         /* Must set this prior to the set call. */
1405                         hw->dev_spec._82575.eee_disable = !result;
1406                         if (e1000_set_eee_i350(hw) != E1000_SUCCESS)
1407                                 err = EIO;
1408                         break;
1409                 default:
1410                         err = EINVAL;
1411                         /* FALLTHRU */
1412                 }
1413                 return (err);
1414         }
1415         if (strcmp(pr_name, "_tx_copy_thresh") == 0) {
1416                 if (pr_val == NULL) {
1417                         err = EINVAL;
1418                         return (err);
1419                 }
1420                 (void) ddi_strtol(pr_val, (char **)NULL, 0, &result);
1421                 if (result < MIN_TX_COPY_THRESHOLD ||
1422                     result > MAX_TX_COPY_THRESHOLD)
1423                         err = EINVAL;
1424                 else {
1425                         igb->tx_copy_thresh = (uint32_t)result;
1426                 }
1427                 return (err);
1428         }
1429         if (strcmp(pr_name, "_tx_recycle_thresh") == 0) {
1430                 if (pr_val == NULL) {
1431                         err = EINVAL;
1432                         return (err);
1433                 }
1434                 (void) ddi_strtol(pr_val, (char **)NULL, 0, &result);
1435                 if (result < MIN_TX_RECYCLE_THRESHOLD ||
1436                     result > MAX_TX_RECYCLE_THRESHOLD)
1437                         err = EINVAL;
1438                 else {
1439                         igb->tx_recycle_thresh = (uint32_t)result;
1440                 }
1441                 return (err);
1442         }
1443         if (strcmp(pr_name, "_tx_overload_thresh") == 0) {
1444                 if (pr_val == NULL) {
1445                         err = EINVAL;
1446                         return (err);
1447                 }
1448                 (void) ddi_strtol(pr_val, (char **)NULL, 0, &result);
1449                 if (result < MIN_TX_OVERLOAD_THRESHOLD ||
1450                     result > MAX_TX_OVERLOAD_THRESHOLD)
1451                         err = EINVAL;
1452                 else {
1453                         igb->tx_overload_thresh = (uint32_t)result;
1454                 }
1455                 return (err);
1456         }
1457         if (strcmp(pr_name, "_tx_resched_thresh") == 0) {
1458                 if (pr_val == NULL) {
1459                         err = EINVAL;
1460                         return (err);
1461                 }
1462                 (void) ddi_strtol(pr_val, (char **)NULL, 0, &result);
1463                 if (result < MIN_TX_RESCHED_THRESHOLD ||
```

```
1464                    result > MAX_TX_RESCHED_THRESHOLD ||
1465                    result > igb->tx_ring_size)
1466                        err = EINVAL;
1467                else {
1468                        igb->tx_resched_thresh = (uint32_t)result;
1469                }
1470                return (err);
1471        }
1472        if (strcmp(pr_name, "_rx_copy_thresh") == 0) {
1473                if (pr_val == NULL) {
1474                        err = EINVAL;
1475                        return (err);
1476                }
1477                (void) ddi_strtol(pr_val, (char **)NULL, 0, &result);
1478                if (result < MIN_RX_COPY_THRESHOLD ||
1479                    result > MAX_RX_COPY_THRESHOLD)
1480                        err = EINVAL;
1481                else {
1482                        igb->rx_copy_thresh = (uint32_t)result;
1483                }
1484                return (err);
1485        }
1486        if (strcmp(pr_name, "_rx_limit_per_intr") == 0) {
1487                if (pr_val == NULL) {
1488                        err = EINVAL;
1489                        return (err);
1490                }
1491                (void) ddi_strtol(pr_val, (char **)NULL, 0, &result);
1492                if (result < MIN_RX_LIMIT_PER_INTR ||
1493                    result > MAX_RX_LIMIT_PER_INTR)
1494                        err = EINVAL;
1495                else {
1496                        igb->rx_limit_per_intr = (uint32_t)result;
1497                }
1498                return (err);
1499        }
1500        if (strcmp(pr_name, "_intr_throttling") == 0) {
1501                if (pr_val == NULL) {
1502                        err = EINVAL;
1503                        return (err);
1504                }
1505                (void) ddi_strtol(pr_val, (char **)NULL, 0, &result);

1507                if (result < igb->capab->min_intr_throttle ||
1508                    result > igb->capab->max_intr_throttle)
1509                        err = EINVAL;
1510                else {
1511                        igb->intr_throttling[0] = (uint32_t)result;

1513                        for (i = 0; i < MAX_NUM_EITR; i++)
1514                                igb->intr_throttling[i] =
1515                                    igb->intr_throttling[0];

1517                        /* Set interrupt throttling rate */
1518                        for (i = 0; i < igb->intr_cnt; i++)
1519                                E1000_WRITE_REG(hw, E1000_EITR(i),
1520                                    igb->intr_throttling[i]);
1521                }
1522                return (err);
1523        }
1524        return (ENOTSUP);
1525 }

1527 int
1528 igb_get_priv_prop(igb_t *igb, const char *pr_name, uint_t pr_valsize,
1529     void *pr_val)
```



```
1530 {
1531         int value;

1533         if (strcmp(pr_name, "_adv_pause_cap") == 0) {
1534                 value = igb->param_adv_pause_cap;
1535         } else if (strcmp(pr_name, "_adv_asym_pause_cap") == 0) {
1536                 value = igb->param_adv_asym_pause_cap;
1537         } else if (strcmp(pr_name, "_eee_support") == 0) {
1538                 /*
1539                  * For now, only supported on I350.  Add new mac.type values
1540                  * (or use < instead) as new cards offer up EEE.
1541                  */
1542                 value = (igb->hw.mac.type != e1000_i350) ? 0 :
1543                     !(igb->hw.dev_spec._82575.eee_disable);
1544         } else if (strcmp(pr_name, "_tx_copy_thresh") == 0) {
1545                 value = igb->tx_copy_thresh;
1546         } else if (strcmp(pr_name, "_tx_recycle_thresh") == 0) {
1547                 value = igb->tx_recycle_thresh;
1548         } else if (strcmp(pr_name, "_tx_overload_thresh") == 0) {
1549                 value = igb->tx_overload_thresh;
1550         } else if (strcmp(pr_name, "_tx_resched_thresh") == 0) {
1551                 value = igb->tx_resched_thresh;
1552         } else if (strcmp(pr_name, "_rx_copy_thresh") == 0) {
1553                 value = igb->rx_copy_thresh;
1554         } else if (strcmp(pr_name, "_rx_limit_per_intr") == 0) {
1555                 value = igb->rx_limit_per_intr;
1556         } else if (strcmp(pr_name, "_intr_throttling") == 0) {
1557                 value = igb->intr_throttling[0];
1558         } else {
1559                 return (ENOTSUP);
1560         }

1562         (void) snprintf(pr_val, pr_valsize, "%d", value);
1563         return (0);
1564 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
  126724 Thu Feb  7 16:20:18 2013
new/usr/src/uts/common/io/igb/igb_main.c
3534 Disable EEE support in igb for I350
**********************************************************
```

```
/*
 * 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 (c) 2007-2012 Intel Corporation. All rights reserved.
 */

/*
 * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2013, Nexenta Systems, Inc. All rights reserved.
 */

#include "igb_sw.h"

static char ident[] = "Intel 1Gb Ethernet";
static char igb_version[] = "igb 1.1.18";

/*
 * Local function protoypes
 */
static int igb_register_mac(igb_t *);
static int igb_identify_hardware(igb_t *);
static int igb_regs_map(igb_t *);
static void igb_init_properties(igb_t *);
static int igb_init_driver_settings(igb_t *);
static void igb_init_locks(igb_t *);
static void igb_destroy_locks(igb_t *);
static int igb_init_mac_address(igb_t *);
static int igb_init(igb_t *);
static int igb_init_adapter(igb_t *);
static void igb_stop_adapter(igb_t *);
static int igb_reset(igb_t *);
static void igb_tx_clean(igb_t *);
static boolean_t igb_tx_drain(igb_t *);
static boolean_t igb_rx_drain(igb_t *);
static int igb_alloc_rings(igb_t *);
static int igb_alloc_rx_data(igb_t *);
static void igb_free_rx_data(igb_t *);
static void igb_free_rings(igb_t *);
static void igb_setup_rings(igb_t *);
static void igb_setup_rx(igb_t *);
static void igb_setup_tx(igb_t *);
static void igb_setup_rx_ring(igb_rx_ring_t *);
```



```
static void igb_setup_tx_ring(igb_tx_ring_t *);
static void igb_setup_rss(igb_t *);
static void igb_setup_mac_rss_classify(igb_t *);
static void igb_setup_mac_classify(igb_t *);
static void igb_init_unicst(igb_t *);
static void igb_setup_multicst(igb_t *);
static void igb_get_phy_state(igb_t *);
static void igb_param_sync(igb_t *);
static void igb_get_conf(igb_t *);
static int igb_get_prop(igb_t *, char *, int, int, int);
static boolean_t igb_is_link_up(igb_t *);
static boolean_t igb_link_check(igb_t *);
static void igb_local_timer(void *);
static void igb_link_timer(void *);
static void igb_arm_watchdog_timer(igb_t *);
static void igb_start_watchdog_timer(igb_t *);
static void igb_restart_watchdog_timer(igb_t *);
static void igb_stop_watchdog_timer(igb_t *);
static void igb_start_link_timer(igb_t *);
static void igb_stop_link_timer(igb_t *);
static void igb_disable_adapter_interrupts(igb_t *);
static void igb_enable_adapter_interrupts_82575(igb_t *);
static void igb_enable_adapter_interrupts_82576(igb_t *);
static void igb_enable_adapter_interrupts_82580(igb_t *);
static boolean_t is_valid_mac_addr(uint8_t *);
static boolean_t igb_stall_check(igb_t *);
static boolean_t igb_set_loopback_mode(igb_t *, uint32_t);
static void igb_set_external_loopback(igb_t *);
static void igb_set_internal_phy_loopback(igb_t *);
static void igb_set_internal_serdes_loopback(igb_t *);
static boolean_t igb_find_mac_address(igb_t *);
static int igb_alloc_intrs(igb_t *);
static int igb_alloc_intr_handles(igb_t *, int);
static int igb_add_intr_handlers(igb_t *);
static void igb_rem_intr_handlers(igb_t *);
static void igb_rem_intrs(igb_t *);
static int igb_enable_intrs(igb_t *);
static int igb_disable_intrs(igb_t *);
static void igb_setup_msix_82575(igb_t *);
static void igb_setup_msix_82576(igb_t *);
static void igb_setup_msix_82580(igb_t *);
static uint_t igb_intr_legacy(void *, void *);
static uint_t igb_intr_msi(void *, void *);
static uint_t igb_intr_rx(void *, void *);
static uint_t igb_intr_tx(void *, void *);
static uint_t igb_intr_tx_other(void *, void *);
static void igb_intr_rx_work(igb_rx_ring_t *);
static void igb_intr_tx_work(igb_tx_ring_t *);
static void igb_intr_link_work(igb_t *);
static void igb_get_driver_control(struct e1000_hw *);
static void igb_release_driver_control(struct e1000_hw *);

static int igb_attach(dev_info_t *, ddi_attach_cmd_t);
static int igb_detach(dev_info_t *, ddi_detach_cmd_t);
static int igb_resume(dev_info_t *);
static int igb_suspend(dev_info_t *);
static int igb_quiesce(dev_info_t *);
static void igb_unconfigure(dev_info_t *, igb_t *);
static int igb_fm_error_cb(dev_info_t *, ddi_fm_error_t *,
    const void *);
static void igb_fm_init(igb_t *);
static void igb_fm_fini(igb_t *);
static void igb_release_multicast(igb_t *);

char *igb_priv_props[] = {
        "_eee_support",
```

```
 128         "_tx_copy_thresh",
 129         "_tx_recycle_thresh",
 130         "_tx_overload_thresh",
 131         "_tx_resched_thresh",
 132         "_rx_copy_thresh",
 133         "_rx_limit_per_intr",
 134         "_intr_throttling",
 135         "_adv_pause_cap",
 136         "_adv_asym_pause_cap",
 137         NULL
 138 };
```

unchanged_portion_omitted_