```
**********************************************************
   70045 Mon Jan 14 13:17:24 2019
new/usr/src/uts/common/io/atge/atge_main.c
10087 atge_attach() doesn't need to check for kmem_zalloc() success
**********************************************************
```

```
/*
 * 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) 2012 Gary Mills
 *
 * Copyright (c) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
 *
 * Copyright (c) 2018, Joyent, Inc.
 */
/*
 * Copyright (c) 2009, Pyun YongHyeon <yongari@FreeBSD.org>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice unmodified, this list of conditions, and the following
 *    disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */

#include <sys/types.h>
#include <sys/stream.h>
#include <sys/strsun.h>
#include <sys/stat.h>
#include <sys/modctl.h>
#include <sys/kstat.h>
```



```
#include <sys/ethernet.h>
#include <sys/devops.h>
#include <sys/debug.h>
#include <sys/conf.h>
#include <sys/mii.h>
#include <sys/miiregs.h>
#include <sys/mac.h>
#include <sys/mac_provider.h>
#include <sys/mac_ether.h>
#include <sys/sysmacros.h>
#include <sys/dditypes.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/byteorder.h>
#include <sys/note.h>
#include <sys/vlan.h>
#include <sys/strsubr.h>
#include <sys/crc32.h>
#include <sys/sdt.h>
#include <sys/pci.h>
#include <sys/pci_cap.h>

#include "atge.h"
#include "atge_cmn_reg.h"
#include "atge_l1c_reg.h"
#include "atge_l1e_reg.h"
#include "atge_l1_reg.h"


/*
 * Atheros/Attansic Ethernet chips are of four types - L1, L2, L1E and L1C.
 * This driver is for L1E/L1/L1C but can be extended to support other chips.
 * L1E comes in 1Gigabit and Fast Ethernet flavors. L1 comes in 1Gigabit
 * flavors only.  L1C comes in both flavours.
 *
 * Atheros/Attansic Ethernet controllers have descriptor based TX and RX
 * with an exception of L1E. L1E's RX side is not descriptor based ring.
 * The L1E's RX uses pages (not to be confused with MMU pages) for
 * receiving pkts. The header has four fields :
 *
 *       uint32_t seqno;    Sequence number of the frame.
 *       uint32_t length;   Length of the frame.
 *       uint32_t flags;    Flags
 *       uint32_t vtag;     We don't use hardware VTAG.
 *
 * We use only one queue for RX (each queue can have two pages) and each
 * page is L1E_RX_PAGE_SZ large in bytes. That's the reason we don't
 * use zero-copy RX because we are limited to two pages and each page
 * accomodates large number of pkts.
 *
 * The TX side on all three chips is descriptor based ring; and all the
 * more reason to have one driver for these chips.
 *
 * We use two locks - atge_intr_lock and atge_tx_lock. Both the locks
 * should be held if the operation has impact on the driver instance.
 *
 * All the three chips have hash-based multicast filter.
 *
 * We use CMB (Coalescing Message Block) for RX but not for TX as there
 * are some issues with TX. RX CMB is used to get the last descriptor
 * posted by the chip. Each CMB is for a RX page (one queue can have two
 * pages) and are uint32_t (4 bytes) long.
 *
 * The descriptor table should have 32-bit physical address limit due to
 * the limitation of having same high address for TX/RX/SMB/CMB. The
 * TX/RX buffers can be 64-bit.
```

```
 128  *
 129  * Every DMA memory in atge is represented by atge_dma_t be it TX/RX Buffers
 130  * or TX/RX descriptor table or SMB/CMB. To keep the code simple, we have
 131  * kept sgl as 1 so that we get contingous pages from root complex.
 132  *
 133  * L1 chip (0x1048) uses descriptor based TX and RX ring. Most of registers are
 134  * common with L1E chip (0x1026).
 135  */

 137 /*
 138  * Function Prototypes for debugging.
 139  */
 140 void    atge_error(dev_info_t *, char *, ...);
 141 void    atge_debug_func(char *, ...);

 143 /*
 144  * Function Prototypes for driver operations.
 145  */
 146 static int      atge_resume(dev_info_t *);
 147 static int      atge_add_intr(atge_t *);
 148 static int      atge_alloc_dma(atge_t *);
 149 static void     atge_remove_intr(atge_t *);
 150 static void     atge_free_dma(atge_t *);
 151 static void     atge_device_reset(atge_t *);
 152 static void     atge_device_init(atge_t *);
 153 static void     atge_device_start(atge_t *);
 154 static void     atge_disable_intrs(atge_t *);
 155 atge_dma_t *atge_alloc_a_dma_blk(atge_t *, ddi_dma_attr_t *, int, int);
 156 void    atge_free_a_dma_blk(atge_dma_t *);
 157 static void     atge_rxfilter(atge_t *);
 158 static void     atge_device_reset_l1_l1e(atge_t *);
 159 void    atge_program_ether(atge_t *atgep);
 160 void    atge_device_restart(atge_t *);
 161 void    atge_device_stop(atge_t *);
 162 static int      atge_send_a_packet(atge_t *, mblk_t *);
 163 static uint32_t atge_ether_crc(const uint8_t *, int);


 166 /*
 167  * L1E/L2E specific functions.
 168  */
 169 void    atge_l1e_device_reset(atge_t *);
 170 void    atge_l1e_stop_mac(atge_t *);
 171 int     atge_l1e_alloc_dma(atge_t *);
 172 void    atge_l1e_free_dma(atge_t *);
 173 void    atge_l1e_init_tx_ring(atge_t *);
 174 void    atge_l1e_init_rx_pages(atge_t *);
 175 void    atge_l1e_program_dma(atge_t *);
 176 void    atge_l1e_send_packet(atge_ring_t *);
 177 mblk_t  *atge_l1e_receive(atge_t *);
 178 uint_t  atge_l1e_interrupt(caddr_t, caddr_t);
 179 void    atge_l1e_gather_stats(atge_t *);
 180 void    atge_l1e_clear_stats(atge_t *);

 182 /*
 183  * L1 specific functions.
 184  */
 185 int     atge_l1_alloc_dma(atge_t *);
 186 void    atge_l1_free_dma(atge_t *);
 187 void    atge_l1_init_tx_ring(atge_t *);
 188 void    atge_l1_init_rx_ring(atge_t *);
 189 void    atge_l1_init_rr_ring(atge_t *);
 190 void    atge_l1_init_cmb(atge_t *);
 191 void    atge_l1_init_smb(atge_t *);
 192 void    atge_l1_program_dma(atge_t *);
 193 void    atge_l1_stop_tx_mac(atge_t *);
```



```
 194 void    atge_l1_stop_rx_mac(atge_t *);
 195 uint_t  atge_l1_interrupt(caddr_t, caddr_t);
 196 void    atge_l1_send_packet(atge_ring_t *);

 198 /*
 199  * L1C specific functions.
 200  */
 201 int     atge_l1c_alloc_dma(atge_t *);
 202 void    atge_l1c_free_dma(atge_t *);
 203 void    atge_l1c_init_tx_ring(atge_t *);
 204 void    atge_l1c_init_rx_ring(atge_t *);
 205 void    atge_l1c_init_rr_ring(atge_t *);
 206 void    atge_l1c_init_cmb(atge_t *);
 207 void    atge_l1c_init_smb(atge_t *);
 208 void    atge_l1c_program_dma(atge_t *);
 209 void    atge_l1c_stop_tx_mac(atge_t *);
 210 void    atge_l1c_stop_rx_mac(atge_t *);
 211 uint_t  atge_l1c_interrupt(caddr_t, caddr_t);
 212 void    atge_l1c_send_packet(atge_ring_t *);
 213 void    atge_l1c_gather_stats(atge_t *);
 214 void    atge_l1c_clear_stats(atge_t *);

 216 /*
 217  * Function prototyps for MII operations.
 218  */
 219 uint16_t        atge_mii_read(void *, uint8_t, uint8_t);
 220 void    atge_mii_write(void *, uint8_t, uint8_t, uint16_t);
 221 uint16_t        atge_l1c_mii_read(void *, uint8_t, uint8_t);
 222 void    atge_l1c_mii_write(void *, uint8_t, uint8_t, uint16_t);
 223 void    atge_l1e_mii_reset(void *);
 224 void    atge_l1_mii_reset(void *);
 225 void    atge_l1c_mii_reset(void *);
 226 static void     atge_mii_notify(void *, link_state_t);
 227 void    atge_tx_reclaim(atge_t *atgep, int cons);

 229 /*
 230  * L1E/L2E chip.
 231  */
 232 static  mii_ops_t atge_l1e_mii_ops = {
 233         MII_OPS_VERSION,
 234         atge_mii_read,
 235         atge_mii_write,
 236         atge_mii_notify,
 237         atge_l1e_mii_reset
 238 };
_____unchanged_portion_omitted_

1050 /*
1051  * Attach entry point in the driver.
1052  */
1053 static int
1054 atge_attach(dev_info_t *devinfo, ddi_attach_cmd_t cmd)
1055 {
1056         atge_t  *atgep;
1057         mac_register_t  *macreg;
1058         int     instance;
1059         uint16_t cap_ptr;
1060         uint16_t burst;
1061         int err;
1062         mii_ops_t *mii_ops;

1064         instance =  ddi_get_instance(devinfo);

1066         switch (cmd) {
1067         case DDI_RESUME:
1068                 return (atge_resume(devinfo));
```

```
1070        case DDI_ATTACH:
1071                ddi_set_driver_private(devinfo, NULL);
1072                break;
1073        default:
1074                return (DDI_FAILURE);

1076        }

1078        atgep = kmem_zalloc(sizeof (atge_t), KM_SLEEP);
1079        ddi_set_driver_private(devinfo, atgep);
1080        atgep->atge_dip = devinfo;

1082        /*
1083         * Setup name and instance number to be used for debugging and
1084         * error reporting.
1085         */
1086        (void) snprintf(atgep->atge_name, sizeof (atgep->atge_name), "%s%d",
1087            "atge", instance);


1090        /*
1091         * Map PCI config space.
1092         */
1093        err = pci_config_setup(devinfo, &atgep->atge_conf_handle);
1094        if (err != DDI_SUCCESS) {
1095                atge_error(devinfo, "pci_config_setup() failed");
1096                goto fail1;
1097        }

1099        (void) atge_identify_hardware(atgep);

1101        /*
1102         * Map Device registers.
1103         */
1104        err = ddi_regs_map_setup(devinfo, ATGE_PCI_REG_NUMBER,
1105            &atgep->atge_io_regs, 0, 0, &atge_dev_attr, &atgep->atge_io_handle);
1106        if (err != DDI_SUCCESS) {
1107                atge_error(devinfo, "ddi_regs_map_setup() failed");
1108                goto fail2;
1109        }

1111        /*
1112         * Add interrupt and its associated handler.
1113         */
1114        err = atge_add_intr(atgep);
1115        if (err != DDI_SUCCESS) {
1116                atge_error(devinfo, "Failed to add interrupt handler");
1117                goto fail3;
1118        }

1120        mutex_init(&atgep->atge_intr_lock, NULL, MUTEX_DRIVER,
1121            DDI_INTR_PRI(atgep->atge_intr_pri));

1123        mutex_init(&atgep->atge_tx_lock, NULL, MUTEX_DRIVER,
1124            DDI_INTR_PRI(atgep->atge_intr_pri));

1126        mutex_init(&atgep->atge_rx_lock, NULL, MUTEX_DRIVER,
1127            DDI_INTR_PRI(atgep->atge_intr_pri));

1129        mutex_init(&atgep->atge_mii_lock, NULL, MUTEX_DRIVER, NULL);

1131        /*
1132         * Used to lock down MBOX register on L1 chip since RX consumer,
1133         * TX producer and RX return ring consumer are shared.
1134         */
```



```
1135        mutex_init(&atgep->atge_mbox_lock, NULL, MUTEX_DRIVER,
1136            DDI_INTR_PRI(atgep->atge_intr_pri));

1138        atgep->atge_link_state = LINK_STATE_DOWN;
1139        atgep->atge_mtu = ETHERMTU;

1141        switch (ATGE_MODEL(atgep)) {
1142        case ATGE_CHIP_L1E:
1143                if (atgep->atge_revid > 0xF0) {
1144                        /* L2E Rev. B. AR8114 */
1145                        atgep->atge_flags |= ATGE_FLAG_FASTETHER;
1146                } else {
1147                        if ((INL(atgep, L1E_PHY_STATUS) &
1148                            PHY_STATUS_100M) != 0) {
1149                                /* L1E AR8121 */
1150                                atgep->atge_flags |= ATGE_FLAG_JUMBO;
1151                        } else {
1152                                /* L2E Rev. A. AR8113 */
1153                                atgep->atge_flags |= ATGE_FLAG_FASTETHER;
1154                        }
1155                }
1156                break;
1157        case ATGE_CHIP_L1:
1158                break;
1159        case ATGE_CHIP_L1C:
1160                /*
1161                 * One odd thing is AR8132 uses the same PHY hardware(F1
1162                 * gigabit PHY) of AR8131. So atphy(4) of AR8132 reports
1163                 * the PHY supports 1000Mbps but that's not true. The PHY
1164                 * used in AR8132 can't establish gigabit link even if it
1165                 * shows the same PHY model/revision number of AR8131.
1166                 *
1167                 * It seems that AR813x/AR815x has silicon bug for SMB. In
1168                 * addition, Atheros said that enabling SMB wouldn't improve
1169                 * performance. However I think it's bad to access lots of
1170                 * registers to extract MAC statistics.
1171                 *
1172                 * Don't use Tx CMB. It is known to have silicon bug.
1173                 */
1174                switch (ATGE_DID(atgep)) {
1175                case ATGE_CHIP_AR8152V2_DEV_ID:
1176                case ATGE_CHIP_AR8152V1_DEV_ID:
1177                        atgep->atge_flags |= ATGE_FLAG_APS |
1178                            ATGE_FLAG_FASTETHER |
1179                            ATGE_FLAG_ASPM_MON | ATGE_FLAG_JUMBO |
1180                            ATGE_FLAG_SMB_BUG | ATGE_FLAG_CMB_BUG;
1181                        break;
1182                case ATGE_CHIP_AR8151V2_DEV_ID:
1183                case ATGE_CHIP_AR8151V1_DEV_ID:
1184                        atgep->atge_flags |= ATGE_FLAG_APS |
1185                            ATGE_FLAG_ASPM_MON | ATGE_FLAG_JUMBO |
1186                            ATGE_FLAG_SMB_BUG | ATGE_FLAG_CMB_BUG;
1187                        break;
1188                case ATGE_CHIP_L1CF_DEV_ID:
1189                        atgep->atge_flags |= ATGE_FLAG_FASTETHER;
1190                        break;
1191                case ATGE_CHIP_L1CG_DEV_ID:
1192                        break;
1193                }
1194                break;
1195        }

1197        /*
1198         * Get DMA parameters from PCIe device control register.
1199         */
1200        err = PCI_CAP_LOCATE(atgep->atge_conf_handle, PCI_CAP_ID_PCI_E,
```

```
1201             &cap_ptr);

1203         if (err == DDI_FAILURE) {
1204                 atgep->atge_dma_rd_burst = DMA_CFG_RD_BURST_128;
1205                 atgep->atge_dma_wr_burst = DMA_CFG_WR_BURST_128;
1206         } else {
1207                 atgep->atge_flags |= ATGE_FLAG_PCIE;
1208                 burst = pci_config_get16(atgep->atge_conf_handle,
1209                     cap_ptr + 0x08);

1211                 /*
1212                  * Max read request size.
1213                  */
1214                 atgep->atge_dma_rd_burst = ((burst >> 12) & 0x07) <<
1215                     DMA_CFG_RD_BURST_SHIFT;

1217                 /*
1218                  * Max Payload Size.
1219                  */
1220                 atgep->atge_dma_wr_burst = ((burst >> 5) & 0x07) <<
1221                     DMA_CFG_WR_BURST_SHIFT;

1223                 ATGE_DB(("%s: %s() MRR : %d, MPS : %d",
1224                     atgep->atge_name, __func__,
1225                     (128 << ((burst >> 12) & 0x07)),
1226                     (128 << ((burst >> 5) & 0x07))));
1227         }

1229         /* Clear data link and flow-control protocol error. */
1230         switch (ATGE_MODEL(atgep)) {
1231         case ATGE_CHIP_L1E:
1232                 break;
1233         case ATGE_CHIP_L1:
1234                 break;
1235         case ATGE_CHIP_L1C:
1236                 OUTL_AND(atgep, ATGE_PEX_UNC_ERR_SEV,
1237                     ~(PEX_UNC_ERR_SEV_UC | PEX_UNC_ERR_SEV_FCP));
1238                 OUTL_AND(atgep, ATGE_LTSSM_ID_CFG, ~LTSSM_ID_WRO_ENB);
1239                 OUTL_OR(atgep, ATGE_PCIE_PHYMISC, PCIE_PHYMISC_FORCE_RCV_DET);
1240                 break;
1241         }

1243         /*
1244          * Allocate DMA resources.
1245          */
1246         err = atge_alloc_dma(atgep);
1247         if (err != DDI_SUCCESS) {
1248                 atge_error(devinfo, "Failed to allocate DMA resources");
1249                 goto fail4;
1250         }

1252         /*
1253          * Get station address.
1254          */
1255         (void) atge_get_macaddr(atgep);

1257         /*
1258          * Setup MII.
1259          */
1260         switch (ATGE_MODEL(atgep)) {
1261         case ATGE_CHIP_L1E:
1262                 mii_ops = &atge_l1e_mii_ops;
1263                 break;
1264         case ATGE_CHIP_L1:
1265                 mii_ops = &atge_l1_mii_ops;
1266                 break;
```



```
1267         case ATGE_CHIP_L1C:
1268                 mii_ops = &atge_l1c_mii_ops;
1269                 break;
1270         }

1272         if ((atgep->atge_mii = mii_alloc(atgep, devinfo,
1273             mii_ops)) == NULL) {
1274                 atge_error(devinfo, "mii_alloc() failed");
1275                 goto fail4;
1276         }

1278         /*
1279          * Register with MAC layer.
1280          */
1281         if ((macreg = mac_alloc(MAC_VERSION)) == NULL) {
1282                 atge_error(devinfo, "mac_alloc() failed due to version");
1283                 goto fail4;
1284         }

1286         macreg->m_type_ident = MAC_PLUGIN_IDENT_ETHER;
1287         macreg->m_driver = atgep;
1288         macreg->m_dip = devinfo;
1289         macreg->m_instance = instance;
1290         macreg->m_src_addr = atgep->atge_ether_addr;
1291         macreg->m_callbacks = &atge_m_callbacks;
1292         macreg->m_min_sdu = 0;
1293         macreg->m_max_sdu = atgep->atge_mtu;
1294         macreg->m_margin = VLAN_TAGSZ;

1296         if ((err = mac_register(macreg, &atgep->atge_mh)) != 0) {
1297                 atge_error(devinfo, "mac_register() failed with :%d", err);
1298                 mac_free(macreg);
1299                 goto fail4;
1300         }

1302         mac_free(macreg);

1304         ATGE_DB(("%s: %s() driver attached successfully",
1305             atgep->atge_name, __func__));

1307         atge_device_reset(atgep);

1309         atgep->atge_chip_state = ATGE_CHIP_INITIALIZED;

1311         /*
1312          * At last - enable interrupts.
1313          */
1314         err = atge_enable_intrs(atgep);
1315         if (err == DDI_FAILURE) {
1316                 goto fail5;
1317         }

1319         /*
1320          * Reset the PHY before starting.
1321          */
1322         switch (ATGE_MODEL(atgep)) {
1323         case ATGE_CHIP_L1E:
1324                 atge_l1e_mii_reset(atgep);
1325                 break;
1326         case ATGE_CHIP_L1:
1327                 atge_l1_mii_reset(atgep);
1328                 break;
1329         case ATGE_CHIP_L1C:
1330                 atge_l1c_mii_reset(atgep);
1331                 break;
1332         }
```

```
1334         /*
1335          * Let the PHY run.
1336          */
1337         mii_start(atgep->atge_mii);

1339         return (DDI_SUCCESS);

1341 fail5:
1342         (void) mac_unregister(atgep->atge_mh);
1343         atge_device_stop(atgep);
1344         mii_stop(atgep->atge_mii);
1345         mii_free(atgep->atge_mii);
1346 fail4:
1347         atge_free_dma(atgep);
1348         mutex_destroy(&atgep->atge_intr_lock);
1349         mutex_destroy(&atgep->atge_tx_lock);
1350         mutex_destroy(&atgep->atge_rx_lock);
1351         atge_remove_intr(atgep);
1352 fail3:
1353         ddi_regs_map_free(&atgep->atge_io_handle);
1354 fail2:
1355         pci_config_teardown(&atgep->atge_conf_handle);
1356 fail1:
1355         if (atgep)
1357         kmem_free(atgep, sizeof (atge_t));

1358         return (DDI_FAILURE);
1359 }
```

_____unchanged_portion_omitted_