```
**********************************************************
   39977 Fri Jul 27 22:35:55 2012
new/usr/src/uts/common/io/ixgbe/ixgbe_82598.c
3014 Intel X540 Support (fix lint)
**********************************************************
______unchanged_portion_omitted_

 710 /**
 711  *  ixgbe_setup_mac_link_82598 - Set MAC link speed
 712  *  @hw: pointer to hardware structure
 713  *  @speed: new link speed
 714  *  @autoneg: TRUE if autonegotiation enabled
 715  *  @autoneg_wait_to_complete: TRUE when waiting for completion is needed
 716  *
 717  *  Set the link speed in the AUTOC register and restarts link.
 718  **/
 719 static s32 ixgbe_setup_mac_link_82598(struct ixgbe_hw *hw,
 720                                       ixgbe_link_speed speed, bool autoneg,
 721                                       bool autoneg_wait_to_complete)
 722 {
 723         s32 status;
 723         s32 status = IXGBE_SUCCESS;
 724         ixgbe_link_speed link_capabilities = IXGBE_LINK_SPEED_UNKNOWN;
 725         u32 curr_autoc = IXGBE_READ_REG(hw, IXGBE_AUTOC);
 726         u32 autoc = curr_autoc;
 727         u32 link_mode = autoc & IXGBE_AUTOC_LMS_MASK;

 729         DEBUGFUNC("ixgbe_setup_mac_link_82598");

 731         /* Check to see if speed passed in is supported. */
 732         status = ixgbe_get_link_capabilities(hw, &link_capabilities, &autoneg);
 733         if (status != IXGBE_SUCCESS)
 734                 return (status);
 732         ixgbe_get_link_capabilities(hw, &link_capabilities, &autoneg);
 735         speed &= link_capabilities;

 737         if (speed == IXGBE_LINK_SPEED_UNKNOWN)
 738                 status = IXGBE_ERR_LINK_SETUP;

 740         /* Set KX4/KX support according to speed requested */
 741         else if (link_mode == IXGBE_AUTOC_LMS_KX4_AN ||
 742                  link_mode == IXGBE_AUTOC_LMS_KX4_AN_1G_AN) {
 743                 autoc &= ~IXGBE_AUTOC_KX4_KX_SUPP_MASK;
 744                 if (speed & IXGBE_LINK_SPEED_10GB_FULL)
 745                         autoc |= IXGBE_AUTOC_KX4_SUPP;
 746                 if (speed & IXGBE_LINK_SPEED_1GB_FULL)
 747                         autoc |= IXGBE_AUTOC_KX_SUPP;
 748                 if (autoc != curr_autoc)
 749                         IXGBE_WRITE_REG(hw, IXGBE_AUTOC, autoc);
 750         }

 752         if (status == IXGBE_SUCCESS) {
 753                 /*
 754                  * Setup and restart the link based on the new values in
 755                  * ixgbe_hw This will write the AUTOC register based on the new
 756                  * stored values
 757                  */
 758                 status = ixgbe_start_mac_link_82598(hw,
 759                                                     autoneg_wait_to_complete);
 760         }

 762         return status;
 763 }


 766 /**
```



```
 767  *  ixgbe_setup_copper_link_82598 - Set the PHY autoneg advertised field
 768  *  @hw: pointer to hardware structure
 769  *  @speed: new link speed
 770  *  @autoneg: TRUE if autonegotiation enabled
 771  *  @autoneg_wait_to_complete: TRUE if waiting is needed to complete
 772  *
 773  *  Sets the link speed in the AUTOC register in the MAC and restarts link.
 774  **/
 775 static s32 ixgbe_setup_copper_link_82598(struct ixgbe_hw *hw,
 776                                          ixgbe_link_speed speed,
 777                                          bool autoneg,
 778                                          bool autoneg_wait_to_complete)
 779 {
 780         s32 status;

 782         DEBUGFUNC("ixgbe_setup_copper_link_82598");

 784         /* Setup the PHY according to input speed */
 785         status = hw->phy.ops.setup_link_speed(hw, speed, autoneg,
 786                                               autoneg_wait_to_complete);
 787         if (status == IXGBE_SUCCESS) {
 788                 /* Set up MAC */
 789                 status =
 790                     ixgbe_start_mac_link_82598(hw, autoneg_wait_to_complete);
 791         }

 793         return status;
 794 }
______unchanged_portion_omitted_

1338 /**
1339  * ixgbe_set_rxpba_82598 - Initialize RX packet buffer
1340  * @hw: pointer to hardware structure
1341  * @num_pb: number of packet buffers to allocate
1342  * @headroom: reserve n KB of headroom
1343  * @strategy: packet buffer allocation strategy
1344  **/
1345 static void ixgbe_set_rxpba_82598(struct ixgbe_hw *hw, int num_pb,
1346                                   u32 headroom, int strategy)
1347 {
1348         u32 rxpktsize = IXGBE_RXPBSIZE_64KB;
1349         u8 i = 0;
1350         UNREFERENCED_1PARAMETER(headroom);

1352         if (!num_pb)
1353                 return;

1355         /* Setup Rx packet buffer sizes */
1356         switch (strategy) {
1357         case PBA_STRATEGY_WEIGHTED:
1358                 /* Setup the first four at 80KB */
1359                 rxpktsize = IXGBE_RXPBSIZE_80KB;
1360                 for (; i < 4; i++)
1361                         IXGBE_WRITE_REG(hw, IXGBE_RXPBSIZE(i), rxpktsize);
1362                 /* Setup the last four at 48KB...don't re-init i */
1363                 rxpktsize = IXGBE_RXPBSIZE_48KB;
1364                 /* Fall Through */
1365                 /* FALLTHRU */
1366         case PBA_STRATEGY_EQUAL:
1367         default:
1368                 /* Divide the remaining Rx packet buffer evenly among the TCs */
1369                 for (; i < IXGBE_MAX_PACKET_BUFFERS; i++)
1370                         IXGBE_WRITE_REG(hw, IXGBE_RXPBSIZE(i), rxpktsize);
1371                 break;
1372         }
```

```
1374         /* Setup Tx packet buffer sizes */
1375         for (i = 0; i < IXGBE_MAX_PACKET_BUFFERS; i++)
1376                 IXGBE_WRITE_REG(hw, IXGBE_TXPBSIZE(i), IXGBE_TXPBSIZE_40KB);

1372         return;
1377 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   69786 Fri Jul 27 22:36:00 2012
new/usr/src/uts/common/io/ixgbe/ixgbe_82599.c
3014 Intel X540 Support (fix lint)
**********************************************************
______unchanged_portion_omitted_

 934 /**
 935  *  ixgbe_setup_copper_link_82599 - Set the PHY autoneg advertised field
 936  *  @hw: pointer to hardware structure
 937  *  @speed: new link speed
 938  *  @autoneg: TRUE if autonegotiation enabled
 939  *  @autoneg_wait_to_complete: TRUE if waiting is needed to complete
 940  *
 941  *  Restarts link on PHY and MAC based on settings passed in.
 942  **/
 943 static s32 ixgbe_setup_copper_link_82599(struct ixgbe_hw *hw,
 944                                          ixgbe_link_speed speed,
 945                                          bool autoneg,
 946                                          bool autoneg_wait_to_complete)
 947 {
 948         s32 status;

 950         DEBUGFUNC("ixgbe_setup_copper_link_82599");

 952         /* Setup the PHY according to input speed */
 953         status = hw->phy.ops.setup_link_speed(hw, speed, autoneg,
 954                                               autoneg_wait_to_complete);
 955         if (status == IXGBE_SUCCESS) {
 956                 /* Set up MAC */
 957                 status =
 958                     ixgbe_start_mac_link_82599(hw, autoneg_wait_to_complete);
 959         }

 961         return status;
 962 }
______unchanged_portion_omitted_

1112 /**
1113  *  ixgbe_reinit_fdir_tables_82599 - Reinitialize Flow Director tables.
1114  *  @hw: pointer to hardware structure
1115  **/
1116 s32 ixgbe_reinit_fdir_tables_82599(struct ixgbe_hw *hw)
1117 {
1118         int i;
1119         u32 fdirctrl = IXGBE_READ_REG(hw, IXGBE_FDIRCTRL);
1120         fdirctrl &= ~IXGBE_FDIRCTRL_INIT_DONE;

1122         DEBUGFUNC("ixgbe_reinit_fdir_tables_82599");

1124         /*
1125          * Before starting reinitialization process,
1126          * FDIRCMD.CMD must be zero.
1127          */
1128         for (i = 0; i < IXGBE_FDIRCMD_CMD_POLL; i++) {
1129                 if (!(IXGBE_READ_REG(hw, IXGBE_FDIRCMD) &
1130                       IXGBE_FDIRCMD_CMD_MASK))
1131                         break;
1132                 usec_delay(10);
1133         }
1134         if (i >= IXGBE_FDIRCMD_CMD_POLL) {
1135                 DEBUGOUT("Flow Director previous command isn't complete, "
1136                          "aborting table re-initialization.\n");
1137                 return IXGBE_ERR_FDIR_REINIT_FAILED;
1138         }
```



```
1140         IXGBE_WRITE_REG(hw, IXGBE_FDIRFREE, 0);
1141         IXGBE_WRITE_FLUSH(hw);
1142         /*
1143          * 82599 adapters flow director init flow cannot be restarted,
1144          * Workaround 82599 silicon errata by performing the following steps
1145          * before re-writing the FDIRCTRL control register with the same value.
1146          * - write 1 to bit 8 of FDIRCMD register &
1147          * - write 0 to bit 8 of FDIRCMD register
1148          */
1149         IXGBE_WRITE_REG(hw, IXGBE_FDIRCMD,
1150                         (IXGBE_READ_REG(hw, IXGBE_FDIRCMD) |
1151                          IXGBE_FDIRCMD_CLEARHT));
1152         IXGBE_WRITE_FLUSH(hw);
1153         IXGBE_WRITE_REG(hw, IXGBE_FDIRCMD,
1154                         (IXGBE_READ_REG(hw, IXGBE_FDIRCMD) &
1155                          ~IXGBE_FDIRCMD_CLEARHT));
1156         IXGBE_WRITE_FLUSH(hw);
1157         /*
1158          * Clear FDIR Hash register to clear any leftover hashes
1159          * waiting to be programmed.
1160          */
1161         IXGBE_WRITE_REG(hw, IXGBE_FDIRHASH, 0x00);
1162         IXGBE_WRITE_FLUSH(hw);

1164         IXGBE_WRITE_REG(hw, IXGBE_FDIRCTRL, fdirctrl);
1165         IXGBE_WRITE_FLUSH(hw);

1167         /* Poll init-done after we write FDIRCTRL register */
1168         for (i = 0; i < IXGBE_FDIR_INIT_DONE_POLL; i++) {
1169                 if (IXGBE_READ_REG(hw, IXGBE_FDIRCTRL) &
1170                                    IXGBE_FDIRCTRL_INIT_DONE)
1171                         break;
1172                 usec_delay(10);
1173         }
1174         if (i >= IXGBE_FDIR_INIT_DONE_POLL) {
1175                 DEBUGOUT("Flow Director Signature poll time exceeded!\n");
1176                 return IXGBE_ERR_FDIR_REINIT_FAILED;
1177         }

1179         /* Clear FDIR statistics registers (read to clear) */
1180         (void) IXGBE_READ_REG(hw, IXGBE_FDIRUSTAT);
1181         (void) IXGBE_READ_REG(hw, IXGBE_FDIRFSTAT);
1182         (void) IXGBE_READ_REG(hw, IXGBE_FDIRMATCH);
1183         (void) IXGBE_READ_REG(hw, IXGBE_FDIRMISS);
1184         (void) IXGBE_READ_REG(hw, IXGBE_FDIRLEN);
1177         IXGBE_READ_REG(hw, IXGBE_FDIRUSTAT);
1178         IXGBE_READ_REG(hw, IXGBE_FDIRFSTAT);
1179         IXGBE_READ_REG(hw, IXGBE_FDIRMATCH);
1180         IXGBE_READ_REG(hw, IXGBE_FDIRMISS);
1181         IXGBE_READ_REG(hw, IXGBE_FDIRLEN);

1186         return IXGBE_SUCCESS;
1187 }
______unchanged_portion_omitted_

1288 /*
1289  * These defines allow us to quickly generate all of the necessary instructions
1290  * in the function below by simply calling out IXGBE_COMPUTE_SIG_HASH_ITERATION
1291  * for values 0 through 15
1292  */
1293 #define IXGBE_ATR_COMMON_HASH_KEY \
1294                 (IXGBE_ATR_BUCKET_HASH_KEY & IXGBE_ATR_SIGNATURE_HASH_KEY)
1295 #if lint
1296 #define IXGBE_COMPUTE_SIG_HASH_ITERATION(_n)
1297 #else
1298 #define IXGBE_COMPUTE_SIG_HASH_ITERATION(_n) \
```

```
1299 do { \
1300         u32 n = (_n); \
1301         if (IXGBE_ATR_COMMON_HASH_KEY & (0x01 << n)) \
1302                 common_hash ^= lo_hash_dword >> n; \
1303         else if (IXGBE_ATR_BUCKET_HASH_KEY & (0x01 << n)) \
1304                 bucket_hash ^= lo_hash_dword >> n; \
1305         else if (IXGBE_ATR_SIGNATURE_HASH_KEY & (0x01 << n)) \
1306                 sig_hash ^= lo_hash_dword << (16 - n); \
1307         if (IXGBE_ATR_COMMON_HASH_KEY & (0x01 << (n + 16))) \
1308                 common_hash ^= hi_hash_dword >> n; \
1309         else if (IXGBE_ATR_BUCKET_HASH_KEY & (0x01 << (n + 16))) \
1310                 bucket_hash ^= hi_hash_dword >> n; \
1311         else if (IXGBE_ATR_SIGNATURE_HASH_KEY & (0x01 << (n + 16))) \
1312                 sig_hash ^= hi_hash_dword << (16 - n); \
1313 } while (0);
1314 #endif

1316 /**
1317  *  ixgbe_atr_compute_sig_hash_82599 - Compute the signature hash
1318  *  @stream: input bitstream to compute the hash on
1319  *
1320  *  This function is almost identical to the function above but contains
1321  *  several optomizations such as unwinding all of the loops, letting the
1322  *  compiler work out all of the conditional ifs since the keys are static
1323  *  defines, and computing two keys at once since the hashed dword stream
1324  *  will be the same for both keys.
1325  **/
1326 u32 ixgbe_atr_compute_sig_hash_82599(union ixgbe_atr_hash_dword input,
1327                                      union ixgbe_atr_hash_dword common)
1328 {
1329         u32 hi_hash_dword, lo_hash_dword, flow_vm_vlan;
1330         u32 sig_hash = 0, bucket_hash = 0, common_hash = 0;

1332         /* record the flow_vm_vlan bits as they are a key part to the hash */
1333         flow_vm_vlan = IXGBE_NTOHL(input.dword);

1335         /* generate common hash dword */
1336         hi_hash_dword = IXGBE_NTOHL(common.dword);

1338         /* low dword is word swapped version of common */
1339         lo_hash_dword = (hi_hash_dword >> 16) | (hi_hash_dword << 16);

1341         /* apply flow ID/VM pool/VLAN ID bits to hash words */
1342         hi_hash_dword ^= flow_vm_vlan ^ (flow_vm_vlan >> 16);

1344         /* Process bits 0 and 16 */
1345         IXGBE_COMPUTE_SIG_HASH_ITERATION(0);

1347         /*
1348          * apply flow ID/VM pool/VLAN ID bits to lo hash dword, we had to
1349          * delay this because bit 0 of the stream should not be processed
1350          * so we do not add the vlan until after bit 0 was processed
1351          */
1352         lo_hash_dword ^= flow_vm_vlan ^ (flow_vm_vlan << 16);

1354         /* Process remaining 30 bit of the key */
1355         IXGBE_COMPUTE_SIG_HASH_ITERATION(1);
1356         IXGBE_COMPUTE_SIG_HASH_ITERATION(2);
1357         IXGBE_COMPUTE_SIG_HASH_ITERATION(3);
1358         IXGBE_COMPUTE_SIG_HASH_ITERATION(4);
1359         IXGBE_COMPUTE_SIG_HASH_ITERATION(5);
1360         IXGBE_COMPUTE_SIG_HASH_ITERATION(6);
1361         IXGBE_COMPUTE_SIG_HASH_ITERATION(7);
1362         IXGBE_COMPUTE_SIG_HASH_ITERATION(8);
1363         IXGBE_COMPUTE_SIG_HASH_ITERATION(9);
1364         IXGBE_COMPUTE_SIG_HASH_ITERATION(10);
```



```
1365         IXGBE_COMPUTE_SIG_HASH_ITERATION(11);
1366         IXGBE_COMPUTE_SIG_HASH_ITERATION(12);
1367         IXGBE_COMPUTE_SIG_HASH_ITERATION(13);
1368         IXGBE_COMPUTE_SIG_HASH_ITERATION(14);
1369         IXGBE_COMPUTE_SIG_HASH_ITERATION(15);

1371         /* combine common_hash result with signature and bucket hashes */
1372         bucket_hash ^= common_hash;
1373         bucket_hash &= IXGBE_ATR_HASH_MASK;

1375         sig_hash ^= common_hash << 16;
1376         sig_hash &= IXGBE_ATR_HASH_MASK << 16;

1378         /* return completed signature hash */
1379         return sig_hash ^ bucket_hash;
1380 }
______unchanged_portion_omitted_

1435 #if lint
1436 #define IXGBE_COMPUTE_BKT_HASH_ITERATION(_n)
1437 #else
1438 #define IXGBE_COMPUTE_BKT_HASH_ITERATION(_n) \
1439 do { \
1440         u32 n = (_n); \
1441         if (IXGBE_ATR_BUCKET_HASH_KEY & (0x01 << n)) \
1442                 bucket_hash ^= lo_hash_dword >> n; \
1443         if (IXGBE_ATR_BUCKET_HASH_KEY & (0x01 << (n + 16))) \
1444                 bucket_hash ^= hi_hash_dword >> n; \
1445 } while (0);
1446 #endif

1447 /**
1448  *  ixgbe_atr_compute_perfect_hash_82599 - Compute the perfect filter hash
1449  *  @atr_input: input bitstream to compute the hash on
1450  *  @input_mask: mask for the input bitstream
1451  *
1452  *  This function serves two main purposes.  First it applys the input_mask
1453  *  to the atr_input resulting in a cleaned up atr_input data stream.
1454  *  Secondly it computes the hash and stores it in the bkt_hash field at
1455  *  the end of the input byte stream.  This way it will be available for
1456  *  future use without needing to recompute the hash.
1457  **/
1458 void ixgbe_atr_compute_perfect_hash_82599(union ixgbe_atr_input *input,
1459                                           union ixgbe_atr_input *input_mask)
1460 {

1462         u32 hi_hash_dword, lo_hash_dword, flow_vm_vlan;
1463         u32 bucket_hash = 0;

1465         /* Apply masks to input data */
1466         input->dword_stream[0]  &= input_mask->dword_stream[0];
1467         input->dword_stream[1]  &= input_mask->dword_stream[1];
1468         input->dword_stream[2]  &= input_mask->dword_stream[2];
1469         input->dword_stream[3]  &= input_mask->dword_stream[3];
1470         input->dword_stream[4]  &= input_mask->dword_stream[4];
1471         input->dword_stream[5]  &= input_mask->dword_stream[5];
1472         input->dword_stream[6]  &= input_mask->dword_stream[6];
1473         input->dword_stream[7]  &= input_mask->dword_stream[7];
1474         input->dword_stream[8]  &= input_mask->dword_stream[8];
1475         input->dword_stream[9]  &= input_mask->dword_stream[9];
1476         input->dword_stream[10] &= input_mask->dword_stream[10];

1478         /* record the flow_vm_vlan bits as they are a key part to the hash */
1479         flow_vm_vlan = IXGBE_NTOHL(input->dword_stream[0]);

1481         /* generate common hash dword */
```

```
1482         hi_hash_dword = IXGBE_NTOHL(input->dword_stream[1] ^
1483                                     input->dword_stream[2] ^
1484                                     input->dword_stream[3] ^
1485                                     input->dword_stream[4] ^
1486                                     input->dword_stream[5] ^
1487                                     input->dword_stream[6] ^
1488                                     input->dword_stream[7] ^
1489                                     input->dword_stream[8] ^
1490                                     input->dword_stream[9] ^
1491                                     input->dword_stream[10]);

1493         /* low dword is word swapped version of common */
1494         lo_hash_dword = (hi_hash_dword >> 16) | (hi_hash_dword << 16);

1496         /* apply flow ID/VM pool/VLAN ID bits to hash words */
1497         hi_hash_dword ^= flow_vm_vlan ^ (flow_vm_vlan >> 16);

1499         /* Process bits 0 and 16 */
1500         IXGBE_COMPUTE_BKT_HASH_ITERATION(0);

1502         /*
1503          * apply flow ID/VM pool/VLAN ID bits to lo hash dword, we had to
1504          * delay this because bit 0 of the stream should not be processed
1505          * so we do not add the vlan until after bit 0 was processed
1506          */
1507         lo_hash_dword ^= flow_vm_vlan ^ (flow_vm_vlan << 16);

1509         /* Process remaining 30 bit of the key */
1510         IXGBE_COMPUTE_BKT_HASH_ITERATION(1);
1511         IXGBE_COMPUTE_BKT_HASH_ITERATION(2);
1512         IXGBE_COMPUTE_BKT_HASH_ITERATION(3);
1513         IXGBE_COMPUTE_BKT_HASH_ITERATION(4);
1514         IXGBE_COMPUTE_BKT_HASH_ITERATION(5);
1515         IXGBE_COMPUTE_BKT_HASH_ITERATION(6);
1516         IXGBE_COMPUTE_BKT_HASH_ITERATION(7);
1517         IXGBE_COMPUTE_BKT_HASH_ITERATION(8);
1518         IXGBE_COMPUTE_BKT_HASH_ITERATION(9);
1519         IXGBE_COMPUTE_BKT_HASH_ITERATION(10);
1520         IXGBE_COMPUTE_BKT_HASH_ITERATION(11);
1521         IXGBE_COMPUTE_BKT_HASH_ITERATION(12);
1522         IXGBE_COMPUTE_BKT_HASH_ITERATION(13);
1523         IXGBE_COMPUTE_BKT_HASH_ITERATION(14);
1524         IXGBE_COMPUTE_BKT_HASH_ITERATION(15);

1526         /*
1527          * Limit hash to 13 bits since max bucket count is 8K.
1528          * Store result at the end of the input stream.
1529          */
1530         input->formatted.bkt_hash = bucket_hash & 0x1FFF;
1531 }
```

*unchanged_portion_omitted_*

```
1553 /*
1554  * These two macros are meant to address the fact that we have registers
1555  * that are either all or in part big-endian.  As a result on big-endian
1556  * systems we will end up byte swapping the value to little-endian before
1557  * it is byte swapped again and written to the hardware in the original
1558  * big-endian format.
1559  */
1560 #define IXGBE_STORE_AS_BE32(_value) \
1561         ((((u32)(_value) >> 24) | (((u32)(_value) & 0x00FF0000) >> 8) | \
1562          (((u32)(_value) & 0x0000FF00) << 8) | ((u32)(_value) << 24))

1564 #define IXGBE_WRITE_REG_BE32(a, reg, value) \
1565         IXGBE_WRITE_REG((a), (reg), IXGBE_STORE_AS_BE32(IXGBE_NTOHL(value)))
```



```
1567 #define IXGBE_STORE_AS_BE16(_value) \
1568         IXGBE_NTOHS(((u16)(_value) >> 8) | ((u16)(_value) << 8))

1570 s32 ixgbe_fdir_set_input_mask_82599(struct ixgbe_hw *hw,
1571                                     union ixgbe_atr_input *input_mask)
1572 {
1573         /* mask IPv6 since it is currently not supported */
1574         u32 fdirm = IXGBE_FDIRM_DIPv6;
1575         u32 fdirtcpm;

1577         DEBUGFUNC("ixgbe_fdir_set_atr_input_mask_82599");

1579         /*
1580          * Program the relevant mask registers.  If src/dst_port or src/dst_addr
1581          * are zero, then assume a full mask for that field.  Also assume that
1582          * a VLAN of 0 is unspecified, so mask that out as well.  L4type
1583          * cannot be masked out in this implementation.
1584          *
1585          * This also assumes IPv4 only.  IPv6 masking isn't supported at this
1586          * point in time.
1587          */

1589         /* verify bucket hash is cleared on hash generation */
1590         if (input_mask->formatted.bkt_hash)
1591                 DEBUGOUT(" bucket hash should always be 0 in mask\n");

1593         /* Program FDIRM and verify partial masks */
1594         switch (input_mask->formatted.vm_pool & 0x7F) {
1595         case 0x0:
1596                 fdirm |= IXGBE_FDIRM_POOL;
1597                 /* FALLTHRU */
1598         case 0x7F:
1599                 break;
1600         default:
1601                 DEBUGOUT(" Error on vm pool mask\n");
1602                 return IXGBE_ERR_CONFIG;
1603         }

1605         switch (input_mask->formatted.flow_type & IXGBE_ATR_L4TYPE_MASK) {
1606         case 0x0:
1607                 fdirm |= IXGBE_FDIRM_L4P;
1608                 if (input_mask->formatted.dst_port ||
1609                     input_mask->formatted.src_port) {
1610                         DEBUGOUT(" Error on src/dst port mask\n");
1611                         return IXGBE_ERR_CONFIG;
1612                 }
1613                 /* FALLTHRU */
1614         case IXGBE_ATR_L4TYPE_MASK:
1615                 break;
1616         default:
1617                 DEBUGOUT(" Error on flow type mask\n");
1618                 return IXGBE_ERR_CONFIG;
1619         }

1621         switch (IXGBE_NTOHS(input_mask->formatted.vlan_id) & 0xEFFF) {
1622         case 0x0000:
1623                 /* mask VLAN ID, fall through to mask VLAN priority */
1624                 fdirm |= IXGBE_FDIRM_VLANID;
1625                 /* FALLTHRU */
1626         case 0x0FFF:
1627                 /* mask VLAN priority */
1628                 fdirm |= IXGBE_FDIRM_VLANP;
1629                 break;
1630         case 0xE000:
1631                 /* mask VLAN ID only, fall through */
1632                 fdirm |= IXGBE_FDIRM_VLANID;
```

```
1633                 /* FALLTHRU */
1634         case 0xEFFF:
1635                 /* no VLAN fields masked */
1636                 break;
1637         default:
1638                 DEBUGOUT(" Error on VLAN mask\n");
1639                 return IXGBE_ERR_CONFIG;
1640         }

1642         switch (input_mask->formatted.flex_bytes & 0xFFFF) {
1643         case 0x0000:
1644                 /* Mask Flex Bytes, fall through */
1645                 fdirm |= IXGBE_FDIRM_FLEX;
1646                 /* FALLTHRU */
1647         case 0xFFFF:
1648                 break;
1649         default:
1650                 DEBUGOUT(" Error on flexible byte mask\n");
1651                 return IXGBE_ERR_CONFIG;
1652         }

1654         /* Now mask VM pool and destination IPv6 - bits 5 and 2 */
1655         IXGBE_WRITE_REG(hw, IXGBE_FDIRM, fdirm);

1657         /* store the TCP/UDP port masks, bit reversed from port layout */
1658         fdirtcpm = ixgbe_get_fdirtcpm_82599(input_mask);

1660         /* write both the same so that UDP and TCP use the same mask */
1661         IXGBE_WRITE_REG(hw, IXGBE_FDIRTCPM, ~fdirtcpm);
1662         IXGBE_WRITE_REG(hw, IXGBE_FDIRUDPM, ~fdirtcpm);

1664         /* store source and destination IP masks (big-enian) */
1665         IXGBE_WRITE_REG_BE32(hw, IXGBE_FDIRSIP4M,
1666                              ~input_mask->formatted.src_ip[0]);
1667         IXGBE_WRITE_REG_BE32(hw, IXGBE_FDIRDIP4M,
1668                              ~input_mask->formatted.dst_ip[0]);

1670         return IXGBE_SUCCESS;
1671 }
```

*unchanged_portion_omitted_*

```
1775 /**
1776  *  ixgbe_fdir_add_perfect_filter_82599 - Adds a perfect filter
1777  *  @hw: pointer to hardware structure
1778  *  @input: input bitstream
1779  *  @input_mask: mask for the input bitstream
1780  *  @soft_id: software index for the filters
1781  *  @queue: queue index to direct traffic to
1782  *
1783  *  Note that the caller to this function must lock before calling, since the
1784  *  hardware writes must be protected from one another.
1785  **/
1786 s32 ixgbe_fdir_add_perfect_filter_82599(struct ixgbe_hw *hw,
1787                                         union ixgbe_atr_input *input,
1788                                         union ixgbe_atr_input *input_mask,
1789                                         u16 soft_id, u8 queue)
1790 {
1791         s32 err = IXGBE_ERR_CONFIG;

1793         DEBUGFUNC("ixgbe_fdir_add_perfect_filter_82599");

1795         /*
1796          * Check flow_type formatting, and bail out before we touch the hardware
1797          * if there's a configuration issue
1798          */
1799         switch (input->formatted.flow_type) {
```



```
1800         case IXGBE_ATR_FLOW_TYPE_IPV4:
1801                 input_mask->formatted.flow_type = IXGBE_ATR_L4TYPE_IPV6_MASK;
1802                 if (input->formatted.dst_port || input->formatted.src_port) {
1803                         DEBUGOUT(" Error on src/dst port\n");
1804                         return IXGBE_ERR_CONFIG;
1805                 }
1806                 break;
1807         case IXGBE_ATR_FLOW_TYPE_SCTPV4:
1808                 if (input->formatted.dst_port || input->formatted.src_port) {
1809                         DEBUGOUT(" Error on src/dst port\n");
1810                         return IXGBE_ERR_CONFIG;
1811                 }
1812                 /* FALLTHRU */
1813         case IXGBE_ATR_FLOW_TYPE_TCPV4:
1814         case IXGBE_ATR_FLOW_TYPE_UDPV4:
1815                 input_mask->formatted.flow_type = IXGBE_ATR_L4TYPE_IPV6_MASK |
1816                                                   IXGBE_ATR_L4TYPE_MASK;
1817                 break;
1818         default:
1819                 DEBUGOUT(" Error on flow type input\n");
1820                 return err;
1821         }

1823         /* program input mask into the HW */
1824         err = ixgbe_fdir_set_input_mask_82599(hw, input_mask);
1825         if (err)
1826                 return err;

1828         /* apply mask and compute/store hash */
1829         ixgbe_atr_compute_perfect_hash_82599(input, input_mask);

1831         /* program filters to filter memory */
1832         return ixgbe_fdir_write_perfect_filter_82599(hw, input,
1833                                                     soft_id, queue);
1834 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   35536 Fri Jul 27 22:36:04 2012
new/usr/src/uts/common/io/ixgbe/ixgbe_api.c
3014 Intel X540 Support (fix lint)
**********************************************************
/******************************************************************************

  Copyright (c) 2001-2012, Intel Corporation
  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,
      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.

   3. Neither the name of the Intel Corporation nor the names of its
      contributors may be used to endorse or promote products derived from
      this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS 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 COPYRIGHT OWNER 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.

******************************************************************************/
/*$FreeBSD: src/sys/dev/ixgbe/ixgbe_api.c,v 1.13 2012/07/05 20:51:44 jfv Exp $*/

#include "ixgbe_api.h"
#include "ixgbe_common.h"

/**
 *  ixgbe_init_shared_code - Initialize the shared code
 *  @hw: pointer to hardware structure
 *
 *  This will assign function pointers and assign the MAC type and PHY code.
 *  Does not touch the hardware. This function must be called prior to any
 *  other function in the shared code. The ixgbe_hw structure should be
 *  memset to 0 prior to calling this function.  The following fields in
 *  hw structure should be filled in prior to calling this function:
 *  hw_addr, back, device_id, vendor_id, subsystem_device_id,
 *  subsystem_vendor_id, and revision_id
 **/
s32 ixgbe_init_shared_code(struct ixgbe_hw *hw)
{
        s32 status;

        DEBUGFUNC("ixgbe_init_shared_code");

        /*
         * Set the mac type
         */
        status = ixgbe_set_mac_type(hw);
        if (status != IXGBE_SUCCESS)
                return (status);
```



```
        ixgbe_set_mac_type(hw);

        switch (hw->mac.type) {
        case ixgbe_mac_82598EB:
                status = ixgbe_init_ops_82598(hw);
                break;
        case ixgbe_mac_82599EB:
                status = ixgbe_init_ops_82599(hw);
                break;
        case ixgbe_mac_X540:
                status = ixgbe_init_ops_X540(hw);
                break;
        default:
                status = IXGBE_ERR_DEVICE_NOT_SUPPORTED;
                break;
        }

        return status;
}
_____unchanged_portion_omitted_

/**
 *  ixgbe_read_phy_reg - Read PHY register
 *  @hw: pointer to hardware structure
 *  @reg_addr: 32 bit address of PHY register to read
 *  @phy_data: Pointer to read data from PHY register
 *
 *  Reads a value from a specified PHY register
 **/
s32 ixgbe_read_phy_reg(struct ixgbe_hw *hw, u32 reg_addr, u32 device_type,
                       u16 *phy_data)
{
        s32 status;

        if (hw->phy.id == 0)
                status = ixgbe_identify_phy(hw);
        else
                status = IXGBE_SUCCESS;
                ixgbe_identify_phy(hw);

        if (status == IXGBE_SUCCESS) {
                status = ixgbe_call_func(hw, hw->phy.ops.read_reg,
                    (hw, reg_addr, device_type, phy_data),
                    IXGBE_NOT_IMPLEMENTED);
        }
        return (status);
        return ixgbe_call_func(hw, hw->phy.ops.read_reg, (hw, reg_addr,
                               device_type, phy_data), IXGBE_NOT_IMPLEMENTED);
}

/**
 *  ixgbe_write_phy_reg - Write PHY register
 *  @hw: pointer to hardware structure
 *  @reg_addr: 32 bit PHY register to write
 *  @phy_data: Data to write to the PHY register
 *
 *  Writes a value to specified PHY register
 **/
s32 ixgbe_write_phy_reg(struct ixgbe_hw *hw, u32 reg_addr, u32 device_type,
                        u16 phy_data)
{
        s32 status;

        if (hw->phy.id == 0)
                status = ixgbe_identify_phy(hw);
        else
```

```
484                 status = IXGBE_SUCCESS;
470                 ixgbe_identify_phy(hw);

486         if (status == IXGBE_SUCCESS) {
487                 status = ixgbe_call_func(hw, hw->phy.ops.write_reg,
488                     (hw, reg_addr, device_type, phy_data),
489                     IXGBE_NOT_IMPLEMENTED);
490         }

492         return status;
472         return ixgbe_call_func(hw, hw->phy.ops.write_reg, (hw, reg_addr,
473                                device_type, phy_data), IXGBE_NOT_IMPLEMENTED);
493 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  117362 Fri Jul 27 22:36:07 2012
new/usr/src/uts/common/io/ixgbe/ixgbe_common.c
3014 Intel X540 Support (fix lint)
**********************************************************
______unchanged_portion_omitted_

 422 /**
 423  *  ixgbe_clear_hw_cntrs_generic - Generic clear hardware counters
 424  *  @hw: pointer to hardware structure
 425  *
 426  *  Clears all hardware statistics counters by reading them from the hardware
 427  *  Statistics counters are clear on read.
 428  **/
 429 s32 ixgbe_clear_hw_cntrs_generic(struct ixgbe_hw *hw)
 430 {
 431         u16 i = 0;

 433         DEBUGFUNC("ixgbe_clear_hw_cntrs_generic");

 435         (void) IXGBE_READ_REG(hw, IXGBE_CRCERRS);
 436         (void) IXGBE_READ_REG(hw, IXGBE_ILLERRC);
 437         (void) IXGBE_READ_REG(hw, IXGBE_ERRBC);
 438         (void) IXGBE_READ_REG(hw, IXGBE_MSPDC);
 435         IXGBE_READ_REG(hw, IXGBE_CRCERRS);
 436         IXGBE_READ_REG(hw, IXGBE_ILLERRC);
 437         IXGBE_READ_REG(hw, IXGBE_ERRBC);
 438         IXGBE_READ_REG(hw, IXGBE_MSPDC);
 439         for (i = 0; i < 8; i++)
 440                 (void) IXGBE_READ_REG(hw, IXGBE_MPC(i));
 440                 IXGBE_READ_REG(hw, IXGBE_MPC(i));

 442         (void) IXGBE_READ_REG(hw, IXGBE_MLFC);
 443         (void) IXGBE_READ_REG(hw, IXGBE_MRFC);
 444         (void) IXGBE_READ_REG(hw, IXGBE_RLEC);
 445         (void) IXGBE_READ_REG(hw, IXGBE_LXONTXC);
 446         (void) IXGBE_READ_REG(hw, IXGBE_LXOFFTXC);
 442         IXGBE_READ_REG(hw, IXGBE_MLFC);
 443         IXGBE_READ_REG(hw, IXGBE_MRFC);
 444         IXGBE_READ_REG(hw, IXGBE_RLEC);
 445         IXGBE_READ_REG(hw, IXGBE_LXONTXC);
 446         IXGBE_READ_REG(hw, IXGBE_LXOFFTXC);
 447         if (hw->mac.type >= ixgbe_mac_82599EB) {
 448                 (void) IXGBE_READ_REG(hw, IXGBE_LXONRXCNT);
 449                 (void) IXGBE_READ_REG(hw, IXGBE_LXOFFRXCNT);
 448                 IXGBE_READ_REG(hw, IXGBE_LXONRXCNT);
 449                 IXGBE_READ_REG(hw, IXGBE_LXOFFRXCNT);
 450         } else {
 451                 (void) IXGBE_READ_REG(hw, IXGBE_LXONRXC);
 452                 (void) IXGBE_READ_REG(hw, IXGBE_LXOFFRXC);
 451                 IXGBE_READ_REG(hw, IXGBE_LXONRXC);
 452                 IXGBE_READ_REG(hw, IXGBE_LXOFFRXC);
 453         }

 455         for (i = 0; i < 8; i++) {
 456                 (void) IXGBE_READ_REG(hw, IXGBE_PXONTXC(i));
 457                 (void) IXGBE_READ_REG(hw, IXGBE_PXOFFTXC(i));
 456                 IXGBE_READ_REG(hw, IXGBE_PXONTXC(i));
 457                 IXGBE_READ_REG(hw, IXGBE_PXOFFTXC(i));
 458                 if (hw->mac.type >= ixgbe_mac_82599EB) {
 459                         (void) IXGBE_READ_REG(hw, IXGBE_PXONRXCNT(i));
 460                         (void) IXGBE_READ_REG(hw, IXGBE_PXOFFRXCNT(i));
 459                         IXGBE_READ_REG(hw, IXGBE_PXONRXCNT(i));
 460                         IXGBE_READ_REG(hw, IXGBE_PXOFFRXCNT(i));
 461                 } else {
 462                         (void) IXGBE_READ_REG(hw, IXGBE_PXONRXC(i));
```



```
 463                         (void) IXGBE_READ_REG(hw, IXGBE_PXOFFRXC(i));
 462                         IXGBE_READ_REG(hw, IXGBE_PXONRXC(i));
 463                         IXGBE_READ_REG(hw, IXGBE_PXOFFRXC(i));
 464                 }
 465         }
 466         if (hw->mac.type >= ixgbe_mac_82599EB)
 467                 for (i = 0; i < 8; i++)
 468                         (void) IXGBE_READ_REG(hw, IXGBE_PXON2OFFCNT(i));
 469         (void) IXGBE_READ_REG(hw, IXGBE_PRC64);
 470         (void) IXGBE_READ_REG(hw, IXGBE_PRC127);
 471         (void) IXGBE_READ_REG(hw, IXGBE_PRC255);
 472         (void) IXGBE_READ_REG(hw, IXGBE_PRC511);
 473         (void) IXGBE_READ_REG(hw, IXGBE_PRC1023);
 474         (void) IXGBE_READ_REG(hw, IXGBE_PRC1522);
 475         (void) IXGBE_READ_REG(hw, IXGBE_GPRC);
 476         (void) IXGBE_READ_REG(hw, IXGBE_BPRC);
 477         (void) IXGBE_READ_REG(hw, IXGBE_MPRC);
 478         (void) IXGBE_READ_REG(hw, IXGBE_GPTC);
 479         (void) IXGBE_READ_REG(hw, IXGBE_GORCL);
 480         (void) IXGBE_READ_REG(hw, IXGBE_GORCH);
 481         (void) IXGBE_READ_REG(hw, IXGBE_GOTCL);
 482         (void) IXGBE_READ_REG(hw, IXGBE_GOTCH);
 468                         IXGBE_READ_REG(hw, IXGBE_PXON2OFFCNT(i));
 469         IXGBE_READ_REG(hw, IXGBE_PRC64);
 470         IXGBE_READ_REG(hw, IXGBE_PRC127);
 471         IXGBE_READ_REG(hw, IXGBE_PRC255);
 472         IXGBE_READ_REG(hw, IXGBE_PRC511);
 473         IXGBE_READ_REG(hw, IXGBE_PRC1023);
 474         IXGBE_READ_REG(hw, IXGBE_PRC1522);
 475         IXGBE_READ_REG(hw, IXGBE_GPRC);
 476         IXGBE_READ_REG(hw, IXGBE_BPRC);
 477         IXGBE_READ_REG(hw, IXGBE_MPRC);
 478         IXGBE_READ_REG(hw, IXGBE_GPTC);
 479         IXGBE_READ_REG(hw, IXGBE_GORCL);
 480         IXGBE_READ_REG(hw, IXGBE_GORCH);
 481         IXGBE_READ_REG(hw, IXGBE_GOTCL);
 482         IXGBE_READ_REG(hw, IXGBE_GOTCH);
 483         if (hw->mac.type == ixgbe_mac_82598EB)
 484                 for (i = 0; i < 8; i++)
 485                         (void) IXGBE_READ_REG(hw, IXGBE_RNBC(i));
 486         (void) IXGBE_READ_REG(hw, IXGBE_RUC);
 487         (void) IXGBE_READ_REG(hw, IXGBE_RFC);
 488         (void) IXGBE_READ_REG(hw, IXGBE_ROC);
 489         (void) IXGBE_READ_REG(hw, IXGBE_RJC);
 490         (void) IXGBE_READ_REG(hw, IXGBE_MNGPRC);
 491         (void) IXGBE_READ_REG(hw, IXGBE_MNGPDC);
 492         (void) IXGBE_READ_REG(hw, IXGBE_MNGPTC);
 493         (void) IXGBE_READ_REG(hw, IXGBE_TORL);
 494         (void) IXGBE_READ_REG(hw, IXGBE_TORH);
 495         (void) IXGBE_READ_REG(hw, IXGBE_TPR);
 496         (void) IXGBE_READ_REG(hw, IXGBE_TPT);
 497         (void) IXGBE_READ_REG(hw, IXGBE_PTC64);
 498         (void) IXGBE_READ_REG(hw, IXGBE_PTC127);
 499         (void) IXGBE_READ_REG(hw, IXGBE_PTC255);
 500         (void) IXGBE_READ_REG(hw, IXGBE_PTC511);
 501         (void) IXGBE_READ_REG(hw, IXGBE_PTC1023);
 502         (void) IXGBE_READ_REG(hw, IXGBE_PTC1522);
 503         (void) IXGBE_READ_REG(hw, IXGBE_MPTC);
 504         (void) IXGBE_READ_REG(hw, IXGBE_BPTC);
 485                         IXGBE_READ_REG(hw, IXGBE_RNBC(i));
 486         IXGBE_READ_REG(hw, IXGBE_RUC);
 487         IXGBE_READ_REG(hw, IXGBE_RFC);
 488         IXGBE_READ_REG(hw, IXGBE_ROC);
 489         IXGBE_READ_REG(hw, IXGBE_RJC);
 490         IXGBE_READ_REG(hw, IXGBE_MNGPRC);
 491         IXGBE_READ_REG(hw, IXGBE_MNGPDC);
```

```
492         IXGBE_READ_REG(hw, IXGBE_MNGPTC);
493         IXGBE_READ_REG(hw, IXGBE_TORL);
494         IXGBE_READ_REG(hw, IXGBE_TORH);
495         IXGBE_READ_REG(hw, IXGBE_TPR);
496         IXGBE_READ_REG(hw, IXGBE_TPT);
497         IXGBE_READ_REG(hw, IXGBE_PTC64);
498         IXGBE_READ_REG(hw, IXGBE_PTC127);
499         IXGBE_READ_REG(hw, IXGBE_PTC255);
500         IXGBE_READ_REG(hw, IXGBE_PTC511);
501         IXGBE_READ_REG(hw, IXGBE_PTC1023);
502         IXGBE_READ_REG(hw, IXGBE_PTC1522);
503         IXGBE_READ_REG(hw, IXGBE_MPTC);
504         IXGBE_READ_REG(hw, IXGBE_BPTC);
505         for (i = 0; i < 16; i++) {
506                 (void) IXGBE_READ_REG(hw, IXGBE_QPRC(i));
507                 (void) IXGBE_READ_REG(hw, IXGBE_QPTC(i));
506                 IXGBE_READ_REG(hw, IXGBE_QPRC(i));
507                 IXGBE_READ_REG(hw, IXGBE_QPTC(i));
508                 if (hw->mac.type >= ixgbe_mac_82599EB) {
509                         (void) IXGBE_READ_REG(hw, IXGBE_QBRC_L(i));
510                         (void) IXGBE_READ_REG(hw, IXGBE_QBRC_H(i));
511                         (void) IXGBE_READ_REG(hw, IXGBE_QBTC_L(i));
512                         (void) IXGBE_READ_REG(hw, IXGBE_QBTC_H(i));
513                         (void) IXGBE_READ_REG(hw, IXGBE_QPRDC(i));
509                         IXGBE_READ_REG(hw, IXGBE_QBRC_L(i));
510                         IXGBE_READ_REG(hw, IXGBE_QBRC_H(i));
511                         IXGBE_READ_REG(hw, IXGBE_QBTC_L(i));
512                         IXGBE_READ_REG(hw, IXGBE_QBTC_H(i));
513                         IXGBE_READ_REG(hw, IXGBE_QPRDC(i));
514                 } else {
515                         (void) IXGBE_READ_REG(hw, IXGBE_QBRC(i));
516                         (void) IXGBE_READ_REG(hw, IXGBE_QBTC(i));
515                         IXGBE_READ_REG(hw, IXGBE_QBRC(i));
516                         IXGBE_READ_REG(hw, IXGBE_QBTC(i));
517                 }
518         }

520         if (hw->mac.type == ixgbe_mac_X540) {
521                 if (hw->phy.id == 0)
522                         (void) ixgbe_identify_phy(hw);
522                         ixgbe_identify_phy(hw);
523                 hw->phy.ops.read_reg(hw, IXGBE_PCRC8ECL,
524                                      IXGBE_MDIO_PCS_DEV_TYPE, &i);
525                 hw->phy.ops.read_reg(hw, IXGBE_PCRC8ECH,
526                                      IXGBE_MDIO_PCS_DEV_TYPE, &i);
527                 hw->phy.ops.read_reg(hw, IXGBE_LDPCECL,
528                                      IXGBE_MDIO_PCS_DEV_TYPE, &i);
529                 hw->phy.ops.read_reg(hw, IXGBE_LDPCECH,
530                                      IXGBE_MDIO_PCS_DEV_TYPE, &i);
531         }

533         return IXGBE_SUCCESS;
534 }
```

*unchanged_portion_omitted_*

```
790 /**
791  *  ixgbe_stop_adapter_generic - Generic stop Tx/Rx units
792  *  @hw: pointer to hardware structure
793  *
794  *  Sets the adapter_stopped flag within ixgbe_hw struct. Clears interrupts,
795  *  disables transmit and receive units. The adapter_stopped flag is used by
796  *  the shared code and drivers to determine if the adapter is in a stopped
797  *  state and should not touch the hardware.
798  **/
799 s32 ixgbe_stop_adapter_generic(struct ixgbe_hw *hw)
800 {
```



```
801         u32 reg_val;
802         u16 i;

804         DEBUGFUNC("ixgbe_stop_adapter_generic");

806         /*
807          * Set the adapter_stopped flag so other driver functions stop touching
808          * the hardware
809          */
810         hw->adapter_stopped = TRUE;

812         /* Disable the receive unit */
813         IXGBE_WRITE_REG(hw, IXGBE_RXCTRL, 0);

815         /* Clear interrupt mask to stop interrupts from being generated */
816         IXGBE_WRITE_REG(hw, IXGBE_EIMC, IXGBE_IRQ_CLEAR_MASK);

818         /* Clear any pending interrupts, flush previous writes */
819         (void) IXGBE_READ_REG(hw, IXGBE_EICR);
819         IXGBE_READ_REG(hw, IXGBE_EICR);

821         /* Disable the transmit unit.  Each queue must be disabled. */
822         for (i = 0; i < hw->mac.max_tx_queues; i++)
823                 IXGBE_WRITE_REG(hw, IXGBE_TXDCTL(i), IXGBE_TXDCTL_SWFLSH);

825         /* Disable the receive unit by stopping each queue */
826         for (i = 0; i < hw->mac.max_rx_queues; i++) {
827                 reg_val = IXGBE_READ_REG(hw, IXGBE_RXDCTL(i));
828                 reg_val &= ~IXGBE_RXDCTL_ENABLE;
829                 reg_val |= IXGBE_RXDCTL_SWFLSH;
830                 IXGBE_WRITE_REG(hw, IXGBE_RXDCTL(i), reg_val);
831         }

833         /* flush all queues disables */
834         IXGBE_WRITE_FLUSH(hw);
835         msec_delay(2);

837         /*
838          * Prevent the PCI-E bus from from hanging by disabling PCI-E master
839          * access and verify no pending requests
840          */
841         return ixgbe_disable_pcie_master(hw);
842 }
```

*unchanged_portion_omitted_*

```
937 /**
938  *  ixgbe_write_eeprom_buffer_bit_bang_generic - Write EEPROM using bit-bang
939  *  @hw: pointer to hardware structure
940  *  @offset: offset within the EEPROM to write
941  *  @words: number of word(s)
942  *  @data: 16 bit word(s) to write to EEPROM
943  *
944  *  Reads 16 bit word(s) from EEPROM through bit-bang method
945  **/
946 s32 ixgbe_write_eeprom_buffer_bit_bang_generic(struct ixgbe_hw *hw, u16 offset,
947                                                u16 words, u16 *data)
948 {
949         s32 status = IXGBE_SUCCESS;
950         u16 i, count;

952         DEBUGFUNC("ixgbe_write_eeprom_buffer_bit_bang_generic");

954         hw->eeprom.ops.init_params(hw);

956         if (words == 0) {
957                 status = IXGBE_ERR_INVALID_ARGUMENT;
```

```
 958                 goto out;
 959         }

 961         if (offset + words > hw->eeprom.word_size) {
 962                 status = IXGBE_ERR_EEPROM;
 963                 goto out;
 964         }

 966         /*
 967          * The EEPROM page size cannot be queried from the chip. We do lazy
 968          * initialization. It is worth to do that when we write large buffer.
 969          */
 970         if ((hw->eeprom.word_page_size == 0) &&
 971             (words > IXGBE_EEPROM_PAGE_SIZE_MAX))
 972                 status = ixgbe_detect_eeprom_page_size_generic(hw, offset);
 973         if (status != IXGBE_SUCCESS)
 974                 goto out;
 972                 ixgbe_detect_eeprom_page_size_generic(hw, offset);

 976         /*
 977          * We cannot hold synchronization semaphores for too long
 978          * to avoid other entity starvation. However it is more efficient
 979          * to read in bursts than synchronizing access for each word.
 980          */
 981         for (i = 0; i < words; i += IXGBE_EEPROM_RD_BUFFER_MAX_COUNT) {
 982                 count = (words - i) / IXGBE_EEPROM_RD_BUFFER_MAX_COUNT > 0 ?
 983                         IXGBE_EEPROM_RD_BUFFER_MAX_COUNT : (words - i);
 984                 status = ixgbe_write_eeprom_buffer_bit_bang(hw, offset + i,
 985                                                             count, &data[i]);

 987                 if (status != IXGBE_SUCCESS)
 988                         break;
 989         }

 991 out:
 992         return status;
 993 }
_____unchanged_portion_omitted_

2078 /**
2079  *  ixgbe_init_rx_addrs_generic - Initializes receive address filters.
2080  *  @hw: pointer to hardware structure
2081  *
2082  *  Places the MAC address in receive address register 0 and clears the rest
2083  *  of the receive address registers. Clears the multicast table. Assumes
2084  *  the receiver is in reset when the routine is called.
2085  **/
2086 s32 ixgbe_init_rx_addrs_generic(struct ixgbe_hw *hw)
2087 {
2088         u32 i;
2089         u32 rar_entries = hw->mac.num_rar_entries;

2091         DEBUGFUNC("ixgbe_init_rx_addrs_generic");

2093         /*
2094          * If the current mac address is valid, assume it is a software override
2095          * to the permanent address.
2096          * Otherwise, use the permanent address from the eeprom.
2097          */
2098         if (ixgbe_validate_mac_addr(hw->mac.addr) ==
2099             IXGBE_ERR_INVALID_MAC_ADDR) {
2100                 /* Get the MAC address from the RAR0 for later reference */
2101                 hw->mac.ops.get_mac_addr(hw, hw->mac.addr);

2103                 DEBUGOUT3(" Keeping Current RAR0 Addr =%.2X %.2X %.2X ",
2104                           hw->mac.addr[0], hw->mac.addr[1],
```



```
2105                           hw->mac.addr[2]);
2106                 DEBUGOUT3("%.2X %.2X %.2X\n", hw->mac.addr[3],
2107                           hw->mac.addr[4], hw->mac.addr[5]);
2108         } else {
2109                 /* Setup the receive address. */
2110                 DEBUGOUT("Overriding MAC Address in RAR[0]\n");
2111                 DEBUGOUT3(" New MAC Addr =%.2X %.2X %.2X ",
2112                           hw->mac.addr[0], hw->mac.addr[1],
2113                           hw->mac.addr[2]);
2114                 DEBUGOUT3("%.2X %.2X %.2X\n", hw->mac.addr[3],
2115                           hw->mac.addr[4], hw->mac.addr[5]);

2117                 hw->mac.ops.set_rar(hw, 0, hw->mac.addr, 0, IXGBE_RAH_AV);

2119                 /* clear VMDq pool/queue selection for RAR 0 */
2120                 hw->mac.ops.clear_vmdq(hw, 0, IXGBE_CLEAR_VMDQ_ALL);
2121         }
2122         hw->addr_ctrl.overflow_promisc = 0;

2124         hw->addr_ctrl.rar_used_count = 1;

2126         /* Zero out the other receive addresses. */
2127         DEBUGOUT1("Clearing RAR[1-%d]\n", rar_entries - 1);
2128         for (i = 1; i < rar_entries; i++) {
2129                 IXGBE_WRITE_REG(hw, IXGBE_RAL(i), 0);
2130                 IXGBE_WRITE_REG(hw, IXGBE_RAH(i), 0);
2131         }

2133         /* Clear the MTA */
2134         hw->addr_ctrl.mta_in_use = 0;
2135         IXGBE_WRITE_REG(hw, IXGBE_MCSTCTRL, hw->mac.mc_filter_type);

2137         DEBUGOUT(" Clearing MTA\n");
2138         for (i = 0; i < hw->mac.mcft_size; i++)
2139                 IXGBE_WRITE_REG(hw, IXGBE_MTA(i), 0);

2141         /* Should always be IXGBE_SUCCESS. */
2142         return ixgbe_init_uta_tables(hw);
2139         ixgbe_init_uta_tables(hw);

2141         return IXGBE_SUCCESS;
2143 }
_____unchanged_portion_omitted_

2324 /**
2325  *  ixgbe_update_mc_addr_list_generic - Updates MAC list of multicast addresses
2326  *  @hw: pointer to hardware structure
2327  *  @mc_addr_list: the list of new multicast addresses
2328  *  @mc_addr_count: number of addresses
2329  *  @next: iterator function to walk the multicast address list
2330  *  @clear: flag, when set clears the table beforehand
2331  *
2332  *  When the clear flag is set, the given list replaces any existing list.
2333  *  Hashes the given addresses into the multicast table.
2334  **/
2335 s32 ixgbe_update_mc_addr_list_generic(struct ixgbe_hw *hw, u8 *mc_addr_list,
2336                                       u32 mc_addr_count, ixgbe_mc_addr_itr next,
2337                                       bool clear)
2338 {
2339         u32 i;
2340         u32 vmdq;

2342         DEBUGFUNC("ixgbe_update_mc_addr_list_generic");

2344         /*
2345          * Set the new number of MC addresses that we are being requested to
```

```
2346          * use.
2347          */
2348         hw->addr_ctrl.num_mc_addrs = mc_addr_count;
2349         hw->addr_ctrl.mta_in_use = 0;

2351         /* Clear mta_shadow */
2352         if (clear) {
2353                 DEBUGOUT(" Clearing MTA\n");
2354                 (void) memset(&hw->mac.mta_shadow, 0,
2355                     sizeof(hw->mac.mta_shadow));
2353                 memset(&hw->mac.mta_shadow, 0, sizeof(hw->mac.mta_shadow));
2356         }

2358         /* Update mta_shadow */
2359         for (i = 0; i < mc_addr_count; i++) {
2360                 DEBUGOUT(" Adding the multicast addresses:\n");
2361                 ixgbe_set_mta(hw, next(hw, &mc_addr_list, &vmdq));
2362         }

2364         /* Enable mta */
2365         for (i = 0; i < hw->mac.mcft_size; i++)
2366                 IXGBE_WRITE_REG_ARRAY(hw, IXGBE_MTA(0), i,
2367                                       hw->mac.mta_shadow[i]);

2369         if (hw->addr_ctrl.mta_in_use > 0)
2370                 IXGBE_WRITE_REG(hw, IXGBE_MCSTCTRL,
2371                                 IXGBE_MCSTCTRL_MFE | hw->mac.mc_filter_type);

2373         DEBUGOUT("ixgbe_update_mc_addr_list_generic Complete\n");
2374         return IXGBE_SUCCESS;
2375 }
_____unchanged_portion_omitted_

2860 /**
2861  *  ixgbe_release_swfw_sync - Release SWFW semaphore
2862  *  @hw: pointer to hardware structure
2863  *  @mask: Mask to specify which semaphore to release
2864  *
2865  *  Releases the SWFW semaphore through the GSSR register for the specified
2866  *  function (CSR, PHY0, PHY1, EEPROM, Flash)
2867  **/
2868 void ixgbe_release_swfw_sync(struct ixgbe_hw *hw, u16 mask)
2869 {
2870         u32 gssr;
2871         u32 swmask = mask;

2873         DEBUGFUNC("ixgbe_release_swfw_sync");

2875         (void) ixgbe_get_eeprom_semaphore(hw);
2873         ixgbe_get_eeprom_semaphore(hw);

2877         gssr = IXGBE_READ_REG(hw, IXGBE_GSSR);
2878         gssr &= ~swmask;
2879         IXGBE_WRITE_REG(hw, IXGBE_GSSR, gssr);

2881         ixgbe_release_eeprom_semaphore(hw);
2882 }
_____unchanged_portion_omitted_

3042 /**
3043  *  ixgbe_get_san_mac_addr_generic - SAN MAC address retrieval from the EEPROM
3044  *  @hw: pointer to hardware structure
3045  *  @san_mac_addr: SAN MAC address
3046  *
3047  *  Reads the SAN MAC address from the EEPROM, if it's available.  This is
3048  *  per-port, so set_lan_id() must be called before reading the addresses.
```



```
3049  *  set_lan_id() is called by identify_sfp(), but this cannot be relied
3050  *  upon for non-SFP connections, so we must call it here.
3051  **/
3052 s32 ixgbe_get_san_mac_addr_generic(struct ixgbe_hw *hw, u8 *san_mac_addr)
3053 {
3054         u16 san_mac_data, san_mac_offset;
3055         u8 i;

3057         DEBUGFUNC("ixgbe_get_san_mac_addr_generic");

3059         /*
3060          * First read the EEPROM pointer to see if the MAC addresses are
3061          * available.  If they're not, no point in calling set_lan_id() here.
3062          */
3063         (void) ixgbe_get_san_mac_addr_offset(hw, &san_mac_offset);
3061         ixgbe_get_san_mac_addr_offset(hw, &san_mac_offset);

3065         if ((san_mac_offset == 0) || (san_mac_offset == 0xFFFF)) {
3066                 /*
3067                  * No addresses available in this EEPROM.  It's not an
3068                  * error though, so just wipe the local address and return.
3069                  */
3070                 for (i = 0; i < 6; i++)
3071                         san_mac_addr[i] = 0xFF;

3073                 goto san_mac_addr_out;
3074         }

3076         /* make sure we know which port we need to program */
3077         hw->mac.ops.set_lan_id(hw);
3078         /* apply the port offset to the address offset */
3079         (hw->bus.func) ? (san_mac_offset += IXGBE_SAN_MAC_ADDR_PORT1_OFFSET) :
3080                          (san_mac_offset += IXGBE_SAN_MAC_ADDR_PORT0_OFFSET);
3081         for (i = 0; i < 3; i++) {
3082                 hw->eeprom.ops.read(hw, san_mac_offset, &san_mac_data);
3083                 san_mac_addr[i * 2] = (u8)(san_mac_data);
3084                 san_mac_addr[i * 2 + 1] = (u8)(san_mac_data >> 8);
3085                 san_mac_offset++;
3086         }

3088 san_mac_addr_out:
3089         return IXGBE_SUCCESS;
3090 }

3092 /**
3093  *  ixgbe_set_san_mac_addr_generic - Write the SAN MAC address to the EEPROM
3094  *  @hw: pointer to hardware structure
3095  *  @san_mac_addr: SAN MAC address
3096  *
3097  *  Write a SAN MAC address to the EEPROM.
3098  **/
3099 s32 ixgbe_set_san_mac_addr_generic(struct ixgbe_hw *hw, u8 *san_mac_addr)
3100 {
3101         s32 status = IXGBE_SUCCESS;
3102         u16 san_mac_data, san_mac_offset;
3103         u8 i;

3105         DEBUGFUNC("ixgbe_set_san_mac_addr_generic");

3107         /* Look for SAN mac address pointer.  If not defined, return */
3108         (void) ixgbe_get_san_mac_addr_offset(hw, &san_mac_offset);
3106         ixgbe_get_san_mac_addr_offset(hw, &san_mac_offset);

3110         if ((san_mac_offset == 0) || (san_mac_offset == 0xFFFF)) {
3111                 status = IXGBE_ERR_NO_SAN_ADDR_PTR;
3112                 goto san_mac_addr_out;
```

```
3113         }

3115         /* Make sure we know which port we need to write */
3116         hw->mac.ops.set_lan_id(hw);
3117         /* Apply the port offset to the address offset */
3118         (hw->bus.func) ? (san_mac_offset += IXGBE_SAN_MAC_ADDR_PORT1_OFFSET) :
3119                          (san_mac_offset += IXGBE_SAN_MAC_ADDR_PORT0_OFFSET);

3121         for (i = 0; i < 3; i++) {
3122                 san_mac_data = (u16)((u16)(san_mac_addr[i * 2 + 1]) << 8);
3123                 san_mac_data |= (u16)(san_mac_addr[i * 2]);
3124                 hw->eeprom.ops.write(hw, san_mac_offset, san_mac_data);
3125                 san_mac_offset++;
3126         }

3128 san_mac_addr_out:
3129         return status;
3130 }
______unchanged_portion_omitted_

3172 /**
3173  *  ixgbe_insert_mac_addr_generic - Find a RAR for this mac address
3174  *  @hw: pointer to hardware structure
3175  *  @addr: Address to put into receive address register
3176  *  @vmdq: VMDq pool to assign
3177  *
3178  *  Puts an ethernet address into a receive address register, or
3179  *  finds the rar that it is aleady in; adds to the pool list
3180  **/
3181 s32 ixgbe_insert_mac_addr_generic(struct ixgbe_hw *hw, u8 *addr, u32 vmdq)
3182 {
3183         static const u32 NO_EMPTY_RAR_FOUND = 0xFFFFFFFF;
3184         u32 first_empty_rar = NO_EMPTY_RAR_FOUND;
3185         u32 rar;
3186         u32 rar_low, rar_high;
3187         u32 addr_low, addr_high;

3189         DEBUGFUNC("ixgbe_insert_mac_addr_generic");

3191         /* swap bytes for HW little endian */
3192         addr_low  = addr[0] | (addr[1] << 8)
3193                             | (addr[2] << 16)
3194                             | (addr[3] << 24);
3195         addr_high = addr[4] | (addr[5] << 8);

3197         /*
3198          * Either find the mac_id in rar or find the first empty space.
3199          * rar_highwater points to just after the highest currently used
3200          * rar in order to shorten the search.  It grows when we add a new
3201          * rar to the top.
3202          */
3203         for (rar = 0; rar < hw->mac.rar_highwater; rar++) {
3204                 rar_high = IXGBE_READ_REG(hw, IXGBE_RAH(rar));

3206                 if (((IXGBE_RAH_AV & rar_high) == 0)
3207                     && first_empty_rar == NO_EMPTY_RAR_FOUND) {
3208                         first_empty_rar = rar;
3209                 } else if ((rar_high & 0xFFFF) == addr_high) {
3210                         rar_low = IXGBE_READ_REG(hw, IXGBE_RAL(rar));
3211                         if (rar_low == addr_low)
3212                                 break;    /* found it already in the rars */
3213                 }
3214         }

3216         if (rar < hw->mac.rar_highwater) {
3217                 /* already there so just add to the pool bits */
```



```
3218                 (void) ixgbe_set_vmdq(hw, rar, vmdq);
3216                 ixgbe_set_vmdq(hw, rar, vmdq);
3219         } else if (first_empty_rar != NO_EMPTY_RAR_FOUND) {
3220                 /* stick it into first empty RAR slot we found */
3221                 rar = first_empty_rar;
3222                 (void) ixgbe_set_rar(hw, rar, addr, vmdq, IXGBE_RAH_AV);
3220                 ixgbe_set_rar(hw, rar, addr, vmdq, IXGBE_RAH_AV);
3223         } else if (rar == hw->mac.rar_highwater) {
3224                 /* add it to the top of the list and inc the highwater mark */
3225                 (void) ixgbe_set_rar(hw, rar, addr, vmdq, IXGBE_RAH_AV);
3223                 ixgbe_set_rar(hw, rar, addr, vmdq, IXGBE_RAH_AV);
3226                 hw->mac.rar_highwater++;
3227         } else if (rar >= hw->mac.num_rar_entries) {
3228                 return IXGBE_ERR_INVALID_MAC_ADDR;
3229         }

3231         /*
3232          * If we found rar[0], make sure the default pool bit (we use pool 0)
3233          * remains cleared to be sure default pool packets will get delivered
3234          */
3235         if (rar == 0)
3236                 (void) ixgbe_clear_vmdq(hw, rar, 0);
3234                 ixgbe_clear_vmdq(hw, rar, 0);

3238         return rar;
3239 }
______unchanged_portion_omitted_

3999 /**
4000  *  ixgbe_set_fw_drv_ver_generic - Sends driver version to firmware
4001  *  @hw: pointer to the HW structure
4002  *  @maj: driver version major number
4003  *  @min: driver version minor number
4004  *  @build: driver version build number
4005  *  @sub: driver version sub build number
4006  *
4007  *  Sends driver version number to firmware through the manageability
4008  *  block.  On success return IXGBE_SUCCESS
4009  *  else returns IXGBE_ERR_SWFW_SYNC when encountering an error acquiring
4010  *  semaphore or IXGBE_ERR_HOST_INTERFACE_COMMAND when command fails.
4011  **/
4012 s32 ixgbe_set_fw_drv_ver_generic(struct ixgbe_hw *hw, u8 maj, u8 min,
4013                                  u8 build, u8 sub)
4014 {
4015         struct ixgbe_hic_drv_info fw_cmd;
4016         int i;
4017         s32 ret_val = IXGBE_SUCCESS;

4019         DEBUGFUNC("ixgbe_set_fw_drv_ver_generic");

4021         if (hw->mac.ops.acquire_swfw_sync(hw, IXGBE_GSSR_SW_MNG_SM)
4022             != IXGBE_SUCCESS) {
4023                 ret_val = IXGBE_ERR_SWFW_SYNC;
4024                 goto out;
4025         }

4027         fw_cmd.hdr.cmd = FW_CEM_CMD_DRIVER_INFO;
4028         fw_cmd.hdr.buf_len = FW_CEM_CMD_DRIVER_INFO_LEN;
4029         fw_cmd.hdr.cmd_or_resp.cmd_resv = FW_CEM_CMD_RESERVED;
4030         fw_cmd.port_num = (u8)hw->bus.func;
4031         fw_cmd.ver_maj = maj;
4032         fw_cmd.ver_min = min;
4033         fw_cmd.ver_build = build;
4034         fw_cmd.ver_sub = sub;
4035         fw_cmd.hdr.checksum = 0;
4036         fw_cmd.hdr.checksum = ixgbe_calculate_checksum((u8 *)&fw_cmd,
```

```
4037                                (FW_CEM_HDR_LEN + fw_cmd.hdr.buf_len));
4038         fw_cmd.pad = 0;
4039         fw_cmd.pad2 = 0;

4041         for (i = 0; i <= FW_CEM_MAX_RETRIES; i++) {
4042                 /* LINTED */
4043                 ret_val = ixgbe_host_interface_command(hw, (u32 *)&fw_cmd,
4044                                                        sizeof(fw_cmd));
4045                 if (ret_val != IXGBE_SUCCESS)
4046                         continue;

4048                 if (fw_cmd.hdr.cmd_or_resp.ret_status ==
4049                     FW_CEM_RESP_STATUS_SUCCESS)
4050                         ret_val = IXGBE_SUCCESS;
4051                 else
4052                         ret_val = IXGBE_ERR_HOST_INTERFACE_COMMAND;

4054                 break;
4055         }

4057         hw->mac.ops.release_swfw_sync(hw, IXGBE_GSSR_SW_MNG_SM);
4058 out:
4059         return ret_val;
4060 }

4062 /**
4063  * ixgbe_set_rxpba_generic - Initialize Rx packet buffer
4064  * @hw: pointer to hardware structure
4065  * @num_pb: number of packet buffers to allocate
4066  * @headroom: reserve n KB of headroom
4067  * @strategy: packet buffer allocation strategy
4068  **/
4069 void ixgbe_set_rxpba_generic(struct ixgbe_hw *hw, int num_pb, u32 headroom,
4070                              int strategy)
4071 {
4072         u32 pbsize = hw->mac.rx_pb_size;
4073         int i = 0;
4074         u32 rxpktsize, txpktsize, txpbthresh;

4076         /* Reserve headroom */
4077         pbsize -= headroom;

4079         if (!num_pb)
4080                 num_pb = 1;

4082         /* Divide remaining packet buffer space amongst the number of packet
4083          * buffers requested using supplied strategy.
4084          */
4085         switch (strategy) {
4086         case PBA_STRATEGY_WEIGHTED:
4087                 /* ixgbe_dcb_pba_80_48 strategy weight first half of packet
4088                  * buffer with 5/8 of the packet buffer space.
4089                  */
4090                 rxpktsize = (pbsize * 5) / (num_pb * 4);
4091                 pbsize -= rxpktsize * (num_pb / 2);
4092                 rxpktsize <<= IXGBE_RXPBSIZE_SHIFT;
4093                 for (; i < (num_pb / 2); i++)
4094                         IXGBE_WRITE_REG(hw, IXGBE_RXPBSIZE(i), rxpktsize);
4095                 /* Fall through to configure remaining packet buffers */
4096                 /* FALLTHRU */
4097         case PBA_STRATEGY_EQUAL:
4098                 rxpktsize = (pbsize / (num_pb - i)) << IXGBE_RXPBSIZE_SHIFT;
4099                 for (; i < num_pb; i++)
4100                         IXGBE_WRITE_REG(hw, IXGBE_RXPBSIZE(i), rxpktsize);
4101                 break;
4102         default:
```



```
4103                 break;
4104         }

4106         /* Only support an equally distributed Tx packet buffer strategy. */
4107         txpktsize = IXGBE_TXPBSIZE_MAX / num_pb;
4108         txpbthresh = (txpktsize / 1024) - IXGBE_TXPKT_SIZE_MAX;
4109         for (i = 0; i < num_pb; i++) {
4110                 IXGBE_WRITE_REG(hw, IXGBE_TXPBSIZE(i), txpktsize);
4111                 IXGBE_WRITE_REG(hw, IXGBE_TXPBTHRESH(i), txpbthresh);
4112         }

4114         /* Clear unused TCs, if any, to zero buffer size*/
4115         for (; i < IXGBE_MAX_PB; i++) {
4116                 IXGBE_WRITE_REG(hw, IXGBE_RXPBSIZE(i), 0);
4117                 IXGBE_WRITE_REG(hw, IXGBE_TXPBSIZE(i), 0);
4118                 IXGBE_WRITE_REG(hw, IXGBE_TXPBTHRESH(i), 0);
4119         }
4120 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    6965 Fri Jul 27 22:36:11 2012
new/usr/src/uts/common/io/ixgbe/ixgbe_common.h
3014 Intel X540 Support (fix lint)
**********************************************************
```

```
/******************************************************************************

  Copyright (c) 2001-2012, Intel Corporation
  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,
      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.

   3. Neither the name of the Intel Corporation nor the names of its
      contributors may be used to endorse or promote products derived from
      this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS 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 COPYRIGHT OWNER 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.

******************************************************************************/
/*$FreeBSD: src/sys/dev/ixgbe/ixgbe_common.h,v 1.12 2012/07/05 20:51:44 jfv Exp

#ifndef _IXGBE_COMMON_H_
#define _IXGBE_COMMON_H_

#include "ixgbe_type.h"
#if lint
/* Use "hw" somehow... */
#define IXGBE_WRITE_REG64(hw, reg, value) hw = hw
#else
#define IXGBE_WRITE_REG64(hw, reg, value) \
        do { \
                IXGBE_WRITE_REG(hw, reg, (u32) value); \
                IXGBE_WRITE_REG(hw, reg + 4, (u32) (value >> 32)); \
        } while (0)
#endif

u16 ixgbe_get_pcie_msix_count_generic(struct ixgbe_hw *hw);

s32 ixgbe_init_ops_generic(struct ixgbe_hw *hw);
s32 ixgbe_init_hw_generic(struct ixgbe_hw *hw);
s32 ixgbe_start_hw_generic(struct ixgbe_hw *hw);
s32 ixgbe_start_hw_gen2(struct ixgbe_hw *hw);
s32 ixgbe_clear_hw_cntrs_generic(struct ixgbe_hw *hw);
s32 ixgbe_read_pba_num_generic(struct ixgbe_hw *hw, u32 *pba_num);
s32 ixgbe_read_pba_string_generic(struct ixgbe_hw *hw, u8 *pba_num,
                                  u32 pba_num_size);
s32 ixgbe_get_mac_addr_generic(struct ixgbe_hw *hw, u8 *mac_addr);
s32 ixgbe_get_bus_info_generic(struct ixgbe_hw *hw);
```



```
void ixgbe_set_lan_id_multi_port_pcie(struct ixgbe_hw *hw);
s32 ixgbe_stop_adapter_generic(struct ixgbe_hw *hw);

s32 ixgbe_led_on_generic(struct ixgbe_hw *hw, u32 index);
s32 ixgbe_led_off_generic(struct ixgbe_hw *hw, u32 index);

s32 ixgbe_init_eeprom_params_generic(struct ixgbe_hw *hw);
s32 ixgbe_write_eeprom_generic(struct ixgbe_hw *hw, u16 offset, u16 data);
s32 ixgbe_write_eeprom_buffer_bit_bang_generic(struct ixgbe_hw *hw, u16 offset,
                                               u16 words, u16 *data);
s32 ixgbe_read_eerd_generic(struct ixgbe_hw *hw, u16 offset, u16 *data);
s32 ixgbe_read_eerd_buffer_generic(struct ixgbe_hw *hw, u16 offset,
                                   u16 words, u16 *data);
s32 ixgbe_write_eewr_generic(struct ixgbe_hw *hw, u16 offset, u16 data);
s32 ixgbe_write_eewr_buffer_generic(struct ixgbe_hw *hw, u16 offset,
                                    u16 words, u16 *data);
s32 ixgbe_read_eeprom_bit_bang_generic(struct ixgbe_hw *hw, u16 offset,
                                       u16 *data);
s32 ixgbe_read_eeprom_buffer_bit_bang_generic(struct ixgbe_hw *hw, u16 offset,
                                              u16 words, u16 *data);
u16 ixgbe_calc_eeprom_checksum_generic(struct ixgbe_hw *hw);
s32 ixgbe_validate_eeprom_checksum_generic(struct ixgbe_hw *hw,
                                           u16 *checksum_val);
s32 ixgbe_update_eeprom_checksum_generic(struct ixgbe_hw *hw);
s32 ixgbe_poll_eerd_eewr_done(struct ixgbe_hw *hw, u32 ee_reg);

s32 ixgbe_set_rar_generic(struct ixgbe_hw *hw, u32 index, u8 *addr, u32 vmdq,
                          u32 enable_addr);
s32 ixgbe_clear_rar_generic(struct ixgbe_hw *hw, u32 index);
s32 ixgbe_init_rx_addrs_generic(struct ixgbe_hw *hw);
s32 ixgbe_update_mc_addr_list_generic(struct ixgbe_hw *hw, u8 *mc_addr_list,
                                      u32 mc_addr_count,
                                      ixgbe_mc_addr_itr func, bool clear);
s32 ixgbe_update_uc_addr_list_generic(struct ixgbe_hw *hw, u8 *addr_list,
                                      u32 addr_count, ixgbe_mc_addr_itr func);
s32 ixgbe_enable_mc_generic(struct ixgbe_hw *hw);
s32 ixgbe_disable_mc_generic(struct ixgbe_hw *hw);
s32 ixgbe_enable_rx_dma_generic(struct ixgbe_hw *hw, u32 regval);
s32 ixgbe_disable_sec_rx_path_generic(struct ixgbe_hw *hw);
s32 ixgbe_enable_sec_rx_path_generic(struct ixgbe_hw *hw);

s32 ixgbe_fc_enable_generic(struct ixgbe_hw *hw);
void ixgbe_fc_autoneg(struct ixgbe_hw *hw);

s32 ixgbe_validate_mac_addr(u8 *mac_addr);
s32 ixgbe_acquire_swfw_sync(struct ixgbe_hw *hw, u16 mask);
void ixgbe_release_swfw_sync(struct ixgbe_hw *hw, u16 mask);
s32 ixgbe_disable_pcie_master(struct ixgbe_hw *hw);

s32 ixgbe_blink_led_start_generic(struct ixgbe_hw *hw, u32 index);
s32 ixgbe_blink_led_stop_generic(struct ixgbe_hw *hw, u32 index);

s32 ixgbe_get_san_mac_addr_generic(struct ixgbe_hw *hw, u8 *san_mac_addr);
s32 ixgbe_set_san_mac_addr_generic(struct ixgbe_hw *hw, u8 *san_mac_addr);

s32 ixgbe_set_vmdq_generic(struct ixgbe_hw *hw, u32 rar, u32 vmdq);
s32 ixgbe_set_vmdq_san_mac_generic(struct ixgbe_hw *hw, u32 vmdq);
s32 ixgbe_clear_vmdq_generic(struct ixgbe_hw *hw, u32 rar, u32 vmdq);
s32 ixgbe_insert_mac_addr_generic(struct ixgbe_hw *hw, u8 *addr, u32 vmdq);
s32 ixgbe_init_uta_tables_generic(struct ixgbe_hw *hw);
s32 ixgbe_set_vfta_generic(struct ixgbe_hw *hw, u32 vlan,
                           u32 vind, bool vlan_on);
s32 ixgbe_set_vlvf_generic(struct ixgbe_hw *hw, u32 vlan, u32 vind,
                           bool vlan_on, bool *vfta_changed);
s32 ixgbe_clear_vfta_generic(struct ixgbe_hw *hw);
s32 ixgbe_find_vlvf_slot(struct ixgbe_hw *hw, u32 vlan);
```

```
129 s32 ixgbe_check_mac_link_generic(struct ixgbe_hw *hw,
130                                 ixgbe_link_speed *speed,
131                                 bool *link_up, bool link_up_wait_to_complete);

133 s32 ixgbe_get_wwn_prefix_generic(struct ixgbe_hw *hw, u16 *wwnn_prefix,
134                                   u16 *wwpn_prefix);

136 s32 ixgbe_get_fcoe_boot_status_generic(struct ixgbe_hw *hw, u16 *bs);
137 void ixgbe_set_mac_anti_spoofing(struct ixgbe_hw *hw, bool enable, int pf);
138 void ixgbe_set_vlan_anti_spoofing(struct ixgbe_hw *hw, bool enable, int vf);
139 s32 ixgbe_get_device_caps_generic(struct ixgbe_hw *hw, u16 *device_caps);
140 void ixgbe_set_rxpba_generic(struct ixgbe_hw *hw, int num_pb, u32 headroom,
141                              int strategy);
142 void ixgbe_enable_relaxed_ordering_gen2(struct ixgbe_hw *hw);
143 s32 ixgbe_set_fw_drv_ver_generic(struct ixgbe_hw *hw, u8 maj, u8 min,
144                                  u8 build, u8 ver);
145 void ixgbe_clear_tx_pending(struct ixgbe_hw *hw);
146 #endif /* IXGBE_COMMON */
```

```
**********************************************************
   19895 Fri Jul 27 22:36:15 2012
new/usr/src/uts/common/io/ixgbe/ixgbe_mbx.c
3014 Intel X540 Support (fix lint)
**********************************************************
______unchanged_portion_omitted_

 407 /**
 408  *  ixgbe_write_mbx_vf - Write a message to the mailbox
 409  *  @hw: pointer to the HW structure
 410  *  @msg: The message buffer
 411  *  @size: Length of buffer
 412  *  @mbx_id: id of mailbox to write
 413  *
 414  *  returns SUCCESS if it successfully copied message into the buffer
 415  **/
 416 static s32 ixgbe_write_mbx_vf(struct ixgbe_hw *hw, u32 *msg, u16 size,
 417                               u16 mbx_id)
 418 {
 419         s32 ret_val;
 420         u16 i;

 422         UNREFERENCED_1PARAMETER(mbx_id);

 424         DEBUGFUNC("ixgbe_write_mbx_vf");

 426         /* lock the mailbox to prevent pf/vf race condition */
 427         ret_val = ixgbe_obtain_mbx_lock_vf(hw);
 428         if (ret_val)
 429                 goto out_no_write;

 431         /* flush msg and acks as we are overwriting the message buffer */
 432         ret_val = ixgbe_check_for_msg_vf(hw, 0);
 433         if (ret_val)
 434                 goto out_no_write;
 435         ret_val = ixgbe_check_for_ack_vf(hw, 0);
 436         if (ret_val)
 437                 goto out_no_write;
 432         ixgbe_check_for_msg_vf(hw, 0);
 433         ixgbe_check_for_ack_vf(hw, 0);

 439         /* copy the caller specified message to the mailbox memory buffer */
 440         for (i = 0; i < size; i++)
 441                 IXGBE_WRITE_REG_ARRAY(hw, IXGBE_VFMBMEM, i, msg[i]);

 443         /* update stats */
 444         hw->mbx.stats.msgs_tx++;

 446         /* Drop VFU and interrupt the PF to tell it a message has been sent */
 447         IXGBE_WRITE_REG(hw, IXGBE_VFMAILBOX, IXGBE_VFMAILBOX_REQ);

 449 out_no_write:
 450         return ret_val;
 451 }
______unchanged_portion_omitted_

 644 /**
 645  *  ixgbe_write_mbx_pf - Places a message in the mailbox
 646  *  @hw: pointer to the HW structure
 647  *  @msg: The message buffer
 648  *  @size: Length of buffer
 649  *  @vf_number: the VF index
 650  *
 651  *  returns SUCCESS if it successfully copied message into the buffer
 652  **/
 653 static s32 ixgbe_write_mbx_pf(struct ixgbe_hw *hw, u32 *msg, u16 size,
```



```
 654                               u16 vf_number)
 655 {
 656         s32 ret_val;
 657         u16 i;

 659         DEBUGFUNC("ixgbe_write_mbx_pf");

 661         /* lock the mailbox to prevent pf/vf race condition */
 662         ret_val = ixgbe_obtain_mbx_lock_pf(hw, vf_number);
 663         if (ret_val)
 664                 goto out_no_write;

 666         /* flush msg and acks as we are overwriting the message buffer */
 667         ret_val = ixgbe_check_for_msg_vf(hw, 0);
 668         if (ret_val)
 669                 goto out_no_write;
 670         ret_val = ixgbe_check_for_ack_vf(hw, 0);
 671         if (ret_val)
 672                 goto out_no_write;
 663         ixgbe_check_for_msg_pf(hw, vf_number);
 664         ixgbe_check_for_ack_pf(hw, vf_number);

 674         /* copy the caller specified message to the mailbox memory buffer */
 675         for (i = 0; i < size; i++)
 676                 IXGBE_WRITE_REG_ARRAY(hw, IXGBE_PFMBMEM(vf_number), i, msg[i]);

 678         /* Interrupt VF to tell it a message has been sent and release buffer*/
 679         IXGBE_WRITE_REG(hw, IXGBE_PFMAILBOX(vf_number), IXGBE_PFMAILBOX_STS);

 681         /* update stats */
 682         hw->mbx.stats.msgs_tx++;

 684 out_no_write:
 685         return ret_val;

 687 }
______unchanged_portion_omitted_
```

```
**********************************************************
    4005 Fri Jul 27 22:36:17 2012
new/usr/src/uts/common/io/ixgbe/ixgbe_osdep.h
3014 Intel X540 Support (fix lint)
**********************************************************
```

```
/*
 * 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 (c) 2012, Joyent, Inc. All rights reserved.
 */

#ifndef _IXGBE_OSDEP_H
#define _IXGBE_OSDEP_H

#ifdef __cplusplus
extern "C" {
#endif

#include <sys/types.h>
#include <sys/byteorder.h>
#include <sys/conf.h>
#include <sys/debug.h>
#include <sys/stropts.h>
#include <sys/stream.h>
#include <sys/strlog.h>
#include <sys/kmem.h>
#include <sys/stat.h>
#include <sys/kstat.h>
#include <sys/modctl.h>
#include <sys/errno.h>
#include <sys/ddi.h>
#include <sys/dditypes.h>
#include <sys/sunddi.h>
#include <sys/pci.h>
#include <sys/atomic.h>
#include <sys/note.h>
#include "ixgbe_debug.h"

/* Cheesy hack for EWARN() */
#define EWARN(H, W, S) cmn_err(CE_NOTE, W)
```



```
/* function declarations */
struct ixgbe_hw;
uint16_t ixgbe_read_pci_cfg(struct ixgbe_hw *, uint32_t);
void ixgbe_write_pci_cfg(struct ixgbe_hw *, uint32_t, uint32_t);

#define usec_delay(x)           drv_usecwait(x)
#define msec_delay(x)           drv_usecwait(x * 1000)

#define OS_DEP(hw)              ((struct ixgbe_osdep *)((hw)->back))

#define false           B_FALSE
#define true            B_TRUE
#define FALSE           B_FALSE
#define TRUE            B_TRUE

#define IXGBE_READ_PCIE_WORD    ixgbe_read_pci_cfg
#define IXGBE_WRITE_PCIE_WORD   ixgbe_write_pci_cfg
#define CMD_MEM_WRT_INVALIDATE  0x0010  /* BIT_4 */
#define PCI_COMMAND_REGISTER    0x04
#define PCI_EX_CONF_CAP         0xE0
#define SPEED_10GB              10000
#define SPEED_1GB               1000
#define SPEED_100               100
#define FULL_DUPLEX             2

#define IXGBE_WRITE_FLUSH(a)    (void) IXGBE_READ_REG(a, IXGBE_STATUS)

#define IXGBE_WRITE_REG(a, reg, value)  \
        ddi_put32((OS_DEP(a))->reg_handle, \
            (uint32_t *)((uintptr_t)(a)->hw_addr + reg), (value))

#define IXGBE_WRITE_REG_ARRAY(a, reg, index, value)     \
        IXGBE_WRITE_REG(a, ((reg) + ((index) << 2)), (value))

#define IXGBE_READ_REG(a, reg)  \
        ddi_get32((OS_DEP(a))->reg_handle, \
            (uint32_t *)((uintptr_t)(a)->hw_addr + reg))

#define IXGBE_READ_REG_ARRAY(a, reg, index)     \
        IXGBE_READ_REG(a, ((reg) + ((index) << 2)))

#define msec_delay_irq  msec_delay
#define IXGBE_HTONL     htonl
#define IXGBE_NTOHL     ntohl
#define IXGBE_NTOHS     ntohs

#ifdef _BIG_ENDIAN
#define IXGBE_CPU_TO_LE32       BSWAP_32
#define IXGBE_LE32_TO_CPUS      BSWAP_32
#else
#define IXGBE_CPU_TO_LE32(x)    (x)
#if lint
/* Use lint-happy operation... */
#define IXGBE_LE32_TO_CPUS(x)
#else
#define IXGBE_LE32_TO_CPUS(x)   (x)
#endif /* lint */
#endif  /* _BIG_ENDIAN */

#define UNREFERENCED_PARAMETER(x)               _NOTE(ARGUNUSED(x))
#define UNREFERENCED_1PARAMETER(_p)             UNREFERENCED_PARAMETER(_p)
#define UNREFERENCED_2PARAMETER(_p, _q)         _NOTE(ARGUNUSED(_p, _q))
#define UNREFERENCED_3PARAMETER(_p, _q, _r)     _NOTE(ARGUNUSED(_p, _q, _r))
#define UNREFERENCED_4PARAMETER(_p, _q, _r, _s) _NOTE(ARGUNUSED(_p, _q,_r, _s))
```

```
130 typedef int8_t          s8;
131 typedef int16_t         s16;
132 typedef int32_t         s32;
133 typedef int64_t         s64;
134 typedef uint8_t         u8;
135 typedef uint16_t        u16;
136 typedef uint32_t        u32;
137 typedef uint64_t        u64;
138 typedef boolean_t       bool;

140 struct ixgbe_osdep {
141         ddi_acc_handle_t reg_handle;
142         ddi_acc_handle_t cfg_handle;
143         struct ixgbe *ixgbe;
144 };
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   51472 Fri Jul 27 22:36:19 2012
new/usr/src/uts/common/io/ixgbe/ixgbe_phy.c
3014 Intel X540 Support (fix lint)
**********************************************************
______unchanged_portion_omitted_

 83 /**
 84  *  ixgbe_identify_phy_generic - Get physical layer module
 85  *  @hw: pointer to hardware structure
 86  *
 87  *  Determines the physical layer module found on the current adapter.
 88  **/
 89 s32 ixgbe_identify_phy_generic(struct ixgbe_hw *hw)
 90 {
 91         s32 status = IXGBE_ERR_PHY_ADDR_INVALID;
 92         u32 phy_addr;
 93         u16 ext_ability = 0;

 95         DEBUGFUNC("ixgbe_identify_phy_generic");

 97         if (hw->phy.type == ixgbe_phy_unknown) {
 98                 for (phy_addr = 0; phy_addr < IXGBE_MAX_PHY_ADDR; phy_addr++) {
 99                         if (ixgbe_validate_phy_addr(hw, phy_addr)) {
100                                 hw->phy.addr = phy_addr;
101                                 (void) ixgbe_get_phy_id(hw);
101                                 ixgbe_get_phy_id(hw);
102                                 hw->phy.type =
103                                         ixgbe_get_phy_type_from_id(hw->phy.id);

105                                 if (hw->phy.type == ixgbe_phy_unknown) {
106                                         hw->phy.ops.read_reg(hw,
107                                                   IXGBE_MDIO_PHY_EXT_ABILITY,
108                                                   IXGBE_MDIO_PMA_PMD_DEV_TYPE,
109                                                   &ext_ability);
110                                         if (ext_ability &
111                                             (IXGBE_MDIO_PHY_10GBASET_ABILITY |
112                                              IXGBE_MDIO_PHY_1000BASET_ABILITY))
113                                                 hw->phy.type =
114                                                         ixgbe_phy_cu_unknown;
115                                         else
116                                                 hw->phy.type =
117                                                         ixgbe_phy_generic;
118                                 }

120                                 status = IXGBE_SUCCESS;
121                                 break;
122                         }
123                 }
124                 /* clear value if nothing found */
125                 if (status != IXGBE_SUCCESS)
126                         hw->phy.addr = 0;
127         } else {
128                 status = IXGBE_SUCCESS;
129         }

131         return status;
132 }
______unchanged_portion_omitted_

465 /**
466  *  ixgbe_setup_phy_link_generic - Set and restart autoneg
467  *  @hw: pointer to hardware structure
468  *
469  *  Restart autonegotiation and PHY and waits for completion.
470  **/
```



```
471 s32 ixgbe_setup_phy_link_generic(struct ixgbe_hw *hw)
472 {
473         s32 status;
473         s32 status = IXGBE_SUCCESS;
474         u32 time_out;
475         u32 max_time_out = 10;
476         u16 autoneg_reg = IXGBE_MII_AUTONEG_REG;
477         bool autoneg = FALSE;
478         ixgbe_link_speed speed;

480         DEBUGFUNC("ixgbe_setup_phy_link_generic");

482         status =
483             ixgbe_get_copper_link_capabilities_generic(hw, &speed, &autoneg);
484         if (status != IXGBE_SUCCESS)
485                 return status;

487         if (speed & IXGBE_LINK_SPEED_10GB_FULL) {
488                 /* Set or unset auto-negotiation 10G advertisement */
489                 hw->phy.ops.read_reg(hw, IXGBE_MII_10GBASE_T_AUTONEG_CTRL_REG,
490                                      IXGBE_MDIO_AUTO_NEG_DEV_TYPE,
491                                      &autoneg_reg);

493                 autoneg_reg &= ~IXGBE_MII_10GBASE_T_ADVERTISE;
494                 if (hw->phy.autoneg_advertised & IXGBE_LINK_SPEED_10GB_FULL)
495                         autoneg_reg |= IXGBE_MII_10GBASE_T_ADVERTISE;

497                 hw->phy.ops.write_reg(hw, IXGBE_MII_10GBASE_T_AUTONEG_CTRL_REG,
498                                       IXGBE_MDIO_AUTO_NEG_DEV_TYPE,
499                                       autoneg_reg);
500         }

502         if (speed & IXGBE_LINK_SPEED_1GB_FULL) {
503                 /* Set or unset auto-negotiation 1G advertisement */
504                 hw->phy.ops.read_reg(hw,
505                                      IXGBE_MII_AUTONEG_VENDOR_PROVISION_1_REG,
506                                      IXGBE_MDIO_AUTO_NEG_DEV_TYPE,
507                                      &autoneg_reg);

509                 autoneg_reg &= ~IXGBE_MII_1GBASE_T_ADVERTISE;
510                 if (hw->phy.autoneg_advertised & IXGBE_LINK_SPEED_1GB_FULL)
511                         autoneg_reg |= IXGBE_MII_1GBASE_T_ADVERTISE;

513                 hw->phy.ops.write_reg(hw,
514                                       IXGBE_MII_AUTONEG_VENDOR_PROVISION_1_REG,
515                                       IXGBE_MDIO_AUTO_NEG_DEV_TYPE,
516                                       autoneg_reg);
517         }

519         if (speed & IXGBE_LINK_SPEED_100_FULL) {
520                 /* Set or unset auto-negotiation 100M advertisement */
521                 hw->phy.ops.read_reg(hw, IXGBE_MII_AUTONEG_ADVERTISE_REG,
522                                      IXGBE_MDIO_AUTO_NEG_DEV_TYPE,
523                                      &autoneg_reg);

525                 autoneg_reg &= ~(IXGBE_MII_100BASE_T_ADVERTISE |
526                                  IXGBE_MII_100BASE_T_ADVERTISE_HALF);
527                 if (hw->phy.autoneg_advertised & IXGBE_LINK_SPEED_100_FULL)
528                         autoneg_reg |= IXGBE_MII_100BASE_T_ADVERTISE;

530                 hw->phy.ops.write_reg(hw, IXGBE_MII_AUTONEG_ADVERTISE_REG,
531                                       IXGBE_MDIO_AUTO_NEG_DEV_TYPE,
532                                       autoneg_reg);
533         }

535         /* Restart PHY autonegotiation and wait for completion */
```

```
536         hw->phy.ops.read_reg(hw, IXGBE_MDIO_AUTO_NEG_CONTROL,
537                              IXGBE_MDIO_AUTO_NEG_DEV_TYPE, &autoneg_reg);

539         autoneg_reg |= IXGBE_MII_RESTART;

541         hw->phy.ops.write_reg(hw, IXGBE_MDIO_AUTO_NEG_CONTROL,
542                               IXGBE_MDIO_AUTO_NEG_DEV_TYPE, autoneg_reg);

544         /* Wait for autonegotiation to finish */
545         for (time_out = 0; time_out < max_time_out; time_out++) {
546                 usec_delay(10);
547                 /* Restart PHY autonegotiation and wait for completion */
548                 status = hw->phy.ops.read_reg(hw, IXGBE_MDIO_AUTO_NEG_STATUS,
549                                               IXGBE_MDIO_AUTO_NEG_DEV_TYPE,
550                                               &autoneg_reg);

552                 autoneg_reg &= IXGBE_MII_AUTONEG_COMPLETE;
553                 if (autoneg_reg == IXGBE_MII_AUTONEG_COMPLETE)
554                         break;
555         }

557         if (time_out == max_time_out) {
558                 status = IXGBE_ERR_LINK_SETUP;
559                 DEBUGOUT("ixgbe_setup_phy_link_generic: time out");
560         }

562         return status;
563 }
_____unchanged_portion_omitted_

686 /**
687  *      ixgbe_setup_phy_link_tnx - Set and restart autoneg
688  *      @hw: pointer to hardware structure
689  *
690  *      Restart autonegotiation and PHY and waits for completion.
691  **/
692 s32 ixgbe_setup_phy_link_tnx(struct ixgbe_hw *hw)
693 {
694         s32 status;
691         s32 status = IXGBE_SUCCESS;
695         u32 time_out;
696         u32 max_time_out = 10;
697         u16 autoneg_reg = IXGBE_MII_AUTONEG_REG;
698         bool autoneg = FALSE;
699         ixgbe_link_speed speed;

701         DEBUGFUNC("ixgbe_setup_phy_link_tnx");

703         status =
704             ixgbe_get_copper_link_capabilities_generic(hw, &speed, &autoneg);
705         if (status != IXGBE_SUCCESS)
706                 return status;

708         if (speed & IXGBE_LINK_SPEED_10GB_FULL) {
709                 /* Set or unset auto-negotiation 10G advertisement */
710                 hw->phy.ops.read_reg(hw, IXGBE_MII_10GBASE_T_AUTONEG_CTRL_REG,
711                                      IXGBE_MDIO_AUTO_NEG_DEV_TYPE,
712                                      &autoneg_reg);

714                 autoneg_reg &= ~IXGBE_MII_10GBASE_T_ADVERTISE;
715                 if (hw->phy.autoneg_advertised & IXGBE_LINK_SPEED_10GB_FULL)
716                         autoneg_reg |= IXGBE_MII_10GBASE_T_ADVERTISE;

718                 hw->phy.ops.write_reg(hw, IXGBE_MII_10GBASE_T_AUTONEG_CTRL_REG,
719                                       IXGBE_MDIO_AUTO_NEG_DEV_TYPE,
720                                       autoneg_reg);
```



```
721         }

723         if (speed & IXGBE_LINK_SPEED_1GB_FULL) {
724                 /* Set or unset auto-negotiation 1G advertisement */
725                 hw->phy.ops.read_reg(hw, IXGBE_MII_AUTONEG_XNP_TX_REG,
726                                      IXGBE_MDIO_AUTO_NEG_DEV_TYPE,
727                                      &autoneg_reg);

729                 autoneg_reg &= ~IXGBE_MII_1GBASE_T_ADVERTISE_XNP_TX;
730                 if (hw->phy.autoneg_advertised & IXGBE_LINK_SPEED_1GB_FULL)
731                         autoneg_reg |= IXGBE_MII_1GBASE_T_ADVERTISE_XNP_TX;

733                 hw->phy.ops.write_reg(hw, IXGBE_MII_AUTONEG_XNP_TX_REG,
734                                       IXGBE_MDIO_AUTO_NEG_DEV_TYPE,
735                                       autoneg_reg);
736         }

738         if (speed & IXGBE_LINK_SPEED_100_FULL) {
739                 /* Set or unset auto-negotiation 100M advertisement */
740                 hw->phy.ops.read_reg(hw, IXGBE_MII_AUTONEG_ADVERTISE_REG,
741                                      IXGBE_MDIO_AUTO_NEG_DEV_TYPE,
742                                      &autoneg_reg);

744                 autoneg_reg &= ~IXGBE_MII_100BASE_T_ADVERTISE;
745                 if (hw->phy.autoneg_advertised & IXGBE_LINK_SPEED_100_FULL)
746                         autoneg_reg |= IXGBE_MII_100BASE_T_ADVERTISE;

748                 hw->phy.ops.write_reg(hw, IXGBE_MII_AUTONEG_ADVERTISE_REG,
749                                       IXGBE_MDIO_AUTO_NEG_DEV_TYPE,
750                                       autoneg_reg);
751         }

753         /* Restart PHY autonegotiation and wait for completion */
754         hw->phy.ops.read_reg(hw, IXGBE_MDIO_AUTO_NEG_CONTROL,
755                              IXGBE_MDIO_AUTO_NEG_DEV_TYPE, &autoneg_reg);

757         autoneg_reg |= IXGBE_MII_RESTART;

759         hw->phy.ops.write_reg(hw, IXGBE_MDIO_AUTO_NEG_CONTROL,
760                               IXGBE_MDIO_AUTO_NEG_DEV_TYPE, autoneg_reg);

762         /* Wait for autonegotiation to finish */
763         for (time_out = 0; time_out < max_time_out; time_out++) {
764                 usec_delay(10);
765                 /* Restart PHY autonegotiation and wait for completion */
766                 status = hw->phy.ops.read_reg(hw, IXGBE_MDIO_AUTO_NEG_STATUS,
767                                               IXGBE_MDIO_AUTO_NEG_DEV_TYPE,
768                                               &autoneg_reg);

770                 autoneg_reg &= IXGBE_MII_AUTONEG_COMPLETE;
771                 if (autoneg_reg == IXGBE_MII_AUTONEG_COMPLETE)
772                         break;
773         }

775         if (time_out == max_time_out) {
776                 status = IXGBE_ERR_LINK_SETUP;
777                 DEBUGOUT("ixgbe_setup_phy_link_tnx: time out");
778         }

780         return status;
781 }
_____unchanged_portion_omitted_

947 /**
948  *  ixgbe_identify_sfp_module_generic - Identifies SFP modules
949  *  @hw: pointer to hardware structure
```

```
 950  *
 951  *  Searches for and identifies the SFP module and assigns appropriate PHY type.
 952  **/
 953 s32 ixgbe_identify_sfp_module_generic(struct ixgbe_hw *hw)
 954 {
 955         s32 status = IXGBE_ERR_PHY_ADDR_INVALID;
 956         u32 vendor_oui = 0;
 957         enum ixgbe_sfp_type stored_sfp_type = hw->phy.sfp_type;
 958         u8 identifier = 0;
 959         u8 comp_codes_1g = 0;
 960         u8 comp_codes_10g = 0;
 961         u8 oui_bytes[3] = {0, 0, 0};
 962         u8 cable_tech = 0;
 963         u8 cable_spec = 0;
 964         u16 enforce_sfp = 0;

 966         DEBUGFUNC("ixgbe_identify_sfp_module_generic");

 968         if (hw->mac.ops.get_media_type(hw) != ixgbe_media_type_fiber) {
 969                 hw->phy.sfp_type = ixgbe_sfp_type_not_present;
 970                 status = IXGBE_ERR_SFP_NOT_PRESENT;
 971                 goto out;
 972         }

 974         status = hw->phy.ops.read_i2c_eeprom(hw,
 975                                              IXGBE_SFF_IDENTIFIER,
 976                                              &identifier);

 978         if (status == IXGBE_ERR_SWFW_SYNC ||
 979             status == IXGBE_ERR_I2C ||
 980             status == IXGBE_ERR_SFP_NOT_PRESENT)
 981                 goto err_read_i2c_eeprom;

 983         /* LAN ID is needed for sfp_type determination */
 984         hw->mac.ops.set_lan_id(hw);

 986         if (identifier != IXGBE_SFF_IDENTIFIER_SFP) {
 987                 hw->phy.type = ixgbe_phy_sfp_unsupported;
 988                 status = IXGBE_ERR_SFP_NOT_SUPPORTED;
 989         } else {
 990                 status = hw->phy.ops.read_i2c_eeprom(hw,
 991                                                      IXGBE_SFF_1GBE_COMP_CODES,
 992                                                      &comp_codes_1g);

 994                 if (status == IXGBE_ERR_SWFW_SYNC ||
 995                     status == IXGBE_ERR_I2C ||
 996                     status == IXGBE_ERR_SFP_NOT_PRESENT)
 997                         goto err_read_i2c_eeprom;

 999                 status = hw->phy.ops.read_i2c_eeprom(hw,
1000                                                      IXGBE_SFF_10GBE_COMP_CODES,
1001                                                      &comp_codes_10g);

1003                 if (status == IXGBE_ERR_SWFW_SYNC ||
1004                     status == IXGBE_ERR_I2C ||
1005                     status == IXGBE_ERR_SFP_NOT_PRESENT)
1006                         goto err_read_i2c_eeprom;
1007                 status = hw->phy.ops.read_i2c_eeprom(hw,
1008                                                      IXGBE_SFF_CABLE_TECHNOLOGY,
1009                                                      &cable_tech);

1011                 if (status == IXGBE_ERR_SWFW_SYNC ||
1012                     status == IXGBE_ERR_I2C ||
1013                     status == IXGBE_ERR_SFP_NOT_PRESENT)
1014                         goto err_read_i2c_eeprom;
```



```
1016                 /* ID Module
1017                  * =========
1018                  * 0   SFP_DA_CU
1019                  * 1   SFP_SR
1020                  * 2   SFP_LR
1021                  * 3   SFP_DA_CORE0 - 82599-specific
1022                  * 4   SFP_DA_CORE1 - 82599-specific
1023                  * 5   SFP_SR/LR_CORE0 - 82599-specific
1024                  * 6   SFP_SR/LR_CORE1 - 82599-specific
1025                  * 7   SFP_act_lmt_DA_CORE0 - 82599-specific
1026                  * 8   SFP_act_lmt_DA_CORE1 - 82599-specific
1027                  * 9   SFP_1g_cu_CORE0 - 82599-specific
1028                  * 10  SFP_1g_cu_CORE1 - 82599-specific
1029                  * 11  SFP_1g_sx_CORE0 - 82599-specific
1030                  * 12  SFP_1g_sx_CORE1 - 82599-specific
1031                  */
1032                 if (hw->mac.type == ixgbe_mac_82598EB) {
1033                         if (cable_tech & IXGBE_SFF_DA_PASSIVE_CABLE)
1034                                 hw->phy.sfp_type = ixgbe_sfp_type_da_cu;
1035                         else if (comp_codes_10g & IXGBE_SFF_10GBASESR_CAPABLE)
1036                                 hw->phy.sfp_type = ixgbe_sfp_type_sr;
1037                         else if (comp_codes_10g & IXGBE_SFF_10GBASELR_CAPABLE)
1038                                 hw->phy.sfp_type = ixgbe_sfp_type_lr;
1039                         else
1040                                 hw->phy.sfp_type = ixgbe_sfp_type_unknown;
1041                 } else if (hw->mac.type == ixgbe_mac_82599EB) {
1042                         if (cable_tech & IXGBE_SFF_DA_PASSIVE_CABLE) {
1043                                 if (hw->bus.lan_id == 0)
1044                                         hw->phy.sfp_type =
1045                                                 ixgbe_sfp_type_da_cu_core0;
1046                                 else
1047                                         hw->phy.sfp_type =
1048                                                 ixgbe_sfp_type_da_cu_core1;
1049                         } else if (cable_tech & IXGBE_SFF_DA_ACTIVE_CABLE) {
1050                                 hw->phy.ops.read_i2c_eeprom(
1051                                                 hw, IXGBE_SFF_CABLE_SPEC_COMP,
1052                                                 &cable_spec);
1053                                 if (cable_spec &
1054                                     IXGBE_SFF_DA_SPEC_ACTIVE_LIMITING) {
1055                                         if (hw->bus.lan_id == 0)
1056                                                 hw->phy.sfp_type =
1057                                                 ixgbe_sfp_type_da_act_lmt_core0;
1058                                         else
1059                                                 hw->phy.sfp_type =
1060                                                 ixgbe_sfp_type_da_act_lmt_core1;
1061                                 } else {
1062                                         hw->phy.sfp_type =
1063                                                     ixgbe_sfp_type_unknown;
1064                                 }
1065                         } else if (comp_codes_10g &
1066                                    (IXGBE_SFF_10GBASESR_CAPABLE |
1067                                     IXGBE_SFF_10GBASELR_CAPABLE)) {
1068                                 if (hw->bus.lan_id == 0)
1069                                         hw->phy.sfp_type =
1070                                                     ixgbe_sfp_type_srlr_core0;
1071                                 else
1072                                         hw->phy.sfp_type =
1073                                                     ixgbe_sfp_type_srlr_core1;
1074                         } else if (comp_codes_1g & IXGBE_SFF_1GBASET_CAPABLE) {
1075                                 if (hw->bus.lan_id == 0)
1076                                         hw->phy.sfp_type =
1077                                                 ixgbe_sfp_type_1g_cu_core0;
1078                                 else
1079                                         hw->phy.sfp_type =
1080                                                 ixgbe_sfp_type_1g_cu_core1;
1081                         } else if (comp_codes_1g & IXGBE_SFF_1GBASESX_CAPABLE) {
```

```
1082                                if (hw->bus.lan_id == 0)
1083                                        hw->phy.sfp_type =
1084                                                ixgbe_sfp_type_1g_sx_core0;
1085                                else
1086                                        hw->phy.sfp_type =
1087                                                ixgbe_sfp_type_1g_sx_core1;
1088                        } else {
1089                                hw->phy.sfp_type = ixgbe_sfp_type_unknown;
1090                        }
1091                }

1093                if (hw->phy.sfp_type != stored_sfp_type)
1094                        hw->phy.sfp_setup_needed = TRUE;

1096                /* Determine if the SFP+ PHY is dual speed or not. */
1097                hw->phy.multispeed_fiber = FALSE;
1098                if (((comp_codes_1g & IXGBE_SFF_1GBASESX_CAPABLE) &&
1099                   (comp_codes_10g & IXGBE_SFF_10GBASESR_CAPABLE)) ||
1100                   ((comp_codes_1g & IXGBE_SFF_1GBASELX_CAPABLE) &&
1101                   (comp_codes_10g & IXGBE_SFF_10GBASELR_CAPABLE)))
1102                        hw->phy.multispeed_fiber = TRUE;

1104                /* Determine PHY vendor */
1105                if (hw->phy.type != ixgbe_phy_nl) {
1106                        hw->phy.id = identifier;
1107                        status = hw->phy.ops.read_i2c_eeprom(hw,
1108                                                IXGBE_SFF_VENDOR_OUI_BYTE0,
1109                                                &oui_bytes[0]);

1111                        if (status == IXGBE_ERR_SWFW_SYNC ||
1112                            status == IXGBE_ERR_I2C ||
1113                            status == IXGBE_ERR_SFP_NOT_PRESENT)
1114                                goto err_read_i2c_eeprom;

1116                        status = hw->phy.ops.read_i2c_eeprom(hw,
1117                                                IXGBE_SFF_VENDOR_OUI_BYTE1,
1118                                                &oui_bytes[1]);

1120                        if (status == IXGBE_ERR_SWFW_SYNC ||
1121                            status == IXGBE_ERR_I2C ||
1122                            status == IXGBE_ERR_SFP_NOT_PRESENT)
1123                                goto err_read_i2c_eeprom;

1125                        status = hw->phy.ops.read_i2c_eeprom(hw,
1126                                                IXGBE_SFF_VENDOR_OUI_BYTE2,
1127                                                &oui_bytes[2]);

1129                        if (status == IXGBE_ERR_SWFW_SYNC ||
1130                            status == IXGBE_ERR_I2C ||
1131                            status == IXGBE_ERR_SFP_NOT_PRESENT)
1132                                goto err_read_i2c_eeprom;

1134                        vendor_oui =
1135                          ((oui_bytes[0] << IXGBE_SFF_VENDOR_OUI_BYTE0_SHIFT) |
1136                           (oui_bytes[1] << IXGBE_SFF_VENDOR_OUI_BYTE1_SHIFT) |
1137                           (oui_bytes[2] << IXGBE_SFF_VENDOR_OUI_BYTE2_SHIFT));

1139                        switch (vendor_oui) {
1140                        case IXGBE_SFF_VENDOR_OUI_TYCO:
1141                                if (cable_tech & IXGBE_SFF_DA_PASSIVE_CABLE)
1142                                        hw->phy.type =
1143                                                ixgbe_phy_sfp_passive_tyco;
1144                                break;
1145                        case IXGBE_SFF_VENDOR_OUI_FTL:
1146                                if (cable_tech & IXGBE_SFF_DA_ACTIVE_CABLE)
1147                                        hw->phy.type = ixgbe_phy_sfp_ftl_active;
```



```
1148                                else
1149                                        hw->phy.type = ixgbe_phy_sfp_ftl;
1150                                break;
1151                        case IXGBE_SFF_VENDOR_OUI_AVAGO:
1152                                hw->phy.type = ixgbe_phy_sfp_avago;
1153                                break;
1154                        case IXGBE_SFF_VENDOR_OUI_INTEL:
1155                                hw->phy.type = ixgbe_phy_sfp_intel;
1156                                break;
1157                        default:
1158                                if (cable_tech & IXGBE_SFF_DA_PASSIVE_CABLE)
1159                                        hw->phy.type =
1160                                                ixgbe_phy_sfp_passive_unknown;
1161                                else if (cable_tech & IXGBE_SFF_DA_ACTIVE_CABLE)
1162                                        hw->phy.type =
1163                                                ixgbe_phy_sfp_active_unknown;
1164                                else
1165                                        hw->phy.type = ixgbe_phy_sfp_unknown;
1166                                break;
1167                        }
1168                }

1170                /* Allow any DA cable vendor */
1171                if (cable_tech & (IXGBE_SFF_DA_PASSIVE_CABLE |
1172                    IXGBE_SFF_DA_ACTIVE_CABLE)) {
1173                        status = IXGBE_SUCCESS;
1174                        goto out;
1175                }

1177                /* Verify supported 1G SFP modules */
1178                if (comp_codes_10g == 0 &&
1179                    !(hw->phy.sfp_type == ixgbe_sfp_type_1g_cu_core1 ||
1180                      hw->phy.sfp_type == ixgbe_sfp_type_1g_cu_core0 ||
1181                      hw->phy.sfp_type == ixgbe_sfp_type_1g_sx_core0  ||
1182                      hw->phy.sfp_type == ixgbe_sfp_type_1g_sx_core1)) {
1183                        hw->phy.type = ixgbe_phy_sfp_unsupported;
1184                        status = IXGBE_ERR_SFP_NOT_SUPPORTED;
1185                        goto out;
1186                }

1188                /* Anything else 82598-based is supported */
1189                if (hw->mac.type == ixgbe_mac_82598EB) {
1190                        status = IXGBE_SUCCESS;
1191                        goto out;
1192                }

1194                (void) ixgbe_get_device_caps(hw, &enforce_sfp);
1188                ixgbe_get_device_caps(hw, &enforce_sfp);
1195                if (!(enforce_sfp & IXGBE_DEVICE_CAPS_ALLOW_ANY_SFP) &&
1196                    !((hw->phy.sfp_type == ixgbe_sfp_type_1g_cu_core0) ||
1197                      (hw->phy.sfp_type == ixgbe_sfp_type_1g_cu_core1) ||
1198                      (hw->phy.sfp_type == ixgbe_sfp_type_1g_sx_core0)  ||
1199                      (hw->phy.sfp_type == ixgbe_sfp_type_1g_sx_core1))) {
1200                        /* Make sure we're a supported PHY type */
1201                        if (hw->phy.type == ixgbe_phy_sfp_intel) {
1202                                status = IXGBE_SUCCESS;
1203                        } else {
1204                                if (hw->allow_unsupported_sfp == TRUE) {
1205                                        EWARN(hw, "WARNING: Intel (R) Network "
1206                                              "Connections are quality tested "
1207                                              "using Intel (R) Ethernet Optics."
1208                                              " Using untested modules is not "
1209                                              "supported and may cause unstable"
1210                                              " operation or damage to the "
1211                                              "module or the adapter. Intel "
1212                                              "Corporation is not responsible "
```

```
1213                                            "for any harm caused by using "
1214                                            "untested modules.\n", status);
1215                                    status = IXGBE_SUCCESS;
1216                            } else {
1217                                    DEBUGOUT("SFP+ module not supported\n");
1218                                    hw->phy.type =
1219                                            ixgbe_phy_sfp_unsupported;
1220                                    status = IXGBE_ERR_SFP_NOT_SUPPORTED;
1221                            }
1222                    }
1223                } else {
1224                        status = IXGBE_SUCCESS;
1225                }
1226        }

1228 out:
1229        return status;

1231 err_read_i2c_eeprom:
1232        hw->phy.sfp_type = ixgbe_sfp_type_not_present;
1233        if (hw->phy.type != ixgbe_phy_nl) {
1234                hw->phy.id = 0;
1235                hw->phy.type = ixgbe_phy_unknown;
1236        }
1237        return IXGBE_ERR_SFP_NOT_PRESENT;
1238 }
```

*unchanged_portion_omitted_*

```
1525 /**
1526  *  ixgbe_i2c_start - Sets I2C start condition
1527  *  @hw: pointer to hardware structure
1528  *
1529  *  Sets I2C start condition (High -> Low on SDA while SCL is High)
1530  **/
1531 static void ixgbe_i2c_start(struct ixgbe_hw *hw)
1532 {
1533        u32 i2cctl = IXGBE_READ_REG(hw, IXGBE_I2CCTL);

1535        DEBUGFUNC("ixgbe_i2c_start");

1537        /* Start condition must begin with data and clock high */
1538        (void) ixgbe_set_i2c_data(hw, &i2cctl, 1);
1532        ixgbe_set_i2c_data(hw, &i2cctl, 1);
1539        ixgbe_raise_i2c_clk(hw, &i2cctl);

1541        /* Setup time for start condition (4.7us) */
1542        usec_delay(IXGBE_I2C_T_SU_STA);

1544        (void) ixgbe_set_i2c_data(hw, &i2cctl, 0);
1538        ixgbe_set_i2c_data(hw, &i2cctl, 0);

1546        /* Hold time for start condition (4us) */
1547        usec_delay(IXGBE_I2C_T_HD_STA);

1549        ixgbe_lower_i2c_clk(hw, &i2cctl);

1551        /* Minimum low period of clock is 4.7 us */
1552        usec_delay(IXGBE_I2C_T_LOW);

1554 }

1556 /**
1557  *  ixgbe_i2c_stop - Sets I2C stop condition
1558  *  @hw: pointer to hardware structure
1559  *
1560  *  Sets I2C stop condition (Low -> High on SDA while SCL is High)
```



```
1561  **/
1562 static void ixgbe_i2c_stop(struct ixgbe_hw *hw)
1563 {
1564        u32 i2cctl = IXGBE_READ_REG(hw, IXGBE_I2CCTL);

1566        DEBUGFUNC("ixgbe_i2c_stop");

1568        /* Stop condition must begin with data low and clock high */
1569        (void) ixgbe_set_i2c_data(hw, &i2cctl, 0);
1563        ixgbe_set_i2c_data(hw, &i2cctl, 0);
1570        ixgbe_raise_i2c_clk(hw, &i2cctl);

1572        /* Setup time for stop condition (4us) */
1573        usec_delay(IXGBE_I2C_T_SU_STO);

1575        (void) ixgbe_set_i2c_data(hw, &i2cctl, 1);
1569        ixgbe_set_i2c_data(hw, &i2cctl, 1);

1577        /* bus free time between stop and start (4.7us)*/
1578        usec_delay(IXGBE_I2C_T_BUF);
1579 }

1581 /**
1582  *  ixgbe_clock_in_i2c_byte - Clocks in one byte via I2C
1583  *  @hw: pointer to hardware structure
1584  *  @data: data byte to clock in
1585  *
1586  *  Clocks in one byte data via I2C data/clock
1587  **/
1588 static s32 ixgbe_clock_in_i2c_byte(struct ixgbe_hw *hw, u8 *data)
1589 {
1590        s32 i, status = IXGBE_SUCCESS;
1584        s32 i;
1591        bool bit = 0;

1593        DEBUGFUNC("ixgbe_clock_in_i2c_byte");

1595        for (i = 7; i >= 0; i--) {
1596                status = ixgbe_clock_in_i2c_bit(hw, &bit);
1597                if (status != IXGBE_SUCCESS)
1598                        break;
1590                ixgbe_clock_in_i2c_bit(hw, &bit);
1599                *data |= bit << i;
1600        }

1602        return status;
1594        return IXGBE_SUCCESS;
1603 }
```

*unchanged_portion_omitted_*

```
1853 /**
1854  *  ixgbe_i2c_bus_clear - Clears the I2C bus
1855  *  @hw: pointer to hardware structure
1856  *
1857  *  Clears the I2C bus by sending nine clock pulses.
1858  *  Used when data line is stuck low.
1859  **/
1860 void ixgbe_i2c_bus_clear(struct ixgbe_hw *hw)
1861 {
1862        u32 i2cctl = IXGBE_READ_REG(hw, IXGBE_I2CCTL);
1863        u32 i;

1865        DEBUGFUNC("ixgbe_i2c_bus_clear");

1867        ixgbe_i2c_start(hw);
```

```
1869          (void) ixgbe_set_i2c_data(hw, &i2cctl, 1);
1861          ixgbe_set_i2c_data(hw, &i2cctl, 1);

1871          for (i = 0; i < 9; i++) {
1872                  ixgbe_raise_i2c_clk(hw, &i2cctl);

1874                  /* Min high period of clock is 4us */
1875                  usec_delay(IXGBE_I2C_T_HIGH);

1877                  ixgbe_lower_i2c_clk(hw, &i2cctl);

1879                  /* Min low period of clock is 4.7us*/
1880                  usec_delay(IXGBE_I2C_T_LOW);
1881          }

1883          ixgbe_i2c_start(hw);

1885          /* Put the i2c bus back to default state */
1886          ixgbe_i2c_stop(hw);
1887  }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   27954 Fri Jul 27 22:36:21 2012
new/usr/src/uts/common/io/ixgbe/ixgbe_x540.c
3014 Intel X540 Support (fix lint)
**********************************************************
______unchanged_portion_omitted_

 147 /**
 148  *  ixgbe_get_link_capabilities_X540 - Determines link capabilities
 149  *  @hw: pointer to hardware structure
 150  *  @speed: pointer to link speed
 151  *  @autoneg: TRUE when autoneg or autotry is enabled
 152  *
 153  *  Determines the link capabilities by reading the AUTOC register.
 154  **/
 155 s32 ixgbe_get_link_capabilities_X540(struct ixgbe_hw *hw,
 156                                      ixgbe_link_speed *speed,
 157                                      bool *autoneg)
 158 {
 159         return ixgbe_get_copper_link_capabilities_generic(hw, speed, autoneg);
 159         ixgbe_get_copper_link_capabilities_generic(hw, speed, autoneg);

 161         return IXGBE_SUCCESS;
 160 }
______unchanged_portion_omitted_

 536 /**
 537  *  ixgbe_validate_eeprom_checksum_X540 - Validate EEPROM checksum
 538  *  @hw: pointer to hardware structure
 539  *  @checksum_val: calculated checksum
 540  *
 541  *  Performs checksum calculation and validates the EEPROM checksum.  If the
 542  *  caller does not need checksum_val, the value can be NULL.
 543  **/
 544 s32 ixgbe_validate_eeprom_checksum_X540(struct ixgbe_hw *hw,
 545                                         u16 *checksum_val)
 546 {
 547         s32 status;
 548         u16 checksum;
 549         u16 read_checksum = 0;

 551         DEBUGFUNC("ixgbe_validate_eeprom_checksum_X540");

 553         /*
 554          * Read the first word from the EEPROM. If this times out or fails, do
 555          * not continue or we could be in for a very long wait while every
 556          * EEPROM read fails
 557          */
 558         status = hw->eeprom.ops.read(hw, 0, &checksum);

 560         if (status != IXGBE_SUCCESS) {
 561                 DEBUGOUT("EEPROM read failed\n");
 562                 goto out;
 563         }

 565         if (hw->mac.ops.acquire_swfw_sync(hw, IXGBE_GSSR_EEP_SM) ==
 566             IXGBE_SUCCESS) {
 567                 checksum = hw->eeprom.ops.calc_checksum(hw);

 569                 /*
 570                  * Do not use hw->eeprom.ops.read because we do not want to take
 571                  * the synchronization semaphores twice here.
 572                  */
 573                 status = ixgbe_read_eerd_generic(hw, IXGBE_EEPROM_CHECKSUM,
 575                 ixgbe_read_eerd_generic(hw, IXGBE_EEPROM_CHECKSUM,
 574                                         &read_checksum);
```



```
 576                 if (status == IXGBE_SUCCESS) {
 577                         /*
 578                          * Verify read checksum from EEPROM is the same as
 579                          * calculated checksum
 580                          */
 581                         if (read_checksum != checksum)
 582                                 status = IXGBE_ERR_EEPROM_CHECKSUM;

 584                         /* If the user cares, return the calculated checksum */
 585                         if (checksum_val)
 586                                 *checksum_val = checksum;
 587                 }
 588         } else {
 589                 status = IXGBE_ERR_SWFW_SYNC;
 590         }

 592         hw->mac.ops.release_swfw_sync(hw, IXGBE_GSSR_EEP_SM);
 593 out:
 594         return status;
 595 }
______unchanged_portion_omitted_

 800 /**
 801  *  ixgbe_release_swfw_sync_X540 - Release SWFW semaphore
 802  *  @hw: pointer to hardware structure
 803  *  @mask: Mask to specify which semaphore to release
 804  *
 805  *  Releases the SWFW semaphore through the SW_FW_SYNC register
 806  *  for the specified function (CSR, PHY0, PHY1, EVM, Flash)
 807  **/
 808 void ixgbe_release_swfw_sync_X540(struct ixgbe_hw *hw, u16 mask)
 809 {
 810         u32 swfw_sync;
 811         u32 swmask = mask;

 813         DEBUGFUNC("ixgbe_release_swfw_sync_X540");

 815         (void) ixgbe_get_swfw_sync_semaphore(hw);
 815         ixgbe_get_swfw_sync_semaphore(hw);

 817         swfw_sync = IXGBE_READ_REG(hw, IXGBE_SWFW_SYNC);
 818         swfw_sync &= ~swmask;
 819         IXGBE_WRITE_REG(hw, IXGBE_SWFW_SYNC, swfw_sync);

 821         ixgbe_release_swfw_sync_semaphore(hw);
 822         msec_delay(5);
 823 }
______unchanged_portion_omitted_
```