```
**********************************************************
   44505 Tue Jan  7 18:43:36 2014
new/usr/src/uts/i86pc/io/pcplusmp/apic_common.c
4443 apic_intrmap_init comment could use an update
**********************************************************
______unchanged_portion_omitted_

1497 void
1498 apic_intrmap_init(int apic_mode)
1499 {
1500         int suppress_brdcst_eoi = 0;

1502         if (psm_vt_ops != NULL) {
1502         /*
1503          * Intel Software Developer's Manual 3A, 10.12.7:
1504          *
1505          * Routing of device interrupts to local APIC units operating in
1506          * x2APIC mode requires use of the interrupt-remapping architecture
1507          * specified in the Intel Virtualization Technology for Directed
1508          * I/O, Revision 1.3.  Because of this, BIOS must enumerate support
1509          * for and software must enable this interrupt remapping with
1510          * Extended Interrupt Mode Enabled before it enabling x2APIC mode in
1511          * the local APIC units.
1512          *
1513          *
1514          * In other words, to use the APIC in x2APIC mode, we need interrupt
1515          * remapping.  Since we don't start up the IOMMU by default, we
1516          * won't be able to do any interrupt remapping and therefore have to
1517          * use the APIC in traditional 'local APIC' mode with memory mapped
1518          * I/O.
1504                  * Since X2APIC requires the use of interrupt remapping
1505                  * (though this is not documented explicitly in the Intel
1506                  * documentation (yet)), initialize interrupt remapping
1507                  * support before initializing the X2APIC unit.
1519          */

1521         if (psm_vt_ops != NULL) {
1522 #endif /* ! codereview */
1523                 if (((apic_intrmap_ops_t *)psm_vt_ops)->
1524                     apic_intrmap_init(apic_mode) == DDI_SUCCESS) {

1526                         apic_vt_ops = psm_vt_ops;

1528                         /*
1529                          * We leverage the interrupt remapping engine to
1530                          * suppress broadcast EOI; thus we must send the
1531                          * directed EOI with the directed-EOI handler.
1532                          */
1533                         if (apic_directed_EOI_supported() == 0) {
1534                                 suppress_brdcst_eoi = 1;
1535                         }

1537                         apic_vt_ops->apic_intrmap_enable(suppress_brdcst_eoi);

1539                         if (apic_detect_x2apic()) {
1540                                 apic_enable_x2apic();
1541                         }

1543                         if (apic_directed_EOI_supported() == 0) {
1544                                 apic_set_directed_EOI_handler();
1545                         }
1546                 }
1547         }
1548 }

1550 /*ARGSUSED*/
```



```
1551 static void
1552 apic_record_ioapic_rdt(void *intrmap_private, ioapic_rdt_t *irdt)
1553 {
1554         irdt->ir_hi <<= APIC_ID_BIT_OFFSET;
1555 }

1557 /*ARGSUSED*/
1558 static void
1559 apic_record_msi(void *intrmap_private, msi_regs_t *mregs)
1560 {
1561         mregs->mr_addr = MSI_ADDR_HDR |
1562             (MSI_ADDR_RH_FIXED << MSI_ADDR_RH_SHIFT) |
1563             (MSI_ADDR_DM_PHYSICAL << MSI_ADDR_DM_SHIFT) |
1564             (mregs->mr_addr << MSI_ADDR_DEST_SHIFT);
1565         mregs->mr_data = (MSI_DATA_TM_EDGE << MSI_DATA_TM_SHIFT) |
1566             mregs->mr_data;
1567 }

1569 /*
1570  * Functions from apic_introp.c
1571  *
1572  * Those functions are used by apic_intr_ops().
1573  */

1575 /*
1576  * MSI support flag:
1577  * reflects whether MSI is supported at APIC level
1578  * it can also be patched through /etc/system
1579  *
1580  *  0 = default value - don't know and need to call apic_check_msi_support()
1581  *      to find out then set it accordingly
1582  *  1 = supported
1583  * -1 = not supported
1584  */
1585 int     apic_support_msi = 0;

1587 /* Multiple vector support for MSI-X */
1588 int     apic_msix_enable = 1;

1590 /* Multiple vector support for MSI */
1591 int     apic_multi_msi_enable = 1;

1593 /*
1594  * check whether the system supports MSI
1595  *
1596  * If PCI-E capability is found, then this must be a PCI-E system.
1597  * Since MSI is required for PCI-E system, it returns PSM_SUCCESS
1598  * to indicate this system supports MSI.
1599  */
1600 int
1601 apic_check_msi_support()
1602 {
1603         dev_info_t *cdip;
1604         char dev_type[16];
1605         int dev_len;

1607         DDI_INTR_IMPLDBG((CE_CONT, "apic_check_msi_support:\n"));

1609         /*
1610          * check whether the first level children of root_node have
1611          * PCI-E capability
1612          */
1613         for (cdip = ddi_get_child(ddi_root_node()); cdip != NULL;
1614             cdip = ddi_get_next_sibling(cdip)) {

1616                 DDI_INTR_IMPLDBG((CE_CONT, "apic_check_msi_support: cdip: 0x%p,"
```

```
1617                    " driver: %s, binding: %s, nodename: %s\n", (void *)cdip,
1618                    ddi_driver_name(cdip), ddi_binding_name(cdip),
1619                    ddi_node_name(cdip)));
1620                dev_len = sizeof (dev_type);
1621                if (ddi_getlongprop_buf(DDI_DEV_T_ANY, cdip, DDI_PROP_DONTPASS,
1622                    "device_type", (caddr_t)dev_type, &dev_len)
1623                    != DDI_PROP_SUCCESS)
1624                        continue;
1625                if (strcmp(dev_type, "pciex") == 0)
1626                        return (PSM_SUCCESS);
1627        }

1629        /* MSI is not supported on this system */
1630        DDI_INTR_IMPLDBG((CE_CONT, "apic_check_msi_support: no 'pciex' "
1631            "device_type found\n"));
1632        return (PSM_FAILURE);
1633 }

1635 /*
1636  * apic_pci_msi_unconfigure:
1637  *
1638  * This and next two interfaces are copied from pci_intr_lib.c
1639  * Do ensure that these two files stay in sync.
1640  * These needed to be copied over here to avoid a deadlock situation on
1641  * certain mp systems that use MSI interrupts.
1642  *
1643  * IMPORTANT regards next three interfaces:
1644  * i) are called only for MSI/X interrupts.
1645  * ii) called with interrupts disabled, and must not block
1646  */
1647 void
1648 apic_pci_msi_unconfigure(dev_info_t *rdip, int type, int inum)
1649 {
1650        ushort_t                msi_ctrl;
1651        int                     cap_ptr = i_ddi_get_msi_msix_cap_ptr(rdip);
1652        ddi_acc_handle_t        handle = i_ddi_get_pci_config_handle(rdip);

1654        ASSERT((handle != NULL) && (cap_ptr != 0));

1656        if (type == DDI_INTR_TYPE_MSI) {
1657                msi_ctrl = pci_config_get16(handle, cap_ptr + PCI_MSI_CTRL);
1658                msi_ctrl &= (~PCI_MSI_MME_MASK);
1659                pci_config_put16(handle, cap_ptr + PCI_MSI_CTRL, msi_ctrl);
1660                pci_config_put32(handle, cap_ptr + PCI_MSI_ADDR_OFFSET, 0);

1662                if (msi_ctrl &  PCI_MSI_64BIT_MASK) {
1663                        pci_config_put16(handle,
1664                            cap_ptr + PCI_MSI_64BIT_DATA, 0);
1665                        pci_config_put32(handle,
1666                            cap_ptr + PCI_MSI_ADDR_OFFSET + 4, 0);
1667                } else {
1668                        pci_config_put16(handle,
1669                            cap_ptr + PCI_MSI_32BIT_DATA, 0);
1670                }

1672        } else if (type == DDI_INTR_TYPE_MSIX) {
1673                uintptr_t       off;
1674                uint32_t        mask;
1675                ddi_intr_msix_t *msix_p = i_ddi_get_msix(rdip);

1677                ASSERT(msix_p != NULL);

1679                /* Offset into "inum"th entry in the MSI-X table & mask it */
1680                off = (uintptr_t)msix_p->msix_tbl_addr + (inum *
1681                    PCI_MSIX_VECTOR_SIZE) + PCI_MSIX_VECTOR_CTRL_OFFSET;
```



```
1683                mask = ddi_get32(msix_p->msix_tbl_hdl, (uint32_t *)off);

1685                ddi_put32(msix_p->msix_tbl_hdl, (uint32_t *)off, (mask | 1));

1687                /* Offset into the "inum"th entry in the MSI-X table */
1688                off = (uintptr_t)msix_p->msix_tbl_addr +
1689                    (inum * PCI_MSIX_VECTOR_SIZE);

1691                /* Reset the "data" and "addr" bits */
1692                ddi_put32(msix_p->msix_tbl_hdl,
1693                    (uint32_t *)(off + PCI_MSIX_DATA_OFFSET), 0);
1694                ddi_put64(msix_p->msix_tbl_hdl, (uint64_t *)off, 0);
1695        }
1696 }

1698 /*
1699  * apic_pci_msi_disable_mode:
1700  */
1701 void
1702 apic_pci_msi_disable_mode(dev_info_t *rdip, int type)
1703 {
1704        ushort_t                msi_ctrl;
1705        int                     cap_ptr = i_ddi_get_msi_msix_cap_ptr(rdip);
1706        ddi_acc_handle_t        handle = i_ddi_get_pci_config_handle(rdip);

1708        ASSERT((handle != NULL) && (cap_ptr != 0));

1710        if (type == DDI_INTR_TYPE_MSI) {
1711                msi_ctrl = pci_config_get16(handle, cap_ptr + PCI_MSI_CTRL);
1712                if (!(msi_ctrl & PCI_MSI_ENABLE_BIT))
1713                        return;

1715                msi_ctrl &= ~PCI_MSI_ENABLE_BIT;        /* MSI disable */
1716                pci_config_put16(handle, cap_ptr + PCI_MSI_CTRL, msi_ctrl);

1718        } else if (type == DDI_INTR_TYPE_MSIX) {
1719                msi_ctrl = pci_config_get16(handle, cap_ptr + PCI_MSIX_CTRL);
1720                if (msi_ctrl & PCI_MSIX_ENABLE_BIT) {
1721                        msi_ctrl &= ~PCI_MSIX_ENABLE_BIT;
1722                        pci_config_put16(handle, cap_ptr + PCI_MSIX_CTRL,
1723                            msi_ctrl);
1724                }
1725        }
1726 }

1728 uint32_t
1729 apic_get_localapicid(uint32_t cpuid)
1730 {
1731        ASSERT(cpuid < apic_nproc && apic_cpus != NULL);

1733        return (apic_cpus[cpuid].aci_local_id);
1734 }

1736 uchar_t
1737 apic_get_ioapicid(uchar_t ioapicindex)
1738 {
1739        ASSERT(ioapicindex < MAX_IO_APIC);

1741        return (apic_io_id[ioapicindex]);
1742 }
```