```
**********************************************************
   41057 Mon May  5 11:11:06 2014
new/usr/src/uts/common/io/iprb/iprb.c
4778 iprb shouldn't abuse ddi_get_time(9f)
Reviewed by: Robert Mustacchi <rm@joyent.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
**********************************************************
______unchanged_portion_omitted_

int
iprb_attach(dev_info_t *dip)
{
        iprb_t          *ip;
        uint16_t        w;
        int             i;
        mac_register_t  *macp;

        ip = kmem_zalloc(sizeof (*ip), KM_SLEEP);
        ddi_set_driver_private(dip, ip);
        ip->dip = dip;

        list_create(&ip->mcast, sizeof (struct iprb_mcast),
            offsetof(struct iprb_mcast, node));

        /* we don't support high level interrupts, so we don't need cookies */
        mutex_init(&ip->culock, NULL, MUTEX_DRIVER, NULL);
        mutex_init(&ip->rulock, NULL, MUTEX_DRIVER, NULL);

        if (pci_config_setup(dip, &ip->pcih) != DDI_SUCCESS) {
                iprb_error(ip, "unable to map configuration space");
                iprb_destroy(ip);
                return (DDI_FAILURE);
        }

        if (ddi_regs_map_setup(dip, 1, &ip->regs, 0, 0, &acc_attr,
            &ip->regsh) != DDI_SUCCESS) {
                iprb_error(ip, "unable to map device registers");
                iprb_destroy(ip);
                return (DDI_FAILURE);
        }

        /* Reset, but first go into idle state */
        PUT32(ip, CSR_PORT, PORT_SEL_RESET);
        drv_usecwait(10);
        PUT32(ip, CSR_PORT, PORT_SW_RESET);
        drv_usecwait(10);
        PUT8(ip, CSR_INTCTL, INTCTL_MASK);
        (void) GET8(ip, CSR_INTCTL);

        /*
         * Precalculate watchdog times.
         */
        ip->tx_timeout = TX_WATCHDOG;
        ip->rx_timeout = RX_WATCHDOG;
        ip->tx_timeout = drv_usectohz(TX_WATCHDOG * 1000000);
        ip->rx_timeout = drv_usectohz(RX_WATCHDOG * 1000000);

        iprb_identify(ip);

        /* Obtain our factory MAC address */
        w = iprb_eeprom_read(ip, 0);
        ip->factaddr[0] = w & 0xff;
        ip->factaddr[1] = w >> 8;
        w = iprb_eeprom_read(ip, 1);
        ip->factaddr[2] = w & 0xff;
        ip->factaddr[3] = w >> 8;
```

```
        w = iprb_eeprom_read(ip, 2);
        ip->factaddr[4] = w & 0xff;
        ip->factaddr[5] = w >> 8;
        bcopy(ip->factaddr, ip->curraddr, 6);

        if (ip->resumebug) {
                /*
                 * Generally, most devices we will ever see will
                 * already have fixed firmware.  Since I can't verify
                 * the validity of the fix (no suitably downrev
                 * hardware), we'll just do our best to avoid it for
                 * devices that exhibit this behavior.
                 */
                if ((iprb_eeprom_read(ip, 10) & 0x02) == 0) {
                        /* EEPROM fix was already applied, assume safe. */
                        ip->resumebug = B_FALSE;
                }
        }

        if ((iprb_eeprom_read(ip, 3) & 0x3) != 0x3) {
                cmn_err(CE_CONT, "?Enabling RX errata workaround.\n");
                ip->rxhangbug = B_TRUE;
        }

        /* Determine whether we have an MII or a legacy 80c24 */
        w = iprb_eeprom_read(ip, 6);
        if ((w & 0x3f00) != 0x0600) {
                if ((ip->miih = mii_alloc(ip, dip, &iprb_mii_ops)) == NULL) {
                        iprb_error(ip, "unable to allocate MII ops vector");
                        iprb_destroy(ip);
                        return (DDI_FAILURE);
                }
                if (ip->canpause) {
                        mii_set_pauseable(ip->miih, B_TRUE, B_FALSE);
                }
        }

        /* Allocate cmds and tx region */
        for (i = 0; i < NUM_TX; i++) {
                /* Command blocks */
                if (iprb_dma_alloc(ip, &ip->cmds[i], CB_SIZE) != DDI_SUCCESS) {
                        iprb_destroy(ip);
                        return (DDI_FAILURE);
                }
        }

        for (i = 0; i < NUM_TX; i++) {
                iprb_dma_t *cb = &ip->cmds[i];
                /* Link the command blocks into a ring */
                PUTCB32(cb, CB_LNK_OFFSET, (ip->cmds[(i + 1) % NUM_TX].paddr));
        }

        for (i = 0; i < NUM_RX; i++) {
                /* Rx packet buffers */
                if (iprb_dma_alloc(ip, &ip->rxb[i], RFD_SIZE) != DDI_SUCCESS) {
                        iprb_destroy(ip);
                        return (DDI_FAILURE);
                }
        }
        if (iprb_dma_alloc(ip, &ip->stats, STATS_SIZE) != DDI_SUCCESS) {
                iprb_destroy(ip);
                return (DDI_FAILURE);
        }

        if (iprb_add_intr(ip) != DDI_SUCCESS) {
                iprb_destroy(ip);
                return (DDI_FAILURE);
        }

        if ((macp = mac_alloc(MAC_VERSION)) == NULL) {
                iprb_error(ip, "unable to allocate mac structure");
                iprb_destroy(ip);
                return (DDI_FAILURE);
        }

        macp->m_type_ident = MAC_PLUGIN_IDENT_ETHER;
        macp->m_driver = ip;
        macp->m_dip = dip;
        macp->m_src_addr = ip->curraddr;
        macp->m_callbacks = &iprb_m_callbacks;
        macp->m_min_sdu = 0;
        macp->m_max_sdu = ETHERMTU;
        macp->m_margin = VLAN_TAGSZ;
        if (mac_register(macp, &ip->mach) != 0) {
                iprb_error(ip, "unable to register mac with framework");
                mac_free(macp);
                iprb_destroy(ip);
                return (DDI_FAILURE);
        }

        mac_free(macp);
        return (DDI_SUCCESS);
}
______unchanged_portion_omitted_

void
iprb_cmd_reclaim(iprb_t *ip)
{
        while (ip->cmd_count) {
                iprb_dma_t *cb = &ip->cmds[ip->cmd_tail];

                SYNCCB(cb, CB_STS_OFFSET, 2, DDI_DMA_SYNC_FORKERNEL);
                if ((GETCB16(cb, CB_STS_OFFSET) & CB_STS_C) == 0) {
                        break;
                }

                ip->cmd_tail++;
                ip->cmd_tail %= NUM_TX;
                ip->cmd_count--;
                if (ip->cmd_count == 0) {
                        ip->tx_wdog = 0;
                } else {
                        ip->tx_wdog = gethrtime();
                        ip->tx_wdog = ddi_get_time();
                }
        }
}
______unchanged_portion_omitted_

int
iprb_cmd_submit(iprb_t *ip, uint16_t cmd)
{
        iprb_dma_t      *ncb = &ip->cmds[ip->cmd_head];
        iprb_dma_t      *lcb = &ip->cmds[ip->cmd_last];

        /* If this command will consume the last CB, interrupt when done */
        ASSERT((ip->cmd_count) < NUM_TX);
        if (ip->cmd_count == (NUM_TX - 1)) {
                cmd |= CB_CMD_I;
        }

        /* clear the status entry */
```

```
        PUTCB16(ncb, CB_STS_OFFSET, 0);

        /* suspend upon completion of this new command */
        cmd |= CB_CMD_S;
        PUTCB16(ncb, CB_CMD_OFFSET, cmd);
        SYNCCB(ncb, 0, 0, DDI_DMA_SYNC_FORDEV);

        /* clear the suspend flag from the last submitted command */
        SYNCCB(lcb, CB_CMD_OFFSET, 2, DDI_DMA_SYNC_FORKERNEL);
        PUTCB16(lcb, CB_CMD_OFFSET, GETCB16(lcb, CB_CMD_OFFSET) & ~CB_CMD_S);
        SYNCCB(lcb, CB_CMD_OFFSET, 2, DDI_DMA_SYNC_FORDEV);


        /*
         * If the chip has a resume bug, then we need to try this as a work
         * around.  Some anecdotal evidence is that this will help solve
         * the resume bug.  Its a performance hit, but only if the EEPROM
         * is not updated.  (In theory we could do this only for 10Mbps HDX,
         * but since it should just about never get used, we keep it simple.)
         */
        if (ip->resumebug) {
                if (iprb_cmd_ready(ip) != DDI_SUCCESS)
                        return (DDI_FAILURE);
                PUT8(ip, CSR_CMD, CUC_NOP);
                (void) GET8(ip, CSR_CMD);
                drv_usecwait(1);
        }

        /* wait for the SCB to be ready to accept a new command */
        if (iprb_cmd_ready(ip) != DDI_SUCCESS)
                return (DDI_FAILURE);

        /*
         * Finally we can resume the CU.  Note that if this the first
         * command in the sequence (i.e. if the CU is IDLE), or if the
         * CU is already busy working, then this CU resume command
         * will not have any effect.
         */
        PUT8(ip, CSR_CMD, CUC_RESUME);
        (void) GET8(ip, CSR_CMD);       /* flush CSR */

        ip->tx_wdog = gethrtime();
        ip->tx_wdog = ddi_get_time();
        ip->cmd_last = ip->cmd_head;
        ip->cmd_head++;
        ip->cmd_head %= NUM_TX;
        ip->cmd_count++;

        return (DDI_SUCCESS);
}
______unchanged_portion_omitted_

void
iprb_update_stats(iprb_t *ip)
{
        iprb_dma_t      *sp = &ip->stats;
        hrtime_t        tstamp;
        time_t          tstamp;
        int             i;

        ASSERT(mutex_owned(&ip->culock));

        /* Collect the hardware stats, but don't keep redoing it */
        tstamp = gethrtime();
        if (tstamp / NANOSEC == ip->stats_time / NANOSEC)
        if ((tstamp = ddi_get_time()) == ip->stats_time) {
                return;
        }

        PUTSTAT(sp, STATS_DONE_OFFSET, 0);
        SYNCSTATS(sp, 0, 0, DDI_DMA_SYNC_FORDEV);

        if (iprb_cmd_ready(ip) != DDI_SUCCESS)
                return;
        PUT32(ip, CSR_GEN_PTR, sp->paddr);
        PUT8(ip, CSR_CMD, CUC_STATSBASE);
        (void) GET8(ip, CSR_CMD);

        if (iprb_cmd_ready(ip) != DDI_SUCCESS)
                return;
        PUT8(ip, CSR_CMD, CUC_STATS_RST);
        (void) GET8(ip, CSR_CMD);       /* flush wb */

        for (i = 10000; i; i -= 10) {
                SYNCSTATS(sp, 0, 0, DDI_DMA_SYNC_FORKERNEL);
                if (GETSTAT(sp, STATS_DONE_OFFSET) == STATS_RST_DONE) {
                        /* yay stats are updated */
                        break;
                }
                drv_usecwait(10);
        }
        if (i == 0) {
                iprb_error(ip, "time out acquiring hardware statistics");
                return;
        }

        ip->ex_coll += GETSTAT(sp, STATS_TX_MAXCOL_OFFSET);
        ip->late_coll += GETSTAT(sp, STATS_TX_LATECOL_OFFSET);
        ip->uflo += GETSTAT(sp, STATS_TX_UFLO_OFFSET);
        ip->defer_xmt += GETSTAT(sp, STATS_TX_DEFER_OFFSET);
        ip->one_coll += GETSTAT(sp, STATS_TX_ONECOL_OFFSET);
        ip->multi_coll += GETSTAT(sp, STATS_TX_MULTCOL_OFFSET);
        ip->collisions += GETSTAT(sp, STATS_TX_TOTCOL_OFFSET);
        ip->fcs_errs += GETSTAT(sp, STATS_RX_FCS_OFFSET);
        ip->align_errs += GETSTAT(sp, STATS_RX_ALIGN_OFFSET);
        ip->norcvbuf += GETSTAT(sp, STATS_RX_NOBUF_OFFSET);
        ip->oflo += GETSTAT(sp, STATS_RX_OFLO_OFFSET);
        ip->runt += GETSTAT(sp, STATS_RX_SHORT_OFFSET);

        ip->stats_time = tstamp;
}
______unchanged_portion_omitted_

mblk_t *
iprb_rx(iprb_t *ip)
{
        iprb_dma_t      *rfd;
        uint16_t        cnt;
        uint16_t        sts;
        int             i;
        mblk_t          *mplist;
        mblk_t          **mpp;
        mblk_t          *mp;

        mplist = NULL;
        mpp = &mplist;

        for (i = 0; i < NUM_RX; i++) {
                rfd = &ip->rxb[ip->rx_index];
                SYNCRFD(rfd, RFD_STS_OFFSET, 2, DDI_DMA_SYNC_FORKERNEL);
                if ((GETRFD16(rfd, RFD_STS_OFFSET) & RFD_STS_C) == 0) {
                        break;
```

```
                }

                ip->rx_wdog = gethrtime();
                ip->rx_wdog = ddi_get_time();

                SYNCRFD(rfd, 0, 0, DDI_DMA_SYNC_FORKERNEL);
                cnt = GETRFD16(rfd, RFD_CNT_OFFSET);
                cnt &= ~(RFD_CNT_EOF | RFD_CNT_F);
                sts = GETRFD16(rfd, RFD_STS_OFFSET);

                if (cnt > (ETHERMAX + VLAN_TAGSZ)) {
                        ip->toolong++;
                        iprb_rx_add(ip);
                        continue;
                }
                if (((sts & RFD_STS_OK) == 0) && (sts & RFD_STS_ERRS)) {
                        iprb_rx_add(ip);
                        continue;
                }
                if ((mp = allocb(cnt, BPRI_MED)) == NULL) {
                        ip->norcvbuf++;
                        iprb_rx_add(ip);
                        continue;
                }
                bcopy(rfd->vaddr + RFD_PKT_OFFSET, mp->b_wptr, cnt);

                /* return it to the RFD list */
                iprb_rx_add(ip);

                mp->b_wptr += cnt;
                ip->ipackets++;
                ip->rbytes += cnt;
                if (mp->b_rptr[0] & 0x1) {
                        if (bcmp(mp->b_rptr, &iprb_bcast, 6) != 0) {
                                ip->multircv++;
                        } else {
                                ip->brdcstrcv++;
                        }
                }
                *mpp = mp;
                mpp = &mp->b_next;
        }
        return (mplist);
}
______unchanged_portion_omitted_

void
iprb_periodic(void *arg)
{
        iprb_t *ip = arg;
        boolean_t reset = B_FALSE;

        mutex_enter(&ip->rulock);
        if (ip->suspended || !ip->running) {
                mutex_exit(&ip->rulock);
                return;
        }

        /*
         * If we haven't received a packet in a while, and if the link
         * is up, then it might be a hung chip.  This problem
         * reportedly only occurs at 10 Mbps.
         */
        if (ip->rxhangbug &&
            ((ip->miih == NULL) || (mii_get_speed(ip->miih) == 10000000)) &&
            ((gethrtime() - ip->rx_wdog) > ip->rx_timeout)) {
```

```
            ((ddi_get_time() - ip->rx_wdog) > ip->rx_timeout)) {
                cmn_err(CE_CONT, "?Possible RU hang, resetting.\n");
                reset = B_TRUE;
        }

        /* update the statistics */
        mutex_enter(&ip->culock);

        if (ip->tx_wdog && ((gethrtime() - ip->tx_wdog) > ip->tx_timeout)) {
        if (ip->tx_wdog && ((ddi_get_time() - ip->tx_wdog) > ip->tx_timeout)) {
                /* transmit/CU hang? */
                cmn_err(CE_CONT, "?CU stalled, resetting.\n");
                reset = B_TRUE;
        }

        if (reset) {
                /* We want to reconfigure */
                iprb_stop(ip);
                if (iprb_start(ip) != DDI_SUCCESS) {
                        iprb_error(ip, "unable to restart chip");
                }
        }

        iprb_update_stats(ip);

        mutex_exit(&ip->culock);
        mutex_exit(&ip->rulock);
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
    9904 Mon May  5 11:11:06 2014
new/usr/src/uts/common/io/iprb/iprb.h
4778 iprb shouldn't abuse ddi_get_time(9f)
Reviewed by: Robert Mustacchi <rm@joyent.com>
Reviewed by: Garrett D'Amore <garrett@damore.org>
**********************************************************
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 * Copyright 2010 Nexenta Systems, Inc.  All rights reserved.
 */

#ifndef _IPRB_H
#define _IPRB_H

/*
 * iprb - Intel Pro/100B Ethernet Driver
 */

/*
 * Tunables.
 */
#define NUM_TX          128     /* outstanding tx queue */
#define NUM_RX          128     /* outstanding rx queue */

/* timeouts for the rx and tx watchdogs (nsec) */
#define RX_WATCHDOG     (15 * NANOSEC)
#define TX_WATCHDOG     (15 * NANOSEC)
#define RX_WATCHDOG     15      /* timeout for rx watchdog (sec) */
#define TX_WATCHDOG     15      /* timeout for tx watchdog (sec) */

/*
 * Driver structures.
 */
typedef struct {
        ddi_acc_handle_t        acch;
        ddi_dma_handle_t        dmah;
        caddr_t                 vaddr;
        uint32_t                paddr;
} iprb_dma_t;
______unchanged_portion_omitted_

typedef struct iprb {
        dev_info_t              *dip;
        ddi_acc_handle_t        pcih;
        ddi_acc_handle_t        regsh;
        caddr_t                 regs;

        uint16_t                devid;
        uint8_t                 revid;

        mac_handle_t            mach;
        mii_handle_t            miih;

        ddi_intr_handle_t       intrh;
```

```
        ddi_periodic_t          perh;

        kmutex_t                culock;
        kmutex_t                rulock;

        uint8_t                 factaddr[6];
        uint8_t                 curraddr[6];

        int                     nmcast;
        list_t                  mcast;
        boolean_t               promisc;
        iprb_dma_t              cmds[NUM_TX];
        iprb_dma_t              rxb[NUM_RX];
        iprb_dma_t              stats;
        hrtime_t                stats_time;
        time_t                  stats_time;

        uint16_t                cmd_head;
        uint16_t                cmd_last;
        uint16_t                cmd_tail;
        uint16_t                cmd_count;

        uint16_t                rx_index;
        uint16_t                rx_last;
        hrtime_t                rx_wdog;
        hrtime_t                rx_timeout;
        hrtime_t                tx_wdog;
        hrtime_t                tx_timeout;
        time_t                  rx_wdog;
        time_t                  rx_timeout;
        time_t                  tx_wdog;
        time_t                  tx_timeout;

        uint16_t                eeprom_bits;

        boolean_t               running;
        boolean_t               suspended;
        boolean_t               wantw;
        boolean_t               rxhangbug;
        boolean_t               resumebug;
        boolean_t               is557;
        boolean_t               canpause;
        boolean_t               canmwi;

        /*
         * Statistics
         */
        uint64_t                ipackets;
        uint64_t                rbytes;
        uint64_t                multircv;
        uint64_t                brdcstrcv;
        uint64_t                opackets;
        uint64_t                obytes;
        uint64_t                multixmt;
        uint64_t                brdcstxmt;
        uint64_t                ex_coll;
        uint64_t                late_coll;
        uint64_t                uflo;
        uint64_t                defer_xmt;
        uint64_t                one_coll;
        uint64_t                multi_coll;
        uint64_t                collisions;
        uint64_t                fcs_errs;
        uint64_t                align_errs;
        uint64_t                norcvbuf;
        uint64_t                oflo;
        uint64_t                runt;
        uint64_t                nocarrier;
        uint64_t                toolong;
        uint64_t                macxmt_errs;
        uint64_t                macrcv_errs;
} iprb_t;
```
______***unchanged_portion_omitted***_