****
25289 Tue Nov 20 20:17:43 2018
new/usr/src/uts/common/io/cxgbe/t4nex/adapter.h
9994 cxgbe t4nex: Handle get_fl_payload() alloc failures
9995 cxgbe t4_devo_attach() should initialize ->sfl
****
______***unchanged_portion_omitted_***

```
#define FL_RUNNING_LOW(fl)      (fl->cap - fl->needed <= fl->lowat)
#define FL_NOT_RUNNING_LOW(fl)  (fl->cap - fl->needed >= 2 * fl->lowat)

struct sge_fl {
        unsigned int flags;
        kmutex_t lock;
        ddi_dma_handle_t dhdl;
        ddi_acc_handle_t ahdl;

        __be64 *desc;           /* KVA of descriptor ring, ptr to addresses */
        uint64_t ba;            /* bus address of descriptor ring */
        struct fl_sdesc *sdesc; /* KVA of software descriptor ring */
        uint32_t cap;           /* max # of buffers, for convenience */
        uint16_t qsize;         /* size (# of entries) of the queue */
        uint16_t cntxt_id;      /* SGE context id for the freelist */
        uint32_t cidx;          /* consumer idx (buffer idx, NOT hw desc idx) */
        uint32_t pidx;          /* producer idx (buffer idx, NOT hw desc idx) */
        uint32_t needed;        /* # of buffers needed to fill up fl. */
        uint32_t lowat;         /* # of buffers <= this means fl needs help */
        uint32_t pending;       /* # of bufs allocated since last doorbell */
        uint32_t offset;        /* current packet within the larger buffer */
        uint16_t copy_threshold; /* anything this size or less is copied up */

        uint64_t copied_up;     /* # of frames copied into mblk and handed up */
        uint64_t passed_up;     /* # of frames wrapped in mblk and handed up */
        uint64_t allocb_fail;   /* # of mblk allocation failures */

        TAILQ_ENTRY(sge_fl) link; /* All starving freelists */
};
```
______***unchanged_portion_omitted_***

```
**********************************************************
   75026 Tue Nov 20 20:17:43 2018
new/usr/src/uts/common/io/cxgbe/t4nex/t4_nexus.c
9994 cxgbe t4nex: Handle get_fl_payload() alloc failures
9995 cxgbe t4_devo_attach() should initialize ->sfl
**********************************************************
______unchanged_portion_omitted_

static int
t4_devo_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        struct adapter *sc = NULL;
        struct sge *s;
        int i, instance, rc = DDI_SUCCESS, rqidx, tqidx, q;
        int irq = 0, nxg, n100g, n40g, n25g, n10g, n1g;
#ifdef TCP_OFFLOAD_ENABLE
        int ofld_rqidx, ofld_tqidx;
#endif
        char name[16];
        struct driver_properties *prp;
        struct intrs_and_queues iaq;
        ddi_device_acc_attr_t da = {
                .devacc_attr_version = DDI_DEVICE_ATTR_V0,
                .devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC,
                .devacc_attr_dataorder = DDI_UNORDERED_OK_ACC
        };
        ddi_device_acc_attr_t da1 = {
                .devacc_attr_version = DDI_DEVICE_ATTR_V0,
                .devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC,
                .devacc_attr_dataorder = DDI_MERGING_OK_ACC
        };

        if (cmd != DDI_ATTACH)
                return (DDI_FAILURE);

        /*
         * Allocate space for soft state.
         */
        instance = ddi_get_instance(dip);
        rc = ddi_soft_state_zalloc(t4_list, instance);
        if (rc != DDI_SUCCESS) {
                cxgb_printf(dip, CE_WARN,
                    "failed to allocate soft state: %d", rc);
                return (DDI_FAILURE);
        }

        sc = ddi_get_soft_state(t4_list, instance);
        sc->dip = dip;
        sc->dev = makedevice(ddi_driver_major(dip), instance);
        mutex_init(&sc->lock, NULL, MUTEX_DRIVER, NULL);
        cv_init(&sc->cv, NULL, CV_DRIVER, NULL);
        mutex_init(&sc->sfl_lock, NULL, MUTEX_DRIVER, NULL);
        TAILQ_INIT(&sc->sfl);

        mutex_enter(&t4_adapter_list_lock);
        SLIST_INSERT_HEAD(&t4_adapter_list, sc, link);
        mutex_exit(&t4_adapter_list_lock);

        sc->pf = getpf(sc);
        if (sc->pf > 8) {
                rc = EINVAL;
                cxgb_printf(dip, CE_WARN,
                    "failed to determine PCI PF# of device");
                goto done;
        }
        sc->mbox = sc->pf;
```

```
        /* Initialize the driver properties */
        prp = &sc->props;
        (void)init_driver_props(sc, prp);

        /*
         * Enable access to the PCI config space.
         */
        rc = pci_config_setup(dip, &sc->pci_regh);
        if (rc != DDI_SUCCESS) {
                cxgb_printf(dip, CE_WARN,
                    "failed to enable PCI config space access: %d", rc);
                goto done;
        }

        /* TODO: Set max read request to 4K */

        /*
         * Enable MMIO access.
         */
        rc = ddi_regs_map_setup(dip, 1, &sc->regp, 0, 0, &da, &sc->regh);
        if (rc != DDI_SUCCESS) {
                cxgb_printf(dip, CE_WARN,
                    "failed to map device registers: %d", rc);
                goto done;
        }

        (void) memset(sc->chan_map, 0xff, sizeof (sc->chan_map));

        /*
         * Initialize cpl handler.
         */
        for (i = 0; i < ARRAY_SIZE(sc->cpl_handler); i++) {
                sc->cpl_handler[i] = cpl_not_handled;
        }

        for (i = 0; i < ARRAY_SIZE(sc->fw_msg_handler); i++) {
                sc->fw_msg_handler[i] = fw_msg_not_handled;
        }

        for (i = 0; i < NCHAN; i++) {
                (void) snprintf(name, sizeof (name), "%s-%d",
                                "reclaim", i);
                sc->tq[i] = ddi_taskq_create(sc->dip,
                   name, 1, TASKQ_DEFAULTPRI, 0);

                if (sc->tq[i] == NULL) {
                        cxgb_printf(dip, CE_WARN,
                                   "failed to create task queues");
                        rc = DDI_FAILURE;
                        goto done;
                }
        }

        /*
         * Prepare the adapter for operation.
         */
        rc = -t4_prep_adapter(sc, false);
        if (rc != 0) {
                cxgb_printf(dip, CE_WARN, "failed to prepare adapter: %d", rc);
                goto done;
        }

        /*
         * Enable BAR1 access.
         */
        sc->doorbells |= DOORBELL_KDB;
        rc = ddi_regs_map_setup(dip, 2, &sc->reg1p, 0, 0, &da1, &sc->reg1h);
        if (rc != DDI_SUCCESS) {
                cxgb_printf(dip, CE_WARN,
                    "failed to map BAR1 device registers: %d", rc);
                goto done;
        } else {
                if (is_t5(sc->params.chip)) {
                        sc->doorbells |= DOORBELL_UDB;
                        if (prp->wc) {
                                /*
                                 * Enable write combining on BAR2.  This is the
                                 * userspace doorbell BAR and is split into 128B
                                 * (UDBS_SEG_SIZE) doorbell regions, each associ
                                 * with an egress queue.  The first 64B has the
                                 * and the second 64B can be used to submit a tx
                                 * request with an implicit doorbell.
                                 */
                                sc->doorbells &= ~DOORBELL_UDB;
                                sc->doorbells |= (DOORBELL_WCWR |
                                    DOORBELL_UDBWC);
                                t4_write_reg(sc, A_SGE_STAT_CFG,
                                    V_STATSOURCE_T5(7) | V_STATMODE(0));
                        }
                }
        }

        /*
         * Do this really early.  Note that minor number = instance.
         */
        (void) snprintf(name, sizeof (name), "%s,%d", T4_NEXUS_NAME, instance);
        rc = ddi_create_minor_node(dip, name, S_IFCHR, instance,
            DDI_NT_NEXUS, 0);
        if (rc != DDI_SUCCESS) {
                cxgb_printf(dip, CE_WARN,
                    "failed to create device node: %d", rc);
                rc = DDI_SUCCESS; /* carry on */
        }

        /* Do this early. Memory window is required for loading config file. */
        setup_memwin(sc);

        /* Prepare the firmware for operation */
        rc = prep_firmware(sc);
        if (rc != 0)
                goto done; /* error message displayed already */

        rc = adap__pre_init_tweaks(sc);
        if (rc != 0)
                goto done;

        rc = get_params__pre_init(sc);
        if (rc != 0)
                goto done; /* error message displayed already */

        t4_sge_init(sc);

        if (sc->flags & MASTER_PF) {
                /* get basic stuff going */
                rc = -t4_fw_initialize(sc, sc->mbox);
                if (rc != 0) {
                        cxgb_printf(sc->dip, CE_WARN,
                            "early init failed: %d.\n", rc);
                        goto done;
                }
        }
```

```
        rc = get_params__post_init(sc);
        if (rc != 0)
                goto done; /* error message displayed already */

        rc = set_params__post_init(sc);
        if (rc != 0)
                goto done; /* error message displayed already */

        /*
         * TODO: This is the place to call t4_set_filter_mode()
         */

        /* tweak some settings */
        t4_write_reg(sc, A_TP_SHIFT_CNT, V_SYNSHIFTMAX(6) | V_RXTSHIFTMAXR1(4) |
            V_RXTSHIFTMAXR2(15) | V_PERSHIFTBACKOFFMAX(8) | V_PERSHIFTMAX(8) |
            V_KEEPALIVEMAXR1(4) | V_KEEPALIVEMAXR2(9));
        t4_write_reg(sc, A_ULP_RX_TDDP_PSZ, V_HPZ0(PAGE_SHIFT - 12));

        /*
         * Work-around for bug 2619
         * Set DisableVlan field in TP_RSS_CONFIG_VRT register so that the
         * VLAN tag extraction is disabled.
         */
        t4_set_reg_field(sc, A_TP_RSS_CONFIG_VRT, F_DISABLEVLAN, F_DISABLEVLAN);

        /* Store filter mode */
        t4_read_indirect(sc, A_TP_PIO_ADDR, A_TP_PIO_DATA, &sc->filter_mode, 1,
            A_TP_VLAN_PRI_MAP);

        /*
         * First pass over all the ports - allocate VIs and initialize some
         * basic parameters like mac address, port type, etc.  We also figure
         * out whether a port is 10G or 1G and use that information when
         * calculating how many interrupts to attempt to allocate.
         */
        n100g = n40g = n25g = n10g = n1g = 0;
        for_each_port(sc, i) {
                struct port_info *pi;

                pi = kmem_zalloc(sizeof (*pi), KM_SLEEP);
                sc->port[i] = pi;

                /* These must be set before t4_port_init */
                pi->adapter = sc;
                /* LINTED: E_ASSIGN_NARROW_CONV */
                pi->port_id = i;
        }

        /* Allocate the vi and initialize parameters like mac addr */
        rc = -t4_port_init(sc, sc->mbox, sc->pf, 0);
        if (rc) {
                cxgb_printf(dip, CE_WARN,
                            "unable to initialize port: %d", rc);
                goto done;
        }

        for_each_port(sc, i) {
                struct port_info *pi = sc->port[i];

                mutex_init(&pi->lock, NULL, MUTEX_DRIVER, NULL);
                pi->mtu = ETHERMTU;

                if (is_100G_port(pi)) {
                        n100g++;
                        pi->tmr_idx = prp->tmr_idx_10g;
                        pi->pktc_idx = prp->pktc_idx_10g;
                } else if (is_40G_port(pi)) {
                        n40g++;
                        pi->tmr_idx = prp->tmr_idx_10g;
                        pi->pktc_idx = prp->pktc_idx_10g;
                } else if (is_25G_port(pi)) {
                        n25g++;
                        pi->tmr_idx = prp->tmr_idx_10g;
                        pi->pktc_idx = prp->pktc_idx_10g;
                } else if (is_10G_port(pi)) {
                        n10g++;
                        pi->tmr_idx = prp->tmr_idx_10g;
                        pi->pktc_idx = prp->pktc_idx_10g;
                } else {
                        n1g++;
                        pi->tmr_idx = prp->tmr_idx_1g;
                        pi->pktc_idx = prp->pktc_idx_1g;
                }

                pi->xact_addr_filt = -1;
                t4_mc_init(pi);

                setbit(&sc->registered_device_map, i);
        }

        nxg = n10g + n25g + n40g + n100g;
        (void) remove_extra_props(sc, nxg, n1g);

        if (sc->registered_device_map == 0) {
                cxgb_printf(dip, CE_WARN, "no usable ports");
                rc = DDI_FAILURE;
                goto done;
        }

        rc = cfg_itype_and_nqueues(sc, nxg, n1g, &iaq);
        if (rc != 0)
                goto done; /* error message displayed already */

        sc->intr_type = iaq.intr_type;
        sc->intr_count = iaq.nirq;

        if (sc->props.multi_rings && (sc->intr_type != DDI_INTR_TYPE_MSIX)) {
                sc->props.multi_rings = 0;
                cxgb_printf(dip, CE_WARN,
                    "Multiple rings disabled as interrupt type is not MSI-X");
        }

        if (sc->props.multi_rings && iaq.intr_fwd) {
                sc->props.multi_rings = 0;
                cxgb_printf(dip, CE_WARN,
                    "Multiple rings disabled as interrupts are forwarded");
        }

        if (!sc->props.multi_rings) {
                iaq.ntxq10g = 1;
                iaq.ntxq1g = 1;
        }
        s = &sc->sge;
        s->nrxq = nxg * iaq.nrxq10g + n1g * iaq.nrxq1g;
        s->ntxq = nxg * iaq.ntxq10g + n1g * iaq.ntxq1g;
        s->neq = s->ntxq + s->nrxq;     /* the fl in an rxq is an eq */
#ifdef TCP_OFFLOAD_ENABLE
        /* control queues, 1 per port + 1 mgmtq */
        s->neq += sc->params.nports + 1;
#endif
        s->niq = s->nrxq + 1;           /* 1 extra for firmware event queue */
```

```
        if (iaq.intr_fwd != 0)
                sc->flags |= INTR_FWD;
#ifdef TCP_OFFLOAD_ENABLE
        if (is_offload(sc) != 0) {

                s->nofldrxq = nxg * iaq.nofldrxq10g + n1g * iaq.nofldrxq1g;
                s->nofldtxq = nxg * iaq.nofldtxq10g + n1g * iaq.nofldtxq1g;
                s->neq += s->nofldtxq + s->nofldrxq;
                s->niq += s->nofldrxq;

                s->ofld_rxq = kmem_zalloc(s->nofldrxq *
                    sizeof (struct sge_ofld_rxq), KM_SLEEP);
                s->ofld_txq = kmem_zalloc(s->nofldtxq *
                    sizeof (struct sge_wrq), KM_SLEEP);
                s->ctrlq = kmem_zalloc(sc->params.nports *
                    sizeof (struct sge_wrq), KM_SLEEP);

        }
#endif
        s->rxq = kmem_zalloc(s->nrxq * sizeof (struct sge_rxq), KM_SLEEP);
        s->txq = kmem_zalloc(s->ntxq * sizeof (struct sge_txq), KM_SLEEP);
        s->iqmap = kmem_zalloc(s->niq * sizeof (struct sge_iq *), KM_SLEEP);
        s->eqmap = kmem_zalloc(s->neq * sizeof (struct sge_eq *), KM_SLEEP);

        sc->intr_handle = kmem_zalloc(sc->intr_count *
            sizeof (ddi_intr_handle_t), KM_SLEEP);

        /*
         * Second pass over the ports.  This time we know the number of rx and
         * tx queues that each port should get.
         */
        rqidx = tqidx = 0;
#ifdef TCP_OFFLOAD_ENABLE
        ofld_rqidx = ofld_tqidx = 0;
#endif
        for_each_port(sc, i) {
                struct port_info *pi = sc->port[i];

                if (pi == NULL)
                        continue;

                t4_mc_cb_init(pi);
                /* LINTED: E_ASSIGN_NARROW_CONV */
                pi->first_rxq = rqidx;
                /* LINTED: E_ASSIGN_NARROW_CONV */
                pi->nrxq = (is_10XG_port(pi)) ? iaq.nrxq10g
                    : iaq.nrxq1g;
                /* LINTED: E_ASSIGN_NARROW_CONV */
                pi->first_txq = tqidx;
                /* LINTED: E_ASSIGN_NARROW_CONV */
                pi->ntxq = (is_10XG_port(pi)) ? iaq.ntxq10g
                    : iaq.ntxq1g;

                rqidx += pi->nrxq;
                tqidx += pi->ntxq;

#ifdef TCP_OFFLOAD_ENABLE
                if (is_offload(sc) != 0) {
                        /* LINTED: E_ASSIGN_NARROW_CONV */
                        pi->first_ofld_rxq = ofld_rqidx;
                        pi->nofldrxq = max(1, pi->nrxq / 4);

                        /* LINTED: E_ASSIGN_NARROW_CONV */
                        pi->first_ofld_txq = ofld_tqidx;
                        pi->nofldtxq = max(1, pi->ntxq / 2);
                        ofld_rqidx += pi->nofldrxq;
                        ofld_tqidx += pi->nofldtxq;
                }
#endif

                /*
                 * Enable hw checksumming and LSO for all ports by default.
                 * They can be disabled using ndd (hw_csum and hw_lso).
                 */
                pi->features |= (CXGBE_HW_CSUM | CXGBE_HW_LSO);
        }

#ifdef TCP_OFFLOAD_ENABLE
                sc->l2t = t4_init_l2t(sc);
#endif

        /*
         * Setup Interrupts.
         */

        i = 0;
        rc = ddi_intr_alloc(dip, sc->intr_handle, sc->intr_type, 0,
            sc->intr_count, &i, DDI_INTR_ALLOC_STRICT);
        if (rc != DDI_SUCCESS) {
                cxgb_printf(dip, CE_WARN,
                    "failed to allocate %d interrupt(s) of type %d: %d, %d",
                    sc->intr_count, sc->intr_type, rc, i);
                goto done;
        }
        ASSERT(sc->intr_count == i); /* allocation was STRICT */
        (void) ddi_intr_get_cap(sc->intr_handle[0], &sc->intr_cap);
        (void) ddi_intr_get_pri(sc->intr_handle[0], &sc->intr_pri);
        if (sc->intr_count == 1) {
                ASSERT(sc->flags & INTR_FWD);
                (void) ddi_intr_add_handler(sc->intr_handle[0], t4_intr_all, sc,
                    &s->fwq);
        } else {
                /* Multiple interrupts.  The first one is always error intr */
                (void) ddi_intr_add_handler(sc->intr_handle[0], t4_intr_err, sc,
                    NULL);
                irq++;

                /* The second one is always the firmware event queue */
                (void) ddi_intr_add_handler(sc->intr_handle[1], t4_intr, sc,
                    &s->fwq);
                irq++;
                /*
                 * Note that if INTR_FWD is set then either the NIC rx
                 * queues or (exclusive or) the TOE rx queueus will be taking
                 * direct interrupts.
                 *
                 * There is no need to check for is_offload(sc) as nofldrxq
                 * will be 0 if offload is disabled.
                 */
                for_each_port(sc, i) {
                        struct port_info *pi = sc->port[i];
                        struct sge_rxq *rxq;
#ifdef TCP_OFFLOAD_ENABLE
                        struct sge_ofld_rxq *ofld_rxq;

                        /*
                         * Skip over the NIC queues if they aren't taking direct
                         * interrupts.
                         */
                        if ((sc->flags & INTR_FWD) &&
                            pi->nofldrxq > pi->nrxq)
```

```
                                goto ofld_queues;
#endif
                        rxq = &s->rxq[pi->first_rxq];
                        for (q = 0; q < pi->nrxq; q++, rxq++) {
                                (void) ddi_intr_add_handler(
                                    sc->intr_handle[irq], t4_intr, sc,
                                    &rxq->iq);
                                irq++;
                        }

#ifdef TCP_OFFLOAD_ENABLE
                        /*
                         * Skip over the offload queues if they aren't taking
                         * direct interrupts.
                         */
                        if ((sc->flags & INTR_FWD))
                                continue;
ofld_queues:
                        ofld_rxq = &s->ofld_rxq[pi->first_ofld_rxq];
                        for (q = 0; q < pi->nofldrxq; q++, ofld_rxq++) {
                                (void) ddi_intr_add_handler(
                                    sc->intr_handle[irq], t4_intr, sc,
                                    &ofld_rxq->iq);
                                irq++;
                        }
#endif
                }

        }
        sc->flags |= INTR_ALLOCATED;

        ASSERT(rc == DDI_SUCCESS);
        ddi_report_dev(dip);

        /*
         * Hardware/Firmware/etc. Version/Revision IDs.
         */
        t4_dump_version_info(sc);

        if (n100g) {
                cxgb_printf(dip, CE_NOTE,
                    "%dx100G (%d rxq, %d txq total) %d %s.",
                    n100g, rqidx, tqidx, sc->intr_count,
                    sc->intr_type == DDI_INTR_TYPE_MSIX ? "MSI-X interrupts" :
                    sc->intr_type == DDI_INTR_TYPE_MSI ? "MSI interrupts" :
                    "fixed interrupt");
        } else if (n40g) {
                cxgb_printf(dip, CE_NOTE,
                    "%dx40G (%d rxq, %d txq total) %d %s.",
                    n40g, rqidx, tqidx, sc->intr_count,
                    sc->intr_type == DDI_INTR_TYPE_MSIX ? "MSI-X interrupts" :
                    sc->intr_type == DDI_INTR_TYPE_MSI ? "MSI interrupts" :
                    "fixed interrupt");
        } else if (n25g) {
                cxgb_printf(dip, CE_NOTE,
                    "%dx25G (%d rxq, %d txq total) %d %s.",
                    n25g, rqidx, tqidx, sc->intr_count,
                    sc->intr_type == DDI_INTR_TYPE_MSIX ? "MSI-X interrupts" :
                    sc->intr_type == DDI_INTR_TYPE_MSI ? "MSI interrupts" :
                    "fixed interrupt");
        } else if (n10g && n1g) {
                cxgb_printf(dip, CE_NOTE,
                    "%dx10G %dx1G (%d rxq, %d txq total) %d %s.",
                    n10g, n1g, rqidx, tqidx, sc->intr_count,
                    sc->intr_type == DDI_INTR_TYPE_MSIX ? "MSI-X interrupts" :
                    sc->intr_type == DDI_INTR_TYPE_MSI ? "MSI interrupts" :
```

```
                    "fixed interrupt");
        } else {
                cxgb_printf(dip, CE_NOTE,
                    "%dx%sG (%d rxq, %d txq per port) %d %s.",
                    n10g ? n10g : n1g,
                    n10g ? "10" : "1",
                    n10g ? iaq.nrxq10g : iaq.nrxq1g,
                    n10g ? iaq.ntxq10g : iaq.ntxq1g,
                    sc->intr_count,
                    sc->intr_type == DDI_INTR_TYPE_MSIX ? "MSI-X interrupts" :
                    sc->intr_type == DDI_INTR_TYPE_MSI ? "MSI interrupts" :
                    "fixed interrupt");
        }

        sc->ksp = setup_kstats(sc);
        sc->ksp_stat = setup_wc_kstats(sc);
        sc->params.drv_memwin = MEMWIN_NIC;

done:
        if (rc != DDI_SUCCESS) {
                (void) t4_devo_detach(dip, DDI_DETACH);

                /* rc may have errno style errors or DDI errors */
                rc = DDI_FAILURE;
        }

        return (rc);
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   96753 Tue Nov 20 20:17:43 2018
new/usr/src/uts/common/io/cxgbe/t4nex/t4_sge.c
9994 cxgbe t4nex: Handle get_fl_payload() alloc failures
9995 cxgbe t4_devo_attach() should initialize ->sfl
**********************************************************
______unchanged_portion_omitted_

/*
 * t4_ring_rx - Process responses from an SGE response queue.
 *
 * This function processes responses from an SGE response queue up to the suppli
 * Responses include received packets as well as control messages from FW
 * or HW.
 * It returns a chain of mblks containing the received data, to be
 * passed up to mac_ring_rx().
 */
mblk_t *
t4_ring_rx(struct sge_rxq *rxq, int budget)
{
        struct sge_iq *iq = &rxq->iq;
        struct sge_fl *fl = &rxq->fl;           /* Use iff IQ_HAS_FL */
        struct adapter *sc = iq->adapter;
        struct rsp_ctrl *ctrl;
        const struct rss_header *rss;
        int ndescs = 0, fl_bufs_used = 0;
        int rsp_type;
        uint32_t lq;
        mblk_t *mblk_head = NULL, **mblk_tail, *m;
        struct cpl_rx_pkt *cpl;
        uint32_t received_bytes = 0, pkt_len = 0;
        bool csum_ok;
        uint16_t err_vec;

        mblk_tail = &mblk_head;

        while (is_new_response(iq, &ctrl)) {

                membar_consumer();

                m = NULL;
                rsp_type = G_RSPD_TYPE(ctrl->u.type_gen);
                lq = be32_to_cpu(ctrl->pldbuflen_qid);
                rss = (const void *)iq->cdesc;

                switch (rsp_type) {
                case X_RSPD_TYPE_FLBUF:

                        ASSERT(iq->flags & IQ_HAS_FL);

                        if (CPL_RX_PKT == rss->opcode) {
                                cpl = (void *)(rss + 1);
                                pkt_len = be16_to_cpu(cpl->len);

                                if (iq->polling && ((received_bytes + pkt_len) >
                                        goto done;

                                m = get_fl_payload(sc, fl, lq, &fl_bufs_used);
                                if (m == NULL)
                                        goto done;
                                if (m == NULL) {
                                        panic("%s: line %d.", __func__,
                                            __LINE__);
                                }

                                iq->intr_next = iq->intr_params;
```

```
                                m->b_rptr += sc->sge.pktshift;
                                if (sc->params.tp.rx_pkt_encap)
                                /* It is enabled only in T6 config file */
                                        err_vec = G_T6_COMPR_RXERR_VEC(ntohs(cpl
                                else
                                        err_vec = ntohs(cpl->err_vec);

                                csum_ok = cpl->csum_calc && !err_vec;

                                /* TODO: what about cpl->ip_frag? */
                                if (csum_ok && !cpl->ip_frag) {
                                        mac_hcksum_set(m, 0, 0, 0, 0xffff,
                                            HCK_FULLCKSUM_OK | HCK_FULLCKSUM |
                                            HCK_IPV4_HDRCKSUM_OK);
                                        rxq->rxcsum++;
                                }
                                rxq->rxpkts++;
                                rxq->rxbytes += pkt_len;
                                received_bytes += pkt_len;

                                *mblk_tail = m;
                                mblk_tail = &m->b_next;

                                break;
                        }

                        m = get_fl_payload(sc, fl, lq, &fl_bufs_used);
                        if (m == NULL)
                                goto done;
                        if (m == NULL) {
                                panic("%s: line %d.", __func__,
                                    __LINE__);
                        }
                        /* FALLTHROUGH */

                case X_RSPD_TYPE_CPL:
                        ASSERT(rss->opcode < NUM_CPL_CMDS);
                        sc->cpl_handler[rss->opcode](iq, rss, m);
                        break;

                default:
                        break;
                }
                iq_next(iq);
                ++ndescs;
                if (!iq->polling && (ndescs == budget))
                        break;
        }

done:

        t4_write_reg(sc, MYPF_REG(A_SGE_PF_GTS),
                     V_CIDXINC(ndescs) | V_INGRESSQID(iq->cntxt_id) |
                     V_SEINTARM(V_QINTR_TIMER_IDX(X_TIMERREG_UPDATE_CIDX)));

        if ((fl_bufs_used > 0) || (iq->flags & IQ_HAS_FL)) {
                int starved;
                FL_LOCK(fl);
                fl->needed += fl_bufs_used;
                starved = refill_fl(sc, fl, fl->cap / 8);
                FL_UNLOCK(fl);
                if (starved)
                        add_fl_to_sfl(sc, fl);
        }
        return (mblk_head);
}
```

```
/*
 * Deals with anything and everything on the given ingress queue.
 */
static int
service_iq(struct sge_iq *iq, int budget)
{
        struct sge_iq *q;
        struct sge_rxq *rxq = iq_to_rxq(iq);    /* Use iff iq is part of rxq */
        struct sge_fl *fl = &rxq->fl;           /* Use iff IQ_HAS_FL */
        struct adapter *sc = iq->adapter;
        struct rsp_ctrl *ctrl;
        const struct rss_header *rss;
        int ndescs = 0, limit, fl_bufs_used = 0;
        int rsp_type;
        uint32_t lq;
        int starved;
        mblk_t *m;
        STAILQ_HEAD(, sge_iq) iql = STAILQ_HEAD_INITIALIZER(iql);

        limit = budget ? budget : iq->qsize / 8;

        /*
         * We always come back and check the descriptor ring for new indirect
         * interrupts and other responses after running a single handler.
         */
        for (;;) {
                while (is_new_response(iq, &ctrl)) {

                        membar_consumer();

                        m = NULL;
                        rsp_type = G_RSPD_TYPE(ctrl->u.type_gen);
                        lq = be32_to_cpu(ctrl->pldbuflen_qid);
                        rss = (const void *)iq->cdesc;

                        switch (rsp_type) {
                        case X_RSPD_TYPE_FLBUF:

                                ASSERT(iq->flags & IQ_HAS_FL);

                                m = get_fl_payload(sc, fl, lq, &fl_bufs_used);
                                if (m == NULL) {
                                        /*
                                         * Rearm the iq with a
                                         * longer-than-default timer
                                         */
                                        t4_write_reg(sc, MYPF_REG(A_SGE_PF_GTS),
                                                        V_INGRESSQID((u32)iq->cn
                                                        V_SEINTARM(V_QINTR_TIMER
                                        if (fl_bufs_used > 0) {
                                                ASSERT(iq->flags & IQ_HAS_FL);
                                                FL_LOCK(fl);
                                                fl->needed += fl_bufs_used;
                                                starved = refill_fl(sc, fl, fl->
                                                FL_UNLOCK(fl);
                                                if (starved)
                                                        add_fl_to_sfl(sc, fl);
                                        panic("%s: line %d.", __func__,
                                            __LINE__);
                                        }
                                        return (0);
                                }

                        /* FALLTHRU */
                        case X_RSPD_TYPE_CPL:
```

```
                                ASSERT(rss->opcode < NUM_CPL_CMDS);
                                sc->cpl_handler[rss->opcode](iq, rss, m);
                                break;

                        case X_RSPD_TYPE_INTR:

                                /*
                                 * Interrupts should be forwarded only to queues
                                 * that are not forwarding their interrupts.
                                 * This means service_iq can recurse but only 1
                                 * level deep.
                                 */
                                ASSERT(budget == 0);

                                q = sc->sge.iqmap[lq - sc->sge.iq_start];
                                if (atomic_cas_uint(&q->state, IQS_IDLE,
                                    IQS_BUSY) == IQS_IDLE) {
                                        if (service_iq(q, q->qsize / 8) == 0) {
                                                (void) atomic_cas_uint(
                                                    &q->state, IQS_BUSY,
                                                    IQS_IDLE);
                                        } else {
                                                STAILQ_INSERT_TAIL(&iql, q,
                                                    link);
                                        }
                                }
                                break;

                        default:
                                break;
                        }

                        iq_next(iq);
                        if (++ndescs == limit) {
                                t4_write_reg(sc, MYPF_REG(A_SGE_PF_GTS),
                                    V_CIDXINC(ndescs) |
                                    V_INGRESSQID(iq->cntxt_id) |
                                    V_SEINTARM(V_QINTR_TIMER_IDX(
                                    X_TIMERREG_UPDATE_CIDX)));
                                ndescs = 0;

                                if (fl_bufs_used > 0) {
                                        ASSERT(iq->flags & IQ_HAS_FL);
                                        FL_LOCK(fl);
                                        fl->needed += fl_bufs_used;
                                        (void) refill_fl(sc, fl, fl->cap / 8);
                                        FL_UNLOCK(fl);
                                        fl_bufs_used = 0;
                                }

                                if (budget != 0)
                                        return (EINPROGRESS);
                        }
                }

                if (STAILQ_EMPTY(&iql) != 0)
                        break;

                /*
                 * Process the head only, and send it to the back of the list if
                 * it's still not done.
                 */
                q = STAILQ_FIRST(&iql);
                STAILQ_REMOVE_HEAD(&iql, link);
                if (service_iq(q, q->qsize / 8) == 0)
```

```
                        (void) atomic_cas_uint(&q->state, IQS_BUSY, IQS_IDLE);
                else
                        STAILQ_INSERT_TAIL(&iql, q, link);
        }

        t4_write_reg(sc, MYPF_REG(A_SGE_PF_GTS), V_CIDXINC(ndescs) |
            V_INGRESSQID((u32)iq->cntxt_id) | V_SEINTARM(iq->intr_next));

        if (iq->flags & IQ_HAS_FL) {
                int starved;

                FL_LOCK(fl);
                fl->needed += fl_bufs_used;
                starved = refill_fl(sc, fl, fl->cap / 4);
                FL_UNLOCK(fl);
                if (starved != 0)
                        add_fl_to_sfl(sc, fl);
        }

        return (0);
}
______unchanged_portion_omitted_

static inline void
init_fl(struct sge_fl *fl, uint16_t qsize)
{

        fl->qsize = qsize;
        fl->allocb_fail = 0;
}
______unchanged_portion_omitted_

/*
 * Note that fl->cidx and fl->offset are left unchanged in case of failure.
 */
static mblk_t *
get_fl_payload(struct adapter *sc, struct sge_fl *fl,
               uint32_t len_newbuf, int *fl_bufs_used)
{
        struct mblk_pair frame = {0};
        struct rxbuf *rxb;
        mblk_t *m = NULL;
        uint_t nbuf = 0, len, copy, n;
        uint32_t cidx, offset, rcidx, roffset;
        uint32_t cidx, offset;

        /*
         * The SGE won't pack a new frame into the current buffer if the entire
         * payload doesn't fit in the remaining space.  Move on to the next buf
         * in that case.
         */
        rcidx = fl->cidx;
        roffset = fl->offset;
        if (fl->offset > 0 && len_newbuf & F_RSPD_NEWBUF) {
                fl->offset = 0;
                if (++fl->cidx == fl->cap)
                        fl->cidx = 0;
                nbuf++;
        }
        cidx = fl->cidx;
        offset = fl->offset;

        len = G_RSPD_LEN(len_newbuf);   /* pktshift + payload length */
        copy = (len <= fl->copy_threshold);
        if (copy != 0) {
                frame.head = m = allocb(len, BPRI_HI);
```

```
                if (m == NULL) {
                        fl->allocb_fail++;
                        cmn_err(CE_WARN,"%s: mbuf allocation failure "
                                        "count = %llu", __func__,
                                        (unsigned long long)fl->allocb_fail);
                        fl->cidx = rcidx;
                        fl->offset = roffset;
                if (m == NULL)
                        return (NULL);
                }
        }

        while (len) {
                rxb = fl->sdesc[cidx].rxb;
                n = min(len, rxb->buf_size - offset);

                (void) ddi_dma_sync(rxb->dhdl, offset, n,
                    DDI_DMA_SYNC_FORKERNEL);

                if (copy != 0)
                        bcopy(rxb->va + offset, m->b_wptr, n);
                else {
                        m = desballoc((unsigned char *)rxb->va + offset, n,
                            BPRI_HI, &rxb->freefunc);
                        if (m == NULL) {
                                fl->allocb_fail++;
                                cmn_err(CE_WARN,
                                        "%s: mbuf allocation failure "
                                        "count = %llu", __func__,
                                        (unsigned long long)fl->allocb_fail);
                                if (frame.head)
                                        freemsgchain(frame.head);
                                fl->cidx = rcidx;
                                fl->offset = roffset;
                                freemsg(frame.head);
                                return (NULL);
                        }
                        atomic_inc_uint(&rxb->ref_cnt);
                        if (frame.head != NULL)
                                frame.tail->b_cont = m;
                        else
                                frame.head = m;
                        frame.tail = m;
                }
                m->b_wptr += n;
                len -= n;
                offset += roundup(n, sc->sge.fl_align);
                ASSERT(offset <= rxb->buf_size);
                if (offset == rxb->buf_size) {
                        offset = 0;
                        if (++cidx == fl->cap)
                                cidx = 0;
                        nbuf++;
                }
        }

        fl->cidx = cidx;
        fl->offset = offset;
        (*fl_bufs_used) += nbuf;

        ASSERT(frame.head != NULL);
        return (frame.head);
}
______unchanged_portion_omitted_
```