

```
*****
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_____

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

262 struct sge_fl {
263     unsigned int flags;
264     kmutex_t lock;
265     ddi_dma_handle_t dhdl;
266     ddi_acc_handle_t ahdl;

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

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

286     TAILQ_ENTRY(sge_fl) link; /* All starving freelists */
287 };
_____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_____

```

```

276 static int
277 t4_devo_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
278 {
279     struct adapter *sc = NULL;
280     struct sge *s;
281     int i, instance, rc = DDI_SUCCESS, rqid, tqidx, q;
282     int irq = 0, n100g, n200g, n40g, n25g, n10g, nlg;
283 #ifdef TCP_OFFLOAD_ENABLE
284     int ofld_rqid, ofld_tqidx;
285 #endif
286     char name[16];
287     struct driver_properties *prp;
288     struct intrs_and_queues iaq;
289     ddi_device_acc_attr_t da = {
290         .devacc_attr_version = DDI_DEVICE_ATTR_V0,
291         .devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC,
292         .devacc_attr_dataorder = DDI_UNORDERED_OK_ACC
293     };
294     ddi_device_acc_attr_t dal = {
295         .devacc_attr_version = DDI_DEVICE_ATTR_V0,
296         .devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC,
297         .devacc_attr_dataorder = DDI_MERGING_OK_ACC
298     };
300     if (cmd != DDI_ATTACH)
301         return (DDI_FAILURE);
303     /*
304      * Allocate space for soft state.
305      */
306     instance = ddi_get_instance(dip);
307     rc = ddi_soft_state_zalloc(t4_list, instance);
308     if (rc != DDI_SUCCESS) {
309         cxgb_printf(dip, CE_WARN,
310             "failed to allocate soft state: %d", rc);
311         return (DDI_FAILURE);
312     }
314     sc = ddi_get_soft_state(t4_list, instance);
315     sc->dip = dip;
316     sc->dev = makedevice(ddi_driver_major(dip), instance);
317     mutex_init(&sc->lock, NULL, MUTEX_DRIVER, NULL);
318     cv_init(&sc->cv, NULL, CV_DRIVER, NULL);
319     mutex_init(&sc->sfl_lock, NULL, MUTEX_DRIVER, NULL);
320     TAILQ_INIT(&sc->sfl);
322     mutex_enter(&t4_adapter_list_lock);
323     SLIST_INSERT_HEAD(&t4_adapter_list, sc, link);
324     mutex_exit(&t4_adapter_list_lock);
326     sc->pf = getpf(sc);
327     if (sc->pf > 8) {
328         rc = EINVAL;
329         cxgb_printf(dip, CE_WARN,
330             "failed to determine PCI PF# of device");
331         goto done;
332     }
333     sc->mbox = sc->pf;

```

```

335     /* Initialize the driver properties */
336     prp = &sc->props;
337     (void)init_driver_props(sc, prp);
339     /*
340      * Enable access to the PCI config space.
341      */
342     rc = pci_config_setup(dip, &sc->pci_regh);
343     if (rc != DDI_SUCCESS) {
344         cxgb_printf(dip, CE_WARN,
345             "failed to enable PCI config space access: %d", rc);
346         goto done;
347     }
349     /* TODO: Set max read request to 4K */
351     /*
352      * Enable MMIO access.
353      */
354     rc = ddi_regs_map_setup(dip, 1, &sc->regp, 0, 0, &da, &sc->regh);
355     if (rc != DDI_SUCCESS) {
356         cxgb_printf(dip, CE_WARN,
357             "failed to map device registers: %d", rc);
358         goto done;
359     }
361     (void)memset(sc->chan_map, 0xff, sizeof (sc->chan_map));
363     /*
364      * Initialize cpl handler.
365      */
366     for (i = 0; i < ARRAY_SIZE(sc->cpl_handler); i++) {
367         sc->cpl_handler[i] = cpl_not_handled;
368     }
370     for (i = 0; i < ARRAY_SIZE(sc->fw_msg_handler); i++) {
371         sc->fw_msg_handler[i] = fw_msg_not_handled;
372     }
374     for (i = 0; i < NCHAN; i++) {
375         (void)snprintf(name, sizeof (name), "%s-%d",
376             "reclaim", i);
377         sc->tq[i] = ddi_taskq_create(sc->dip,
378             name, 1, TASKQ_DEFAULTPRI, 0);
380         if (sc->tq[i] == NULL) {
381             cxgb_printf(dip, CE_WARN,
382                 "failed to create task queues");
383             rc = DDI_FAILURE;
384             goto done;
385         }
386     }
388     /*
389      * Prepare the adapter for operation.
390      */
391     rc = -t4_prep_adapter(sc, false);
392     if (rc != 0) {
393         cxgb_printf(dip, CE_WARN, "failed to prepare adapter: %d", rc);
394         goto done;
395     }
397     /*
398      * Enable BAR1 access.
399      */

```

```

400     sc->doorbells |= DOORBELL_KDB;
401     rc = ddi_regs_map_setup(dip, 2, &sc->reglp, 0, 0, &dal, &sc->reglh);
402     if (rc != DDI_SUCCESS) {
403         cxgb_printf(dip, CE_WARN,
404             "failed to map BAR1 device registers: %d", rc);
405         goto done;
406     } else {
407         if (is_t5(sc->params.chip)) {
408             sc->doorbells |= DOORBELL_UDB;
409             if (prp->wc) {
410                 /*
411                  * Enable write combining on BAR2. This is the
412                  * userspace doorbell BAR and is split into 128B
413                  * (UDBS_SEG_SIZE) doorbell regions, each associ
414                  * with an egress queue. The first 64B has the
415                  * and the second 64B can be used to submit a tx
416                  * request with an implicit doorbell.
417                  */
418                 sc->doorbells &= ~DOORBELL_UDB;
419                 sc->doorbells |= (DOORBELL_WCWR |
420                     DOORBELL_UDBWC);
421                 t4_write_reg(sc, A_SGE_STAT_CFG,
422                     V_STATSOURCE_T5(7) | V_STATMODE(0));
423             }
424         }
425     }
426
427     /*
428     * Do this really early. Note that minor number = instance.
429     */
430     (void) snprintf(name, sizeof(name), "%s,%d", T4_NEXUS_NAME, instance);
431     rc = ddi_create_minor_node(dip, name, S_IFCHR, instance,
432         DDI_NT_NEXUS, 0);
433     if (rc != DDI_SUCCESS) {
434         cxgb_printf(dip, CE_WARN,
435             "failed to create device node: %d", rc);
436         rc = DDI_SUCCESS; /* carry on */
437     }
438
439     /* Do this early. Memory window is required for loading config file. */
440     setup_memwin(sc);
441
442     /* Prepare the firmware for operation */
443     rc = prep_firmware(sc);
444     if (rc != 0)
445         goto done; /* error message displayed already */
446
447     rc = adap_pre_init_tweaks(sc);
448     if (rc != 0)
449         goto done;
450
451     rc = get_params_pre_init(sc);
452     if (rc != 0)
453         goto done; /* error message displayed already */
454
455     t4_sge_init(sc);
456
457     if (sc->flags & MASTER_PF) {
458         /* get basic stuff going */
459         rc = -t4_fw_initialize(sc, sc->mbox);
460         if (rc != 0) {
461             cxgb_printf(sc->dip, CE_WARN,
462                 "early init failed: %d.\n", rc);
463             goto done;
464         }
465     }

```

```

467     rc = get_params_post_init(sc);
468     if (rc != 0)
469         goto done; /* error message displayed already */
470
471     rc = set_params_post_init(sc);
472     if (rc != 0)
473         goto done; /* error message displayed already */
474
475     /*
476     * TODO: This is the place to call t4_set_filter_mode()
477     */
478
479     /* tweak some settings */
480     t4_write_reg(sc, A_TP_SHIFT_CNT, V_SYNSHIFTMAX(6) | V_RXTSHIFTMAXR1(4) |
481         V_RXTSHIFTMAXR2(15) | V_PERSHIFTBACKOFFMAX(8) | V_PERSHIFTMAX(8) |
482         V_KEEPAALIVEMAXR1(4) | V_KEEPAALIVEMAXR2(9));
483     t4_write_reg(sc, A_ULP_RX_TDDP_PSZ, V_HPZ0(PAGE_SHIFT - 12));
484
485     /*
486     * Work-around for bug 2619
487     * Set DisableVlan field in TP_RSS_CONFIG_VRT register so that the
488     * VLAN tag extraction is disabled.
489     */
490     t4_set_reg_field(sc, A_TP_RSS_CONFIG_VRT, F_DISABLEVLAN, F_DISABLEVLAN);
491
492     /* Store filter mode */
493     t4_read_indirect(sc, A_TP_PIO_ADDR, A_TP_PIO_DATA, &sc->filter_mode, 1,
494         A_TP_VLAN_PRI_MAP);
495
496     /*
497     * First pass over all the ports - allocate VIs and initialize some
498     * basic parameters like mac address, port type, etc. We also figure
499     * out whether a port is 10G or 1G and use that information when
500     * calculating how many interrupts to attempt to allocate.
501     */
502     n100g = n40g = n25g = n10g = n1g = 0;
503     for_each_port(sc, i) {
504         struct port_info *pi;
505
506         pi = kmem_zalloc(sizeof(*pi), KM_SLEEP);
507         sc->port[i] = pi;
508
509         /* These must be set before t4_port_init */
510         pi->adapter = sc;
511         /* LINTED: E_ASSIGN_NARROW_CONV */
512         pi->port_id = i;
513     }
514
515     /* Allocate the vi and initialize parameters like mac addr */
516     rc = -t4_port_init(sc, sc->mbox, sc->pf, 0);
517     if (rc) {
518         cxgb_printf(dip, CE_WARN,
519             "unable to initialize port: %d", rc);
520         goto done;
521     }
522
523     for_each_port(sc, i) {
524         struct port_info *pi = sc->port[i];
525
526         mutex_init(&pi->lock, NULL, MUTEX_DRIVER, NULL);
527         pi->mtu = ETHERMTU;
528
529         if (is_100G_port(pi)) {
530             n100g++;
531             pi->tmr_idx = prp->tmr_idx_10g;

```

```

532         pi->pktc_idx = prp->pktc_idx_10g;
533     } else if (is_40G_port(pi)) {
534         n40g++;
535         pi->tmr_idx = prp->tmr_idx_10g;
536         pi->pktc_idx = prp->pktc_idx_10g;
537     } else if (is_25G_port(pi)) {
538         n25g++;
539         pi->tmr_idx = prp->tmr_idx_10g;
540         pi->pktc_idx = prp->pktc_idx_10g;
541     } else if (is_10G_port(pi)) {
542         n10g++;
543         pi->tmr_idx = prp->tmr_idx_10g;
544         pi->pktc_idx = prp->pktc_idx_10g;
545     } else {
546         nlg++;
547         pi->tmr_idx = prp->tmr_idx_lg;
548         pi->pktc_idx = prp->pktc_idx_lg;
549     }
550
551     pi->xact_addr_filt = -1;
552     t4_mc_init(pi);
553
554     setbit(&sc->registered_device_map, i);
555 }
556
557 nxg = n10g + n25g + n40g + n100g;
558 (void) remove_extra_props(sc, nxg, nlg);
559
560 if (sc->registered_device_map == 0) {
561     cxgb_printf(dip, CE_WARN, "no usable ports");
562     rc = DDI_FAILURE;
563     goto done;
564 }
565
566 rc = cfg_itype_and_nqueues(sc, nxg, nlg, &iaq);
567 if (rc != 0)
568     goto done; /* error message displayed already */
569
570 sc->intr_type = iaq.intr_type;
571 sc->intr_count = iaq.nirq;
572
573 if (sc->props.multi_rings && (sc->intr_type != DDI_INTR_TYPE_MSIX)) {
574     sc->props.multi_rings = 0;
575     cxgb_printf(dip, CE_WARN,
576         "Multiple rings disabled as interrupt type is not MSI-X");
577 }
578
579 if (sc->props.multi_rings && iaq.intr_fwd) {
580     sc->props.multi_rings = 0;
581     cxgb_printf(dip, CE_WARN,
582         "Multiple rings disabled as interrupts are forwarded");
583 }
584
585 if (!sc->props.multi_rings) {
586     iaq.ntxq10g = 1;
587     iaq.ntxqlg = 1;
588 }
589 s = &sc->sge;
590 s->nrxq = nxg * iaq.nrxq10g + nlg * iaq.nrxqlg;
591 s->ntxq = nxg * iaq.ntxq10g + nlg * iaq.ntxqlg;
592 s->neq = s->ntxq + s->nrxq; /* the fl in an rxq is an eq */
593 #ifdef TCP_OFFLOAD_ENABLE
594 /* control queues, 1 per port + 1 mgmtq */
595 s->neq += sc->params.nports + 1;
596 #endif
597 s->niq = s->nrxq + 1; /* 1 extra for firmware event queue */

```

```

598     if (iaq.intr_fwd != 0)
599         sc->flags |= INTR_FWD;
600 #ifdef TCP_OFFLOAD_ENABLE
601     if (is_offload(sc) != 0) {
602
603         s->nofldrxq = nxg * iaq.nofldrxq10g + nlg * iaq.nofldrxqlg;
604         s->nofldtxq = nxg * iaq.nofldtxq10g + nlg * iaq.nofldtxqlg;
605         s->neq += s->nofldtxq + s->nofldrxq;
606         s->niq += s->nofldrxq;
607
608         s->ofld_rxq = kmem_zalloc(s->nofldrxq *
609             sizeof(struct sge_ofld_rxq), KM_SLEEP);
610         s->ofld_txq = kmem_zalloc(s->nofldtxq *
611             sizeof(struct sge_wrq), KM_SLEEP);
612         s->ctrlq = kmem_zalloc(sc->params.nports *
613             sizeof(struct sge_wrq), KM_SLEEP);
614     }
615 #endif
616
617 s->rxq = kmem_zalloc(s->nrxq * sizeof(struct sge_rxq), KM_SLEEP);
618 s->txq = kmem_zalloc(s->ntxq * sizeof(struct sge_txq), KM_SLEEP);
619 s->iqmap = kmem_zalloc(s->niq * sizeof(struct sge_iq *), KM_SLEEP);
620 s->eqmap = kmem_zalloc(s->neq * sizeof(struct sge_eq *), KM_SLEEP);
621
622 sc->intr_handle = kmem_zalloc(sc->intr_count *
623     sizeof(ddd_intr_handle_t), KM_SLEEP);
624
625 /*
626  * Second pass over the ports. This time we know the number of rx and
627  * tx queues that each port should get.
628  */
629 rqid = tqid = 0;
630 #ifdef TCP_OFFLOAD_ENABLE
631 ofld_rqid = ofld_tqid = 0;
632 #endif
633 for_each_port(sc, i) {
634     struct port_info *pi = sc->port[i];
635
636     if (pi == NULL)
637         continue;
638
639     t4_mc_cb_init(pi);
640     /* LINTED: E_ASSIGN_NARROW_CONV */
641     pi->first_rxq = rqid;
642     /* LINTED: E_ASSIGN_NARROW_CONV */
643     pi->nrxq = (is_10XG_port(pi)) ? iaq.nrxq10g
644         : iaq.nrxqlg;
645     /* LINTED: E_ASSIGN_NARROW_CONV */
646     pi->first_txq = tqid;
647     /* LINTED: E_ASSIGN_NARROW_CONV */
648     pi->ntxq = (is_10XG_port(pi)) ? iaq.ntxq10g
649         : iaq.ntxqlg;
650
651     rqid += pi->nrxq;
652     tqid += pi->ntxq;
653
654 #ifdef TCP_OFFLOAD_ENABLE
655     if (is_offload(sc) != 0) {
656         /* LINTED: E_ASSIGN_NARROW_CONV */
657         pi->first_ofld_rxq = ofld_rqid;
658         pi->nofldrxq = max(1, pi->nrxq / 4);
659
660         /* LINTED: E_ASSIGN_NARROW_CONV */
661         pi->first_ofld_txq = ofld_tqid;
662         pi->nofldtxq = max(1, pi->ntxq / 2);

```

```

664         ofld_rqidx += pi->nofldrnxq;
665         ofld_tqidx += pi->nofldtxq;
666     }
667 #endif

669     /*
670     * Enable hw checksumming and LSO for all ports by default.
671     * They can be disabled using ndd (hw_csum and hw_lso).
672     */
673     pi->features |= (CXGBE_HW_CSUM | CXGBE_HW_LSO);
674 }

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

680     /*
681     * Setup Interrupts.
682     */

684     i = 0;
685     rc = ddi_intr_alloc(dip, sc->intr_handle, sc->intr_type, 0,
686         sc->intr_count, &i, DDI_INTR_ALLOC_STRICT);
687     if (rc != DDI_SUCCESS) {
688         cxgb_printf(dip, CE_WARN,
689             "failed to allocate %d interrupt(s) of type %d: %d, %d",
690             sc->intr_count, sc->intr_type, rc, i);
691         goto done;
692     }
693     ASSERT(sc->intr_count == i); /* allocation was STRICT */
694     (void) ddi_intr_get_cap(sc->intr_handle[0], &sc->intr_cap);
695     (void) ddi_intr_get_pri(sc->intr_handle[0], &sc->intr_pri);
696     if (sc->intr_count == 1) {
697         ASSERT(sc->flags & INTR_FWD);
698         (void) ddi_intr_add_handler(sc->intr_handle[0], t4_intr_all, sc,
699             &sc->fwq);
700     } else {
701         /* Multiple interrupts. The first one is always error intr */
702         (void) ddi_intr_add_handler(sc->intr_handle[0], t4_intr_err, sc,
703             NULL);
704         irq++;

706         /* The second one is always the firmware event queue */
707         (void) ddi_intr_add_handler(sc->intr_handle[1], t4_intr, sc,
708             &sc->fwq);
709         irq++;
710         /*
711         * Note that if INTR_FWD is set then either the NIC rx
712         * queues or (exclusive or) the TOE rx queueus will be taking
713         * direct interrupts.
714         *
715         * There is no need to check for is_offload(sc) as nofldrxq
716         * will be 0 if offload is disabled.
717         */
718         for_each_port(sc, i) {
719             struct port_info *pi = sc->port[i];
720             struct sge_rnxq *rnxq;
721 #ifdef TCP_OFFLOAD_ENABLE
722             struct sge_ofld_rnxq *ofld_rnxq;

724             /*
725             * Skip over the NIC queues if they aren't taking direct
726             * interrupts.
727             */
728             if ((sc->flags & INTR_FWD) &&
729                 pi->nofldrnxq > pi->nrxq)

```

```

730         goto ofld_queues;
731 #endif
732         rnxq = &sc->rnxq[pi->first_rnxq];
733         for (q = 0; q < pi->nrxq; q++, rnxq++) {
734             (void) ddi_intr_add_handler(
735                 sc->intr_handle[irq], t4_intr, sc,
736                 &rnxq->iq);
737             irq++;
738         }

740 #ifdef TCP_OFFLOAD_ENABLE
741     /*
742     * Skip over the offload queues if they aren't taking
743     * direct interrupts.
744     */
745     if ((sc->flags & INTR_FWD))
746         continue;
747 ofld_queues:
748     ofld_rnxq = &sc->ofld_rnxq[pi->first_ofld_rnxq];
749     for (q = 0; q < pi->nofldrnxq; q++, ofld_rnxq++) {
750         (void) ddi_intr_add_handler(
751             sc->intr_handle[irq], t4_intr, sc,
752             &ofld_rnxq->iq);
753         irq++;
754     }
755 #endif
756 }

758 }
759 sc->flags |= INTR_ALLOCATED;

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

764     /*
765     * Hardware/Firmware/etc. Version/Revision IDs.
766     */
767     t4_dump_version_info(sc);

769     if (nl00g) {
770         cxgb_printf(dip, CE_NOTE,
771             "%dx100G (%d rnxq, %d txq total) %d %s.",
772             nl00g, rqid, tqidx, sc->intr_count,
773             sc->intr_type == DDI_INTR_TYPE_MSIX ? "MSI-X interrupts" :
774             sc->intr_type == DDI_INTR_TYPE_MSI ? "MSI interrupts" :
775             "fixed interrupt");
776     } else if (n40g) {
777         cxgb_printf(dip, CE_NOTE,
778             "%dx40G (%d rnxq, %d txq total) %d %s.",
779             n40g, rqid, tqidx, sc->intr_count,
780             sc->intr_type == DDI_INTR_TYPE_MSIX ? "MSI-X interrupts" :
781             sc->intr_type == DDI_INTR_TYPE_MSI ? "MSI interrupts" :
782             "fixed interrupt");
783     } else if (n25g) {
784         cxgb_printf(dip, CE_NOTE,
785             "%dx25G (%d rnxq, %d txq total) %d %s.",
786             n25g, rqid, tqidx, sc->intr_count,
787             sc->intr_type == DDI_INTR_TYPE_MSIX ? "MSI-X interrupts" :
788             sc->intr_type == DDI_INTR_TYPE_MSI ? "MSI interrupts" :
789             "fixed interrupt");
790     } else if (nl0g && nl1g) {
791         cxgb_printf(dip, CE_NOTE,
792             "%dx10G %dx1G (%d rnxq, %d txq total) %d %s.",
793             nl0g, nl1g, rqid, tqidx, sc->intr_count,
794             sc->intr_type == DDI_INTR_TYPE_MSIX ? "MSI-X interrupts" :
795             sc->intr_type == DDI_INTR_TYPE_MSI ? "MSI interrupts" :

```

```
796         "fixed interrupt");
797     } else {
798         cxgb_printf(dip, CE_NOTE,
799             "%dx%sG (%d rxq, %d txq per port) %d %s.",
800             n10g ? n10g : n1g,
801             n10g ? "10" : "1",
802             n10g ? iaq.nrxql0g : iaq.nrxqlg,
803             n10g ? iaq.ntxql0g : iaq.ntxqlg,
804             sc->intr_count,
805             sc->intr_type == DDI_INTR_TYPE_MSIX ? "MSI-X interrupts" :
806             sc->intr_type == DDI_INTR_TYPE_MSI ? "MSI interrupts" :
807             "fixed interrupt");
808     }
809
810     sc->ksp = setup_kstats(sc);
811     sc->ksp_stat = setup_wc_kstats(sc);
812     sc->params.drv_memwin = MEMWIN_NIC;
813
814 done:
815     if (rc != DDI_SUCCESS) {
816         (void) t4_devo_detach(dip, DDI_DETACH);
817
818         /* rc may have errno style errors or DDI errors */
819         rc = DDI_FAILURE;
820     }
821
822     return (rc);
823 }
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_____

728 /*
729 * t4_ring_rx - Process responses from an SGE response queue.
730 *
731 * This function processes responses from an SGE response queue up to the suppli
732 * Responses include received packets as well as control messages from FW
733 * or HW.
734 * It returns a chain of mblks containing the received data, to be
735 * passed up to mac_ring_rx().
736 */
737 mblk_t *
738 t4_ring_rx(struct sge_rxq *rxq, int budget)
739 {
740     struct sge_iq *iq = &rxq->iq;
741     struct sge_fl *fl = &rxq->fl;          /* Use iff IQ_HAS_FL */
742     struct adapter *sc = iq->adapter;
743     struct rsp_ctrl *ctrl;
744     const struct rss_header *rss;
745     int ndescs = 0, fl_bufs_used = 0;
746     int rsp_type;
747     uint32_t lq;
748     mblk_t *mblk_head = NULL, **mblk_tail, *m;
749     struct cpl_rx_pkt *cpl;
750     uint32_t received_bytes = 0, pkt_len = 0;
751     bool csum_ok;
752     uint16_t err_vec;

754     mblk_tail = &mblk_head;

756     while (is_new_response(iq, &ctrl)) {

758         membar_consumer();

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

765         switch (rsp_type) {
766             case X_RSPD_TYPE_FLBUF:

768                 ASSERT(iq->flags & IQ_HAS_FL);

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

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

777                     m = get_fl_payload(sc, fl, lq, &fl_bufs_used);
778                     if (m == NULL)
779                         goto done;
778                     if (m == NULL) {
779                         panic("%s: line %d.", __func__,
780                             __LINE__);
781                     }

784                     if (m == NULL)
785                         goto done;
786                     if (m == NULL) {
787                         panic("%s: line %d.", __func__,
788                             __LINE__);
789                     }

791                     iq->intr_next = iq->intr_params;

```

```

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

789         csum_ok = cpl->csum_calc && !err_vec;

791         /* TODO: what about cpl->ip_frag? */
792         if (csum_ok && !cpl->ip_frag) {
793             mac_hcksum_set(m, 0, 0, 0, 0xffff,
794                 HCK_FULLLCKSUM_OK | HCK_FULLCKSUM |
795                 HCK_IPV4_HDRCKSUM_OK);
796             rxq->rxcsument++;
797         }
798         rxq->rxpkts++;
799         rxq->rxbytes += pkt_len;
800         received_bytes += pkt_len;

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

805         break;
806     }

808     m = get_fl_payload(sc, fl, lq, &fl_bufs_used);
809     if (m == NULL)
810         goto done;
811     if (m == NULL) {
812         panic("%s: line %d.", __func__,
813             __LINE__);
814     }
815     /* FALLTHROUGH */

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

820     default:
821         break;
822     }
823     iq_next(iq);
824     ++ndescs;
825     if (!iq->polling && (ndescs == budget))
826         break;

827 done:

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

833     if ((fl_bufs_used > 0) || (iq->flags & IQ_HAS_FL)) {
834         int starved;
835         FL_LOCK(fl);
836         fl->needed += fl_bufs_used;
837         starved = refill_fl(sc, fl, fl->cap / 8);
838         FL_UNLOCK(fl);
839         if (starved)
840             add_fl_to_sfl(sc, fl);
841     }
842     return (mblk_head);
843 }

```

```

845 /*
846  * Deals with anything and everything on the given ingress queue.
847  */
848 static int
849 service_iq(struct sge_iq *iq, int budget)
850 {
851     struct sge_iq *q;
852     struct sge_rxq *rxq = iq_to_rxq(iq);    /* Use iff iq is part of rxq */
853     struct sge_fl *fl = &rxq->fl;        /* Use iff IQ_HAS_FL */
854     struct adapter *sc = iq->adapter;
855     struct rsp_ctrl *ctrl;
856     const struct rss_header *rss;
857     int ndescs = 0, limit, fl_bufs_used = 0;
858     int rsp_type;
859     uint32_t lq;
860     int starved;
861     mblk_t *m;
862     STAILQ_HEAD(, sge_iq) iql = STAILQ_HEAD_INITIALIZER(iql);

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

866     /*
867     * We always come back and check the descriptor ring for new indirect
868     * interrupts and other responses after running a single handler.
869     */
870     for (;;) {
871         while (is_new_response(iq, &ctrl)) {

873             membar_consumer();

875             m = NULL;
876             rsp_type = G_RSPD_TYPE(ctrl->u.type_gen);
877             lq = be32_to_cpu(ctrl->pldbufid_qid);
878             rss = (const void *)iq->cdesc;

880             switch (rsp_type) {
881             case X_RSPD_TYPE_FLBUF:

883                 ASSERT(iq->flags & IQ_HAS_FL);

885                 m = get_fl_payload(sc, fl, lq, &fl_bufs_used);
886                 if (m == NULL) {
887                     /*
888                      * Rearm the iq with a
889                      * longer-than-default timer
890                      */
891                     t4_write_reg(sc, MYPF_REG(A_SGE_PF_GTS),
892                                 V_INGRESSQID((u32)iq->cn
893                                               V_SEINTARM(V_QINTR_TIMER
894                                               if (fl_bufs_used > 0) {
895                                 ASSERT(iq->flags & IQ_HAS_FL);
896                                 FL_LOCK(fl);
897                                 fl->needed += fl_bufs_used;
898                                 starved = refill_fl(sc, fl, fl->
899                                 FL_UNLOCK(fl);
900                                 if (starved)
901                                     add_fl_to_sfl(sc, fl);
902                                 panic("%s: line %d.", __func__,
903                                       __LINE__);
904                             }
905                         }
906                     return (0);
907                 }
908             }

909             /* FALLTHRU */
910             case X_RSPD_TYPE_CPL:

```

```

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

913             case X_RSPD_TYPE_INTR:

915                 /*
916                  * Interrupts should be forwarded only to queues
917                  * that are not forwarding their interrupts.
918                  * This means service_iq can recurse but only 1
919                  * level deep.
920                  */
921                 ASSERT(budget == 0);

923                 q = sc->sge.iqmap[lq - sc->sge.iq_start];
924                 if (atomic_cas_uint(&q->state, IQS_IDLE,
925                                   IQS_BUSY) == IQS_IDLE) {
926                     if (service_iq(q, q->qsize / 8) == 0) {
927                         (void) atomic_cas_uint(
928                             &q->state, IQS_BUSY,
929                             IQS_IDLE);
930                     } else {
931                         STAILQ_INSERT_TAIL(&iql, q,
932                                           link);
933                     }
934                 }
935                 break;

937             default:
938                 break;
939             }

941             iq_next(iq);
942             if (++ndescs == limit) {
943                 t4_write_reg(sc, MYPF_REG(A_SGE_PF_GTS),
944                             V_CIDXINC(ndescs) |
945                             V_INGRESSQID(iq->cntxt_id) |
946                             V_SEINTARM(V_QINTR_TIMER_IDX(
947                                 X_TIMERREG_UPDATE_CIDX)));
948                 ndescs = 0;

950                 if (fl_bufs_used > 0) {
951                     ASSERT(iq->flags & IQ_HAS_FL);
952                     FL_LOCK(fl);
953                     fl->needed += fl_bufs_used;
954                     (void) refill_fl(sc, fl, fl->cap / 8);
955                     FL_UNLOCK(fl);
956                     fl_bufs_used = 0;
957                 }

959                 if (budget != 0)
960                     return (EINPROGRESS);
961             }
962         }

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

967         /*
968         * Process the head only, and send it to the back of the list if
969         * it's still not done.
970         */
971         q = STAILQ_FIRST(&iql);
972         STAILQ_REMOVE_HEAD(&iql, link);
973         if (service_iq(q, q->qsize / 8) == 0)

```

```

974         (void) atomic_cas_uint(&q->state, IQS_BUSY, IQS_IDLE);
975     else
976         STAILQ_INSERT_TAIL(&iql, q, link);
977 }

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

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

984     FL_LOCK(fl);
985     fl->needed += fl_bufs_used;
986     starved = refill_fl(sc, fl, fl->cap / 4);
987     FL_UNLOCK(fl);
988     if (starved != 0)
989         add_fl_to_sfl(sc, fl);
990 }

992 return (0);
993 }

```

unchanged portion omitted

```

1262 static inline void
1263 init_fl(struct sge_fl *fl, uint16_t qsize)
1264 {
1266     fl->qsize = qsize;
1267     fl->allocb_fail = 0;
1268 }

```

unchanged portion omitted

```

2335 /*
2336  * Note that fl->cidx and fl->offset are left unchanged in case of failure.
2337  */
2338 static mblk_t *
2339 get_fl_payload(struct adapter *sc, struct sge_fl *fl,
2340     uint32_t len_newbuf, int *fl_bufs_used)
2341 {
2342     struct mblk_pair frame = {0};
2343     struct rxbuf *rxbuf;
2344     mblk_t *m = NULL;
2345     uint_t nbuf = 0, len, copy, n;
2346     uint32_t cidx, offset, rcidx, roffset;
2347     uint32_t cidx, offset;

2348     /*
2349      * The SGE won't pack a new frame into the current buffer if the entire
2350      * payload doesn't fit in the remaining space. Move on to the next buf
2351      * in that case.
2352      */
2353     rcidx = fl->cidx;
2354     roffset = fl->offset;
2355     if (fl->offset > 0 && len_newbuf & F_RSPD_NEWBUF) {
2356         fl->offset = 0;
2357         if (++fl->cidx == fl->cap)
2358             fl->cidx = 0;
2359         nbuf++;
2360     }
2361     cidx = fl->cidx;
2362     offset = fl->offset;

2364     len = G_RSPD_LEN(len_newbuf); /* pktshift + payload length */
2365     copy = (len <= fl->copy_threshold);
2366     if (copy != 0) {
2367         frame.head = m = allocb(len, BPRI_HI);

```

```

2368     if (m == NULL) {
2369         fl->allocb_fail++;
2370         cmn_err(CE_WARN, "%s: mbuf allocation failure "
2371             "count = %llu", __func__,
2372             (unsigned long long)fl->allocb_fail);
2373         fl->cidx = rcidx;
2374         fl->offset = roffset;
2375     }
2376     if (m == NULL)
2377         return (NULL);
2378 }

2379 while (len) {
2380     rxbuf = fl->sdesc[cidx].rxbuf;
2381     n = min(len, rxbuf->buf_size - offset);

2383     (void) ddi_dma_sync(rxbuf->dhdl, offset, n,
2384         DDI_DMA_SYNC_FORKERNEL);

2386     if (copy != 0)
2387         bcopy(rxbuf->va + offset, m->b_wptr, n);
2388     else {
2389         m = desballoc((unsigned char *)rxbuf->va + offset, n,
2390             BPRI_HI, &rxbuf->freefunc);
2391         if (m == NULL) {
2392             fl->allocb_fail++;
2393             cmn_err(CE_WARN,
2394                 "%s: mbuf allocation failure "
2395                 "count = %llu", __func__,
2396                 (unsigned long long)fl->allocb_fail);
2397             if (frame.head)
2398                 freemsgchain(frame.head);
2399             fl->cidx = rcidx;
2400             fl->offset = roffset;
2401             freemsg(frame.head);
2402             return (NULL);
2403         }
2404         atomic_inc_uint(&rxbuf->ref_cnt);
2405         if (frame.head != NULL)
2406             frame.tail->b_cont = m;
2407         else
2408             frame.head = m;
2409         frame.tail = m;
2410     }
2411     m->b_wptr += n;
2412     len -= n;
2413     offset += roundup(n, sc->sge.fl_align);
2414     ASSERT(offset <= rxbuf->buf_size);
2415     if (offset == rxbuf->buf_size) {
2416         offset = 0;
2417         if (++cidx == fl->cap)
2418             cidx = 0;
2419         nbuf++;
2420     }
2421 }

2422 fl->cidx = cidx;
2423 fl->offset = offset;
2424 (*fl_bufs_used) += nbuf;

2426     ASSERT(frame.head != NULL);
2427     return (frame.head);
2428 }

```

unchanged portion omitted