

[PATCH 2/10] cxgb3 – main source file

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2006-11/msg05508.html>

- *From:* "Divy Le Ray <divy@xxxxxxxxxxxx>" <divy@xxxxxxxxxxxx>
 - *Date:* Fri, 17 Nov 2006 12:23:28 -0800
-

From: Divy Le Ray <divy@xxxxxxxxxxxx>

This patch implements the main source file for the Chelsio T3 network adapter driver.

Signed-off-by: Divy Le Ray <divy@xxxxxxxxxxxx>

drivers/net/cxgb3/cxgb3_main.c | 2443 +++
1 files changed, 2443 insertions(+), 0 deletions(-)

```
diff --git a/drivers/net/cxgb3/cxgb3_main.c b/drivers/net/cxgb3/cxgb3_main.c
new file mode 100644
index 0000000..d7af78c
--- /dev/null
+++ b/drivers/net/cxgb3/cxgb3_main.c
@@ -0,0 +1,2443 @@
+/*
+ * This file is part of the Chelsio T3 Ethernet driver for Linux.
+ *
+ * Copyright (C) 2003-2006 Chelsio Communications. All rights reserved.
+ *
+ * This program is distributed in the hope that it will be useful, but WITHOUT
+ * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
+ * FITNESS FOR A PARTICULAR PURPOSE. See the LICENSE file included in this
+ * release for licensing terms and conditions.
+ */
+
+#include <linux/module.h>
+#include <linux/moduleparam.h>
+#include <linux/init.h>
+#include <linux/pci.h>
+#include <linux/dma-mapping.h>
+#include <linux/netdevice.h>
+#include <linux/etherdevice.h>
+#include <linux/if_vlan.h>
+#include <linux/mii.h>
+#include <linux/sockios.h>
+#include <linux/workqueue.h>
```

[PATCH 2/10] cxgb3 – main source file

```
+#include <linux/proc_fs.h>
+#include <linux/rtnetlink.h>
+#include <asm/uaccess.h>
+#include <linux/debugfs.h>
+
+#include "common.h"
+#include "cxgb3_ioctl.h"
+#include "regs.h"
+#include "cxgb3_offload.h"
+#include "version.h"
+
+#include "cxgb3_ctl_defs.h"
+#include "t3_cpl.h"
+#include "firmware_exports.h"
+
+enum {
+ MAX_TXQ_ENTRIES = 16384,
+ MAX_CTRL_TXQ_ENTRIES = 1024,
+ MAX_RSPQ_ENTRIES = 16384,
+ MAX_RX_BUFFERS = 16384,
+ MAX_RX_JUMBO_BUFFERS = 16384,
+ MIN_TXQ_ENTRIES = 4,
+ MIN_CTRL_TXQ_ENTRIES = 4,
+ MIN_RSPQ_ENTRIES = 32,
+ MIN_FL_ENTRIES = 32
+};
+
+#define PORT_MASK ((1 << MAX_NPORTS) - 1)
+
+#define DFLT_MSG_ENABLE (NETIF_MSG_DRV | NETIF_MSG_PROBE | NETIF_MSG_LINK | \
+ NETIF_MSG_TIMER | NETIF_MSG_IFDOWN | NETIF_MSG_IFUP | \
+ NETIF_MSG_RX_ERR | NETIF_MSG_TX_ERR)
+
+#define EEPROM_MAGIC 0x38E2F10C
+
+#define to_net_dev(class) container_of(class, struct net_device, class_dev)
+
+#define CH_DEVICE(devid, ssid, idx) \
+ { PCI_VENDOR_ID_CHELSIO, devid, PCI_ANY_ID, ssid, 0, 0, idx }
+
+static struct pci_device_id cxgb3_pci_tbl[] = {
+ CH_DEVICE(0x20, 1, 0), /* PE9000 */
+ CH_DEVICE(0x21, 1, 1), /* T302E */
+ CH_DEVICE(0x22, 1, 2), /* T310E */
+ CH_DEVICE(0x23, 1, 3), /* T320X */
+ CH_DEVICE(0x24, 1, 1), /* T302X */
+ CH_DEVICE(0x25, 1, 3), /* T320E */
+ CH_DEVICE(0x26, 1, 2), /* T310X */
+ CH_DEVICE(0x30, 1, 2), /* T3B10 */
+ CH_DEVICE(0x31, 1, 3), /* T3B20 */
+ CH_DEVICE(0x32, 1, 1), /* T3B02 */
+};
```

[PATCH 2/10] cxgb3 – main source file

```
+ { 0, }
+};
+
+MODULE_DESCRIPTION(DRV_DESC);
+MODULE_AUTHOR("Chelsio Communications");
+MODULE_LICENSE("GPL");
+MODULE_VERSION(DRV_VERSION);
+MODULE_DEVICE_TABLE(pci, cxgb3_pci_tbl);
+
+static int dflt_msg_enable = DFLT_MSG_ENABLE;
+
+module_param(dflt_msg_enable, int, 0644);
+MODULE_PARM_DESC(dflt_msg_enable, "Chelsio T3 default message enable bitmap");
+
+/*
+ * The driver uses the best interrupt scheme available on a platform in the
+ * order MSI-X, MSI, legacy pin interrupts. This parameter determines which
+ * of these schemes the driver may consider as follows:
+ *
+ * msi = 2: choose from among all three options
+ * msi = 1: only consider MSI and pin interrupts
+ * msi = 0: force pin interrupts
+ */
+static int msi = 2;
+
+module_param(msi, int, 0644);
+MODULE_PARM_DESC(msi, "whether to use MSI or MSI-X");
+
+/*
+ * The driver enables offload as a default.
+ * To disable it, use ofld_disable = 1.
+ */
+
+static int ofld_disable = 0;
+
+module_param(ofld_disable, int, 0644);
+MODULE_PARM_DESC(ofld_disable, "whether to enable offload at init time or not");
+
+/*
+ * We have work elements that we need to cancel when an interface is taken
+ * down. Normally the work elements would be executed by keventd but that
+ * can deadlock because of linkwatch. If our close method takes the rtnl
+ * lock and linkwatch is ahead of our work elements in keventd, linkwatch
+ * will block keventd as it needs the rtnl lock, and we'll deadlock waiting
+ * for our work to complete. Get our own work queue to solve this.
+ */
+static struct workqueue_struct *cxgb3_wq;
+
+/**
+ * link_report – show link status and link speed/duplex
+ * @p: the port whose settings are to be reported
```

[PATCH 2/10] cxgb3 – main source file

```
+ *
+ * Shows the link status, speed, and duplex of a port.
+ */
+static void link_report(struct port_info *p)
+{
+ if (!netif_carrier_ok(p->dev))
+ printk(KERN_INFO "%s: link down\n", p->dev->name);
+ else {
+ const char *s = "10Mbps";
+
+ switch (p->link_config.speed) {
+ case SPEED_10000: s = "10Gbps"; break;
+ case SPEED_1000: s = "1000Mbps"; break;
+ case SPEED_100: s = "100Mbps"; break;
+ }
+
+ printk(KERN_INFO "%s: link up, %s, %s-duplex\n",
+ p->dev->name, s,
+ p->link_config.duplex == DUPLEX_FULL ? "full" : "half");
+ }
+ }
+
+/**
+ * t3_os_link_changed – handle link status changes
+ * @adapter: the adapter associated with the link change
+ * @port_id: the port index whose link status has changed
+ * @link_stat: the new status of the link
+ * @speed: the new speed setting
+ * @duplex: the new duplex setting
+ * @pause: the new flow-control setting
+ *
+ * This is the OS-dependent handler for link status changes. The OS
+ * neutral handler takes care of most of the processing for these events,
+ * then calls this handler for any OS-specific processing.
+ */
+void t3_os_link_changed(struct adapter *adapter, int port_id, int link_stat,
+ int speed, int duplex, int pause)
+{
+ struct port_info *p = &adapter->port[port_id];
+
+ /* Skip changes from disabled ports. */
+ if (!netif_running(p->dev))
+ return;
+
+ if (link_stat != netif_carrier_ok(p->dev)) {
+ if (link_stat)
+ netif_carrier_on(p->dev);
+ else
+ netif_carrier_off(p->dev);
+
+ if (adapter->params.rev > 0 && uses_xaui(adapter))
```

[PATCH 2/10] cxgb3 – main source file

```
+ t3_write_reg(adapter,  
+ XGM_REG(A_XGM_XAUI_ACT_CTRL, port_id),  
+ link_stat ? F_TXACTENABLE | F_RXEN : 0);  
+ link_report(p);  
+ }  
+}  
+  
+static void cxgb_set_rxmode(struct net_device *dev)  
+{  
+ struct t3_rx_mode rm;  
+ struct adapter *adapter = dev->priv;  
+  
+ init_rx_mode(&rm, dev, dev->mc_list);  
+ t3_mac_set_rx_mode(&adapter->port[dev->if_port].mac, &rm);  
+}  
+  
+/**  
+ * link_start – enable a port  
+ * @p: the port to enable  
+ *  
+ * Performs the MAC and PHY actions needed to enable a port.  
+ */  
+static void link_start(struct port_info *p)  
+{  
+ struct t3_rx_mode rm;  
+ struct cmac *mac = &p->mac;  
+ struct net_device *dev = p->dev;  
+  
+ init_rx_mode(&rm, dev, dev->mc_list);  
+ t3_mac_reset(mac);  
+ t3_mac_set_mtu(mac, dev->mtu);  
+ t3_mac_set_address(mac, 0, dev->dev_addr);  
+ t3_mac_set_rx_mode(mac, &rm);  
+ t3_link_start(&p->phy, mac, &p->link_config);  
+ t3_mac_enable(mac, MAC_DIRECTION_RX | MAC_DIRECTION_TX);  
+}  
+  
+static inline void cxgb_disable_msi(struct adapter *adapter)  
+{  
+ if (adapter->flags & USING_MSIX) {  
+ pci_disable_msix(adapter->pdev);  
+ adapter->flags &= ~USING_MSIX;  
+ } else if (adapter->flags & USING_MSI) {  
+ pci_disable_msi(adapter->pdev);  
+ adapter->flags &= ~USING_MSI;  
+ }  
+}  
+  
+/**  
+ * Interrupt handler for asynchronous events used with MSI-X.  
+ */
```

[PATCH 2/10] cxgb3 – main source file

```
+static irqreturn_t t3_async_intr_handler(int irq, void *cookie)
+{
+ t3_slow_intr_handler(cookie);
+ return IRQ_HANDLED;
+}
+
+/*
+ * Name the MSI-X interrupts.
+ */
+static void name_msix_vecs(adapter_t *adap)
+{
+ int i, n = sizeof(adap->msix_info[0].desc) - 1;
+ int nqs0 = adap->port[0].nqs0;
+
+ snprintf(adap->msix_info[0].desc, n, "%s", adap->name);
+ adap->msix_info[0].desc[n] = 0;
+
+ for (i = 1; i <= nqs0 + adap->port[1].nqs0; i++) {
+ snprintf(adap->msix_info[i].desc, n, "%s (queue %d)",
+ port_name(adap, i > nqs0),
+ i <= nqs0 ? i - 1 : i - 1 - nqs0);
+ adap->msix_info[i].desc[n] = 0;
+ }
+}
+
+static int request_msix_data_irqs(adapter_t *adap)
+{
+ int i, j, err, qidx = 0;
+
+ for_each_port(adap, i) {
+ int nqs0 = adap->port[i].nqs0;
+
+ for (j = 0; j < nqs0; ++j) {
+ err = request_irq(adap->msix_info[qidx + 1].vec,
+ t3_intr_handler(adap,
+ adap->sge.qs[qidx].rspq.polling),
+ 0, adap->msix_info[qidx + 1].desc,
+ &adap->sge.qs[qidx]);
+ if (err) {
+ while (--qidx >= 0)
+ free_irq(adap->msix_info[qidx + 1].vec,
+ &adap->sge.qs[qidx]);
+ return err;
+ }
+ qidx++;
+ }
+ }
+ return 0;
+}
+
+/**
```

[PATCH 2/10] cxgb3 – main source file

```
+ * setup_rss – configure RSS
+ * @adap: the adapter
+ *
+ * Sets up RSS to distribute packets to multiple receive queues. We
+ * configure the RSS CPU lookup table to distribute to the number of HW
+ * receive queues, and the response queue lookup table to narrow that
+ * down to the response queues actually configured for each port.
+ * We always configure the RSS mapping for two ports since the mapping
+ * table has plenty of entries.
+ */
+static void setup_rss(adapter_t *adap)
+{
+ int i;
+ unsigned int nq0 = adap->port[0].nqsets;
+ unsigned int nq1 = max((unsigned int)adap->port[1].nqsets, 1U);
+ u8 cpus[SGE_QSETS + 1];
+ u16 rspq_map[RSS_TABLE_SIZE];
+
+ for (i = 0; i < SGE_QSETS; ++i)
+ cpus[i] = i;
+ cpus[SGE_QSETS] = 0xff; /* terminator */
+
+ for (i = 0; i < RSS_TABLE_SIZE / 2; ++i) {
+ rspq_map[i] = i % nq0;
+ rspq_map[i + RSS_TABLE_SIZE / 2] = (i % nq1) + nq0;
+ }
+
+ t3_config_rss(adap, F_RQFEEDBACKENABLE | F_TNLLKPEN | F_TNLMAPEN |
+ F_TNLPRTEN | F_TNL2TUPEN | F_TNL4TUPEN |
+ V_RRCPLCPUSIZE(6) , cpus, rspq_map);
+}
+
+/*
+ * If we have multiple receive queues per port serviced by NAPI we need one
+ * netdevice per queue as NAPI operates on netdevices. We already have one
+ * netdevice, namely the one associated with the interface, so we use dummy
+ * ones for any additional queues. Note that these netdevices exist purely
+ * so that NAPI has something to work with, they do not represent network
+ * ports and are not registered.
+ */
+static int init_dummy_netdevs(adapter_t *adap)
+{
+ int i, j, dummy_idx = 0;
+ struct net_device *nd;
+
+ for_each_port(adap, i) {
+ const struct port_info *pi = &adap->port[i];
+
+ for (j = 0; j < pi->nqsets - 1; j++) {
+ if (!adap->dummy_netdev[dummy_idx]) {
+ nd = alloc_netdev(0, "", ether_setup);
```

[PATCH 2/10] cxgb3 – main source file

```
+ if (!nd)
+ goto free_all;
+
+ nd->priv = adap;
+ nd->weight = 64;
+ set_bit(__LINK_STATE_START, &nd->state);
+ adap->dummy_netdev[dummy_idx] = nd;
+ }
+ strcpy(adap->dummy_netdev[dummy_idx]->name,
+ pi->dev->name);
+ dummy_idx++;
+ }
+ }
+ return 0;
+
+free_all:
+ while (--dummy_idx >= 0) {
+ free_netdev(adap->dummy_netdev[dummy_idx]);
+ adap->dummy_netdev[dummy_idx] = NULL;
+ }
+ return -ENOMEM;
+}
+
+/*
+ * Wait until all NAPI handlers are descheduled. This includes the handlers of
+ * both netdevices representing interfaces and the dummy ones for the extra
+ * queues.
+ */
+static void quiesce_rx(adapter_t *adap)
+{
+ int i;
+ struct net_device *dev;
+
+ for_each_port(adap, i) {
+ dev = adap->port[i].dev;
+ while (test_bit(__LINK_STATE_RX_SCHED, &dev->state))
+ msleep(1);
+ }
+
+ for (i = 0; i < ARRAY_SIZE(adap->dummy_netdev); i++) {
+ dev = adap->dummy_netdev[i];
+ if (dev)
+ while (test_bit(__LINK_STATE_RX_SCHED, &dev->state))
+ msleep(1);
+ }
+ }
+
+/**
+ * setup_sge_qsets – configure SGE Tx/Rx/response queues
+ * @adap: the adapter
+ */
```

[PATCH 2/10] cxgb3 – main source file

```
+ * Determines how many sets of SGE queues to use and initializes them.
+ * We support multiple queue sets per port if we have MSI-X, otherwise
+ * just one queue set per port.
+ */
+static int setup_sge_qsets(adapter_t *adap)
+{
+ int i, j, err, irq_idx = 0, qset_idx = 0, dummy_dev_idx = 0;
+ unsigned int ntxq = is_offload(adap) ? SGE_TXQ_PER_SET : 1;
+
+ if (adap->params.rev > 0 && !(adap->flags & USING_MSI))
+ irq_idx = -1;
+
+ for_each_port(adap, i) {
+ const struct port_info *pi = &adap->port[i];
+
+ for (j = 0; j < pi->nqsets; ++j, ++qset_idx) {
+ err = t3_sge_alloc_qset(adap, qset_idx, 1,
+ (adap->flags & USING_MSIX) ? qset_idx + 1 : irq_idx,
+ &adap->params.sge.qset[qset_idx], ntxq,
+ j == 0 ? pi->dev :
+ adap->dummy_netdev[dummy_dev_idx++]);
+ if (err) {
+ t3_free_sge_resources(adap);
+ return err;
+ }
+ }
+ }
+
+ return 0;
+}
+
+static ssize_t attr_show(struct class_device *cd, char *buf,
+ ssize_t (*format)(struct adapter *, char *))
+{
+ ssize_t len;
+ struct adapter *adap = to_net_dev(cd)->priv;
+
+ /* Synchronize with ioctls that may shut down the device */
+ rtnl_lock();
+ len = (*format)(adap, buf);
+ rtnl_unlock();
+ return len;
+}
+
+static ssize_t attr_store(struct class_device *cd, const char *buf, size_t len,
+ ssize_t (*set)(struct adapter *, unsigned int),
+ unsigned int min_val, unsigned int max_val)
+{
+ char *endp;
+ ssize_t ret;
+ unsigned int val;
```

[PATCH 2/10] cxgb3 – main source file

```
+ struct adapter *adap = to_net_dev(cd)->priv;
+
+ if (!capable(CAP_NET_ADMIN))
+ return -EPERM;
+
+ val = simple_strtoul(buf, &endp, 0);
+ if (endp == buf || val < min_val || val > max_val)
+ return -EINVAL;
+
+ rtnl_lock();
+ ret = (*set)(adap, val);
+ if (!ret)
+ ret = len;
+ rtnl_unlock();
+ return ret;
+}
+
+#define CXGB3_SHOW(name, val_expr) \
+static ssize_t format_##name(struct adapter *adap, char *buf) \
+{ \
+ return sprintf(buf, "%u\n", val_expr); \
+} \
+static ssize_t show_##name(struct class_device *cd, char *buf) \
+{ \
+ return attr_show(cd, buf, format_##name); \
+}
+
+static ssize_t set_nfilters(struct adapter *adap, unsigned int val)
+{
+ if (adap->flags & FULL_INIT_DONE)
+ return -EBUSY;
+ if (val && adap->params.rev == 0)
+ return -EINVAL;
+ if (val > t3_mc5_size(&adap->mc5) - adap->params.mc5.nservers)
+ return -EINVAL;
+ adap->params.mc5.nfilters = val;
+ return 0;
+}
+
+static ssize_t store_nfilters(struct class_device *cd, const char *buf,
+ size_t len)
+{
+ return attr_store(cd, buf, len, set_nfilters, 0, ~0);
+}
+
+static ssize_t set_nservers(struct adapter *adap, unsigned int val)
+{
+ if (adap->flags & FULL_INIT_DONE)
+ return -EBUSY;
+ if (val > t3_mc5_size(&adap->mc5) - adap->params.mc5.nfilters)
+ return -EINVAL;
```

[PATCH 2/10] cxgb3 – main source file

```
+ adap->params.mc5.nservers = val;
+ return 0;
+}
+
+static ssize_t store_nservers(struct class_device *cd, const char *buf,
+ size_t len)
+{
+ return attr_store(cd, buf, len, set_nservers, 0, ~0);
+}
+
+#define CXGB3_ATTR_R(name, val_expr) \
+CXGB3_SHOW(name, val_expr) \
+static CLASS_DEVICE_ATTR(name, S_IRUGO, show_##name, NULL)
+
+#define CXGB3_ATTR_RW(name, val_expr, store_method) \
+CXGB3_SHOW(name, val_expr) \
+static CLASS_DEVICE_ATTR(name, S_IRUGO | S_IWUSR, show_##name, store_method)
+
+CXGB3_ATTR_R(cam_size, t3_mc5_size(&adap->mc5));
+CXGB3_ATTR_RW(nfilters, adap->params.mc5.nfilters, store_nfilters);
+CXGB3_ATTR_RW(nservers, adap->params.mc5.nservers, store_nservers);
+
+static struct attribute *cxgb3_attrs[] = {
+ &class_device_attr_cam_size.attr,
+ &class_device_attr_nfilters.attr,
+ &class_device_attr_nservers.attr,
+ NULL
+};
+
+static struct attribute_group cxgb3_attr_group = { .attrs = cxgb3_attrs };
+
+static ssize_t tm_attr_show(struct class_device *cd, char *buf, int sched)
+{
+ ssize_t len;
+ unsigned int v, addr, bpt, cpt;
+ struct adapter *adap = to_net_dev(cd)->priv;
+
+ addr = A_TP_TX_MOD_Q1_Q0_RATE_LIMIT - sched / 2;
+ rtnl_lock();
+ t3_write_reg(adap, A_TP_TM_PIO_ADDR, addr);
+ v = t3_read_reg(adap, A_TP_TM_PIO_DATA);
+ if (sched & 1)
+ v >>= 16;
+ bpt = (v >> 8) & 0xff;
+ cpt = v & 0xff;
+ if (!cpt)
+ len = sprintf(buf, "disabled\n");
+ else {
+ v = (adap->params.vpd.cclk * 1000) / cpt;
+ len = sprintf(buf, "%u Kbps\n", (v * bpt) / 125);
+ }
+}
```

[PATCH 2/10] cxgb3 – main source file

```
+ rtnl_unlock();
+ return len;
+}
+
+static ssize_t tm_attr_store(struct class_device *cd, const char *buf,
+ size_t len, int sched)
+{
+ char *endp;
+ ssize_t ret;
+ unsigned int val;
+ struct adapter *adap = to_net_dev(cd)->priv;
+
+ if (!capable(CAP_NET_ADMIN))
+ return -EPERM;
+
+ val = simple_strtoul(buf, &endp, 0);
+ if (endp == buf || val > 10000000)
+ return -EINVAL;
+
+ rtnl_lock();
+ ret = t3_config_sched(adap, val, sched);
+ if (!ret)
+ ret = len;
+ rtnl_unlock();
+ return ret;
+}
+
+#define TM_ATTR(name, sched) \
+static ssize_t show_##name(struct class_device *cd, char *buf) \
+{ \
+ return tm_attr_show(cd, buf, sched); \
+} \
+static ssize_t store_##name(struct class_device *cd, const char *buf, size_t len) \
+{ \
+ return tm_attr_store(cd, buf, len, sched); \
+} \
+static CLASS_DEVICE_ATTR(name, S_IRUGO | S_IWUSR, show_##name, store_##name)
+
+TM_ATTR(sched0, 0);
+TM_ATTR(sched1, 1);
+TM_ATTR(sched2, 2);
+TM_ATTR(sched3, 3);
+TM_ATTR(sched4, 4);
+TM_ATTR(sched5, 5);
+TM_ATTR(sched6, 6);
+TM_ATTR(sched7, 7);
+
+static struct attribute *offload_attrs[] = {
+ &class_device_attr_sched0.attr,
+ &class_device_attr_sched1.attr,
+ &class_device_attr_sched2.attr,
```

[PATCH 2/10] cxgb3 – main source file

```
+ &class_device_attr_sched3.attr,
+ &class_device_attr_sched4.attr,
+ &class_device_attr_sched5.attr,
+ &class_device_attr_sched6.attr,
+ &class_device_attr_sched7.attr,
+ NULL
+};
+
+static struct attribute_group offload_attr_group = { .attrs = offload_attrs };
+
+/*
+ * Sends an sk_buff to an offload queue driver
+ * after dealing with any active network taps.
+ */
+static inline int offload_tx(struct t3cdev *tdev, struct sk_buff *skb)
+{
+ int ret;
+
+ local_bh_disable();
+ ret = t3_offload_tx(tdev, skb);
+ local_bh_enable();
+ return ret;
+}
+
+static int write_smt_entry(struct adapter *adapter, int idx)
+{
+ struct net_device *dev = adapter->port[idx].dev;
+ struct cpl_smt_write_req *req;
+ struct sk_buff *skb = alloc_skb(sizeof(*req), GFP_KERNEL);
+
+ if (!skb) return -ENOMEM;
+
+ req = (struct cpl_smt_write_req *)__skb_put(skb, sizeof(*req));
+ req->wr.wr_hi = htonl(V_WR_OP(FW_WROPCODE_FORWARD));
+ OPCODE_TID(req) = htonl(MK_OPCODE_TID(CPL_SMT_WRITE_REQ, idx));
+ req->mtu_idx = NMTUS - 1; /* should be 0 but there's a T3 bug */
+ req->ifff = idx;
+ memset(req->src_mac1, 0, sizeof(req->src_mac1));
+ memcpy(req->src_mac0, dev->dev_addr, ETH_ALEN);
+ skb->priority = 1;
+ offload_tx(&adapter->tdev, skb);
+ return 0;
+}
+
+static int init_smt(struct adapter *adapter)
+{
+ int i;
+
+ for_each_port(adapter, i)
+ write_smt_entry(adapter, i);
+ return 0;
+}
```

```

+}
+
+static void init_port_mtus(adapter_t *adapter)
+{
+ unsigned int mtus = adapter->port[0].dev->mtu;
+
+ if (adapter->port[1].dev)
+ mtus |= adapter->port[1].dev->mtu << 16;
+ t3_write_reg(adapter, A_TP_MTU_PORT_TABLE, mtus);
+}
+
+/**
+ * cxgb_up – enable the adapter
+ * @adapter: adapter being enabled
+ *
+ * Called when the first port is enabled, this function performs the
+ * actions necessary to make an adapter operational, such as completing
+ * the initialization of HW modules, and enabling interrupts.
+ *
+ * Must be called with the rtnl lock held.
+ */
+static int cxgb_up(struct adapter *adap)
+{
+ int err = 0;
+
+ if (!(adap->flags & FULL_INIT_DONE)) {
+ err = t3_check_fw_version(adap);
+ if (err) {
+ dev_err(&adap->pdev->dev,
+ "adapter FW is not compatible with driver\n");
+ goto out;
+ }
+
+ err = init_dummy_netdevs(adap);
+ if (err)
+ goto out;
+
+ err = t3_init_hw(adap, 0);
+ if (err)
+ goto out;
+
+ err = setup_sge_qsets(adap);
+ if (err)
+ goto out;
+
+ setup_rss(adap);
+ adap->flags |= FULL_INIT_DONE;
+ }
+
+ t3_intr_clear(adap);
+
+

```

[PATCH 2/10] cxgb3 – main source file

```
+ if (adap->flags & USING_MSIX) {
+ name_msix_vecs(adap);
+ err = request_irq(adap->msix_info[0].vec,
+ t3_async_intr_handler, 0,
+ adap->msix_info[0].desc, adap);
+ if (err)
+ goto irq_err;
+
+ if (request_msix_data_irqs(adap)) {
+ free_irq(adap->msix_info[0].vec, adap);
+ goto irq_err;
+ }
+ } else if ((err = request_irq(adap->pdev->irq,
+ t3_intr_handler(adap,
+ adap->sge.qs[0].rspq.polling),
+ (adap->flags & USING_MSI) ? 0 : SA_SHIRQ,
+ adap->name, adap)))
+ goto irq_err;
+
+ t3_sge_start(adap);
+ t3_intr_enable(adap);
+out:
+ return err;
+irq_err:
+ CH_ERR("%s: request_irq failed, err %d\n", adapter_name(adap), err);
+ goto out;
+}
+
+/*
+ * Release resources when all the ports and offloading have been stopped.
+ */
+static void cxgb_down(struct adapter *adapter)
+{
+ t3_sge_stop(adapter);
+ spin_lock_irq(&adapter->work_lock); /* sync with PHY intr task */
+ t3_intr_disable(adapter);
+ spin_unlock_irq(&adapter->work_lock);
+
+ if (adapter->flags & USING_MSIX) {
+ int i, n = 0;
+
+ free_irq(adapter->msix_info[0].vec, adapter);
+ for_each_port(adapter, i)
+ n += adapter->port[i].nqsets;
+
+ for (i = 0; i < n; ++i)
+ free_irq(adapter->msix_info[i + 1].vec,
+ &adapter->sge.qs[i]);
+ } else
+ free_irq(adapter->pdev->irq, adapter);
+}
```

[PATCH 2/10] cxgb3 – main source file

```
+ flush_workqueue(cxgb3_wq); /* wait for external IRQ handler */
+ quiesce_rx(adapter);
+}
+
+static void schedule_chk_task(struct adapter *adap)
+{
+ unsigned int timeo;
+
+ timeo = adap->params.linkpoll_period ?
+ (HZ * adap->params.linkpoll_period) / 10 :
+ adap->params.stats_update_period * HZ;
+ if (timeo)
+ queue_delayed_work(cxgb3_wq, &adap->adap_check_task, timeo);
+}
+
+static int offload_open(struct net_device *dev)
+{
+ struct adapter *adapter = (struct adapter *)dev->priv;
+ struct t3cdev *tdev = T3CDEV(dev);
+ int adap_up = adapter->open_device_map & PORT_MASK;
+ int err = 0;
+
+ if (test_and_set_bit(OFFLOAD_DEVMAP_BIT, &adapter->open_device_map))
+ return 0;
+
+ if (!adap_up && (err = cxgb_up(adapter)) < 0)
+ return err;
+
+ t3_tp_set_offload_mode(adapter, 1);
+ tdev->lldev = adapter->port[0].dev;
+ err = cxgb3_offload_activate(adapter);
+ if (err)
+ goto out;
+
+ init_port_mtus(adapter);
+ t3_load_mtus(adapter, adap->params.mtus, adap->params.a_wnd,
+ adap->params.b_wnd,
+ adap->params.rev == 0 ?
+ adap->port[0].dev->mtu : 0xffff);
+ init_smt(adapter);
+
+ /* Never mind if the next step fails */
+ sysfs_create_group(&tdev->lldev->class_dev.kobj,
+ &offload_attr_group);
+
+ /* Call back all registered clients */
+ cxgb3_add_clients(tdev);
+
+out:
+ /* restore them in case the offload module has changed them */
+ if (err) {
```

[PATCH 2/10] cxgb3 – main source file

```
+ t3_tp_set_offload_mode(adapter, 0);
+ clear_bit(OFFLOAD_DEVMAP_BIT, &adapter->open_device_map);
+ cxgb3_set_dummy_ops(tdev);
+ }
+ return err;
+}
+
+static int offload_close(struct t3cdev *tdev)
+{
+ struct adapter *adapter = tdev2adap(tdev);
+
+ if (!test_bit(OFFLOAD_DEVMAP_BIT, &adapter->open_device_map))
+ return 0;
+
+ /* Call back all registered clients */
+ cxgb3_remove_clients(tdev);
+
+ sysfs_remove_group(&tdev->lldev->class_dev.kobj, &offload_attr_group);
+
+ tdev->lldev = NULL;
+ cxgb3_set_dummy_ops(tdev);
+ t3_tp_set_offload_mode(adapter, 0);
+ clear_bit(OFFLOAD_DEVMAP_BIT, &adapter->open_device_map);
+
+ if (!adapter->open_device_map)
+ cxgb_down(adapter);
+
+ cxgb3_offload_deactivate(adapter);
+ return 0;
+}
+
+static int cxgb_open(struct net_device *dev)
+{
+ int err;
+ struct adapter *adapter = dev->priv;
+ int other_ports = adapter->open_device_map & PORT_MASK;
+
+ if (!adapter->open_device_map && (err = cxgb_up(adapter)) < 0)
+ return err;
+
+ set_bit(dev->if_port, &adapter->open_device_map);
+
+ if (!ofld_disable) {
+ err = offload_open(dev);
+ if (err)
+ printk(KERN_WARNING
+ "Could not initialize offload capabilities\n");
+ }
+
+ link_start(&adapter->port[dev->if_port]);
+ t3_port_intr_enable(adapter, dev->if_port);
```

[PATCH 2/10] cxgb3 – main source file

```
+ netif_start_queue(dev);
+ if (!other_ports)
+ schedule_chk_task(adapter);
+
+ return 0;
+}
+
+static int cxgb_close(struct net_device *dev)
+{
+ struct adapter *adapter = dev->priv;
+ struct port_info *p = &adapter->port[dev->if_port];
+
+ t3_port_intr_disable(adapter, dev->if_port);
+ netif_stop_queue(dev);
+ p->phy.ops->power_down(&p->phy, 1);
+ netif_carrier_off(dev);
+ t3_mac_disable(&p->mac, MAC_DIRECTION_TX | MAC_DIRECTION_RX);
+
+ spin_lock(&adapter->work_lock); /* sync with update task */
+ clear_bit(dev->if_port, &adapter->open_device_map);
+ spin_unlock(&adapter->work_lock);
+
+ if (!(adapter->open_device_map & PORT_MASK))
+ cancel_rearming_delayed_workqueue(cxgb3_wq,
+ &adapter->adap_check_task);
+
+ if (!adapter->open_device_map)
+ cxgb_down(adapter);
+
+ return 0;
+}
+
+static struct net_device_stats *cxgb_get_stats(struct net_device *dev)
+{
+ struct adapter *adapter = dev->priv;
+ struct port_info *p = &adapter->port[dev->if_port];
+ struct net_device_stats *ns = &p->netstats;
+ const struct mac_stats *pstats;
+
+ spin_lock(&adapter->stats_lock);
+ pstats = t3_mac_update_stats(&p->mac);
+ spin_unlock(&adapter->stats_lock);
+
+ ns->tx_bytes = pstats->tx_octets;
+ ns->tx_packets = pstats->tx_frames;
+ ns->rx_bytes = pstats->rx_octets;
+ ns->rx_packets = pstats->rx_frames;
+ ns->multicast = pstats->rx_mcast_frames;
+
+ ns->tx_errors = pstats->tx_underrun;
+ ns->rx_errors = pstats->rx_symbol_errs + pstats->rx_fcs_errs +
```

[PATCH 2/10] cxgb3 – main source file

```
+ pstats->rx_too_long + pstats->rx_jabber + pstats->rx_short +
+ pstats->rx_fifo_ovfl;
+
+ /* detailed rx_errors */
+ ns->rx_length_errors = pstats->rx_jabber + pstats->rx_too_long;
+ ns->rx_over_errors = 0;
+ ns->rx_crc_errors = pstats->rx_fcs_errs;
+ ns->rx_frame_errors = pstats->rx_symbol_errs;
+ ns->rx_fifo_errors = pstats->rx_fifo_ovfl;
+ ns->rx_missed_errors = pstats->rx_cong_drops;
+
+ /* detailed tx_errors */
+ ns->tx_aborted_errors = 0;
+ ns->tx_carrier_errors = 0;
+ ns->tx_fifo_errors = pstats->tx_underrun;
+ ns->tx_heartbeat_errors = 0;
+ ns->tx_window_errors = 0;
+ return ns;
+}
+
+static u32 get_msglevel(struct net_device *dev)
+{
+ struct adapter *adapter = dev->priv;
+
+ return adapter->msg_enable;
+}
+
+static void set_msglevel(struct net_device *dev, u32 val)
+{
+ struct adapter *adapter = dev->priv;
+
+ adapter->msg_enable = val;
+}
+
+static char stats_strings[][ETH_GSTRING_LEN] = {
+ "TxOctetsOK ",
+ "TxFramesOK ",
+ "TxMulticastFramesOK",
+ "TxBroadcastFramesOK",
+ "TxPauseFrames ",
+ "TxUnderrun ",
+ "TxExtUnderrun ",
+
+ "TxFrames64 ",
+ "TxFrames65To127 ",
+ "TxFrames128To255 ",
+ "TxFrames256To511 ",
+ "TxFrames512To1023 ",
+ "TxFrames1024To1518 ",
+ "TxFrames1519ToMax ",
+
+ }
```


[PATCH 2/10] cxgb3 – main source file

```
+ t3_get_fw_version(adapter, &fw_vers);
+
+ strcpy(info->driver, DRV_NAME);
+ strcpy(info->version, DRV_VERSION);
+ strcpy(info->bus_info, pci_name(adapter->pdev));
+ if (!fw_vers)
+ strcpy(info->fw_version, "N/A");
+ else
+ snprintf(info->fw_version, sizeof(info->fw_version),
+ "%s %u.%u", (fw_vers >> 24) ? "T" : "N",
+ (fw_vers >> 12) & 0xfff, fw_vers & 0xfff);
+ }
+
+static void get_strings(struct net_device *dev, u32 stringset, u8 *data)
+{
+ if (stringset == ETH_SS_STATS)
+ memcpy(data, stats_strings, sizeof(stats_strings));
+ }
+
+static unsigned long collect_sge_port_stats(struct adapter *adapter, int port,
+ int idx)
+{
+ int i;
+ unsigned long tot = 0;
+ struct port_info *p = &adapter->port[port];
+
+ for (i = 0; i < p->nqsets; ++i)
+ tot += adapter->sge.qs[i + p->first_qset].port_stats[idx];
+ return tot;
+ }
+
+static void get_stats(struct net_device *dev, struct ethtool_stats *stats,
+ u64 *data)
+{
+ struct adapter *adapter = dev->priv;
+ int port_id = dev->if_port;
+ const struct mac_stats *s;
+
+ spin_lock(&adapter->stats_lock);
+ s = t3_mac_update_stats(&adapter->port[port_id].mac);
+ spin_unlock(&adapter->stats_lock);
+
+ *data++ = s->tx_octets;
+ *data++ = s->tx_frames;
+ *data++ = s->tx_mcast_frames;
+ *data++ = s->tx_bcast_frames;
+ *data++ = s->tx_pause;
+ *data++ = s->tx_underrun;
+ *data++ = s->tx_fifo_urun;
+
+ *data++ = s->tx_frames_64;
```

[PATCH 2/10] cxgb3 – main source file

```
+ *data++ = s->tx_frames_65_127;
+ *data++ = s->tx_frames_128_255;
+ *data++ = s->tx_frames_256_511;
+ *data++ = s->tx_frames_512_1023;
+ *data++ = s->tx_frames_1024_1518;
+ *data++ = s->tx_frames_1519_max;
+
+ *data++ = s->rx_octets;
+ *data++ = s->rx_frames;
+ *data++ = s->rx_mcast_frames;
+ *data++ = s->rx_bcast_frames;
+ *data++ = s->rx_pause;
+ *data++ = s->rx_fcs_errs;
+ *data++ = s->rx_symbol_errs;
+ *data++ = s->rx_short;
+ *data++ = s->rx_jabber;
+ *data++ = s->rx_too_long;
+ *data++ = s->rx_fifo_ovfl;
+
+ *data++ = s->rx_frames_64;
+ *data++ = s->rx_frames_65_127;
+ *data++ = s->rx_frames_128_255;
+ *data++ = s->rx_frames_256_511;
+ *data++ = s->rx_frames_512_1023;
+ *data++ = s->rx_frames_1024_1518;
+ *data++ = s->rx_frames_1519_max;
+
+ *data++ = adapter->port[port_id].phy.fifo_errors;
+
+ *data++ = collect_sge_port_stats(adapter, port_id, SGE_PSTAT_TSO);
+ *data++ = collect_sge_port_stats(adapter, port_id, SGE_PSTAT_VLANEX);
+ *data++ = collect_sge_port_stats(adapter, port_id, SGE_PSTAT_VLANINS);
+ *data++ = collect_sge_port_stats(adapter, port_id, SGE_PSTAT_TX_CSUM);
+ *data++ = collect_sge_port_stats(adapter, port_id,
+ SGE_PSTAT_RX_CSUM_GOOD);
+ *data++ = s->rx_cong_drops;
+}
+
+static inline void reg_block_dump(struct adapter *ap, void *buf,
+ unsigned int start, unsigned int end)
+{
+ u32 *p = buf + start;
+
+ for ( ; start <= end; start += sizeof(u32))
+ *p++ = t3_read_reg(ap, start);
+}
+
+static void get_regs(struct net_device *dev, struct ethtool_regs *regs,
+ void *buf)
+{
+ struct adapter *ap = dev->priv;
```

```

+
+ /*
+ * Version scheme:
+ * bits 0..9: chip version
+ * bits 10..15: chip revision
+ * bit 31: set for PCIe cards
+ */
+ regs->version = 3 | (ap->params.rev << 10) | (is_pcie(ap) << 31);
+
+ /*
+ * We skip the MAC statistics registers because they are clear-on-read.
+ * Also reading multi-register stats would need to synchronize with the
+ * periodic mac stats accumulation. Hard to justify the complexity.
+ */
+ memset(buf, 0, T3_REGMAP_SIZE);
+ reg_block_dump(ap, buf, 0, A_SG_RSPQ_CREDIT_RETURN);
+ reg_block_dump(ap, buf, A_SG_HI_DRB_HI_THRSH, A_ULPRX_PBL_ULIMIT);
+ reg_block_dump(ap, buf, A_ULPTX_CONFIG, A_MPS_INT_CAUSE);
+ reg_block_dump(ap, buf, A_CPL_SWITCH_CNTRL, A_CPL_MAP_TBL_DATA);
+ reg_block_dump(ap, buf, A_SMB_GLOBAL_TIME_CFG, A_XGM_SERDES_STAT3);
+ reg_block_dump(ap, buf, A_XGM_SERDES_STATUS0,
+ XGM_REG(A_XGM_SERDES_STAT3, 1));
+ reg_block_dump(ap, buf, XGM_REG(A_XGM_SERDES_STATUS0, 1),
+ XGM_REG(A_XGM_RX_SPI4_SOP_EOP_CNT, 1));
+ }
+
+static int restart_autoneg(struct net_device *dev)
+{
+ struct adapter *adapter = dev->priv;
+ struct port_info *p = &adapter->port[dev->if_port];
+
+ if (!netif_running(dev))
+ return -EAGAIN;
+ if (p->link_config.autoneg != AUTONEG_ENABLE)
+ return -EINVAL;
+ p->phy.ops->autoneg_restart(&p->phy);
+ return 0;
+}
+
+static int cxgb3_phys_id(struct net_device *dev, u32 data)
+{
+ int i;
+ struct adapter *adapter = dev->priv;
+
+ if (data == 0)
+ data = 2;
+
+ for (i = 0; i < data * 2; i++) {
+ t3_set_reg_field(adapter, A_T3DBG_GPIO_EN, F_GPIO0_OUT_VAL,
+ (i & 1) ? F_GPIO0_OUT_VAL : 0);
+ if (msleep_interruptible(500))

```

```

+ break;
+ }
+ t3_set_reg_field(adapter, A_T3DBG_GPIO_EN, F_GPIO0_OUT_VAL,
+ F_GPIO0_OUT_VAL);
+ return 0;
+ }
+
+static int get_settings(struct net_device *dev, struct ethtool_cmd *cmd)
+{
+ struct adapter *adapter = dev->priv;
+ struct port_info *p = &adapter->port[dev->if_port];
+
+ cmd->supported = p->link_config.supported;
+ cmd->advertising = p->link_config.advertising;
+
+ if (netif_carrier_ok(dev)) {
+ cmd->speed = p->link_config.speed;
+ cmd->duplex = p->link_config.duplex;
+ } else {
+ cmd->speed = -1;
+ cmd->duplex = -1;
+ }
+
+ cmd->port = (cmd->supported & SUPPORTED_TP) ? PORT_TP : PORT_FIBRE;
+ cmd->phy_address = p->phy.addr;
+ cmd->transceiver = XCVR_EXTERNAL;
+ cmd->autoneg = p->link_config.autoneg;
+ cmd->maxtxpkt = 0;
+ cmd->maxrxpkt = 0;
+ return 0;
+ }
+
+static int speed_duplex_to_caps(int speed, int duplex)
+{
+ int cap = 0;
+
+ switch (speed) {
+ case SPEED_10:
+ if (duplex == DUPLEX_FULL)
+ cap = SUPPORTED_10baseT_Full;
+ else
+ cap = SUPPORTED_10baseT_Half;
+ break;
+ case SPEED_100:
+ if (duplex == DUPLEX_FULL)
+ cap = SUPPORTED_100baseT_Full;
+ else
+ cap = SUPPORTED_100baseT_Half;
+ break;
+ case SPEED_1000:
+ if (duplex == DUPLEX_FULL)

```

[PATCH 2/10] cxgb3 – main source file

```
+ cap = SUPPORTED_1000baseT_Full;
+ else
+ cap = SUPPORTED_1000baseT_Half;
+ break;
+ case SPEED_10000:
+ if (duplex == DUPLEX_FULL)
+ cap = SUPPORTED_10000baseT_Full;
+ }
+ return cap;
+}
+
+#define ADVERTISED_MASK (ADVERTISED_10baseT_Half | ADVERTISED_10baseT_Full | \
+ ADVERTISED_100baseT_Half | ADVERTISED_100baseT_Full | \
+ ADVERTISED_1000baseT_Half | ADVERTISED_1000baseT_Full | \
+ ADVERTISED_10000baseT_Full)
+
+static int set_settings(struct net_device *dev, struct ethtool_cmd *cmd)
+{
+ struct adapter *adapter = dev->priv;
+ struct port_info *p = &adapter->port[dev->if_port];
+ struct link_config *lc = &p->link_config;
+
+ if (!(lc->supported & SUPPORTED_Autoneg))
+ return -EOPNOTSUPP; /* can't change speed/duplex */
+
+ if (cmd->autoneg == AUTONEG_DISABLE) {
+ int cap = speed_duplex_to_caps(cmd->speed, cmd->duplex);
+
+ if (!(lc->supported & cap) || cmd->speed == SPEED_1000)
+ return -EINVAL;
+ lc->requested_speed = cmd->speed;
+ lc->requested_duplex = cmd->duplex;
+ lc->advertising = 0;
+ } else {
+ cmd->advertising &= ADVERTISED_MASK;
+ cmd->advertising &= lc->supported;
+ if (!cmd->advertising)
+ return -EINVAL;
+ lc->requested_speed = SPEED_INVALID;
+ lc->requested_duplex = DUPLEX_INVALID;
+ lc->advertising = cmd->advertising | ADVERTISED_Autoneg;
+ }
+ lc->autoneg = cmd->autoneg;
+ if (netif_running(dev))
+ t3_link_start(&p->phy, &p->mac, lc);
+ return 0;
+}
+
+static void get_pauseparam(struct net_device *dev,
+ struct ethtool_pauseparam *epause)
+{
```

[PATCH 2/10] cxgb3 – main source file

```
+ struct adapter *adapter = dev->priv;
+ struct port_info *p = &adapter->port[dev->if_port];
+
+ epause->autoneg = (p->link_config.requested_fc & PAUSE_AUTONEG) != 0;
+ epause->rx_pause = (p->link_config.fc & PAUSE_RX) != 0;
+ epause->tx_pause = (p->link_config.fc & PAUSE_TX) != 0;
+}
+
+static int set_pauseparam(struct net_device *dev,
+ struct ethtool_pauseparam *epause)
+{
+ struct adapter *adapter = dev->priv;
+ struct port_info *p = &adapter->port[dev->if_port];
+ struct link_config *lc = &p->link_config;
+
+ if (epause->autoneg == AUTONEG_DISABLE)
+ lc->requested_fc = 0;
+ else if (lc->supported & SUPPORTED_Autoneg)
+ lc->requested_fc = PAUSE_AUTONEG;
+ else
+ return -EINVAL;
+
+ if (epause->rx_pause)
+ lc->requested_fc |= PAUSE_RX;
+ if (epause->tx_pause)
+ lc->requested_fc |= PAUSE_TX;
+ if (lc->autoneg == AUTONEG_ENABLE) {
+ if (netif_running(dev))
+ t3_link_start(&p->phy, &p->mac, lc);
+ } else {
+ lc->fc = lc->requested_fc & (PAUSE_RX | PAUSE_TX);
+ if (netif_running(dev))
+ t3_mac_set_speed_duplex_fc(&p->mac, -1, -1, lc->fc);
+ }
+ return 0;
+}
+
+static u32 get_rx_csum(struct net_device *dev)
+{
+ struct adapter *adapter = dev->priv;
+
+ return adapter->port[dev->if_port].rx_csum_offload;
+}
+
+static int set_rx_csum(struct net_device *dev, u32 data)
+{
+ struct adapter *adapter = dev->priv;
+
+ adapter->port[dev->if_port].rx_csum_offload = data;
+ return 0;
+}
```

[PATCH 2/10] cxgb3 – main source file

```
+
+static void get_sge_param(struct net_device *dev, struct ethtool_ringparam *e)
+{
+ struct adapter *adapter = dev->priv;
+
+ e->rx_max_pending = MAX_RX_BUFFERS;
+ e->rx_mini_max_pending = 0;
+ e->rx_jumbo_max_pending = MAX_RX_JUMBO_BUFFERS;
+ e->tx_max_pending = MAX_TXQ_ENTRIES;
+
+ e->rx_pending = adapter->params.sge.qset[0].fl_size;
+ e->rx_mini_pending = adapter->params.sge.qset[0].rspq_size;
+ e->rx_jumbo_pending = adapter->params.sge.qset[0].jumbo_size;
+ e->tx_pending = adapter->params.sge.qset[0].txq_size[0];
+}
+
+static int set_sge_param(struct net_device *dev, struct ethtool_ringparam *e)
+{
+ int i;
+ struct adapter *adapter = dev->priv;
+
+ if (e->rx_pending > MAX_RX_BUFFERS ||
+ e->rx_jumbo_pending > MAX_RX_JUMBO_BUFFERS ||
+ e->tx_pending > MAX_TXQ_ENTRIES ||
+ e->rx_mini_pending > MAX_RSPQ_ENTRIES ||
+ e->rx_mini_pending < MIN_RSPQ_ENTRIES ||
+ e->rx_pending < MIN_FL_ENTRIES ||
+ e->rx_jumbo_pending < MIN_FL_ENTRIES ||
+ e->tx_pending < adapter->params.nports * MIN_TXQ_ENTRIES)
+ return -EINVAL;
+
+ if (adapter->flags & FULL_INIT_DONE)
+ return -EBUSY;
+
+ for (i = 0; i < SGE_QSETS; ++i) {
+ struct qset_params *q = &adapter->params.sge.qset[i];
+
+ q->rspq_size = e->rx_mini_pending;
+ q->fl_size = e->rx_pending;
+ q->jumbo_size = e->rx_jumbo_pending;
+ q->txq_size[0] = e->tx_pending;
+ q->txq_size[1] = e->tx_pending;
+ q->txq_size[2] = e->tx_pending;
+ }
+ return 0;
+}
+
+static int set_coalesce(struct net_device *dev, struct ethtool_coalesce *c)
+{
+ struct adapter *adapter = dev->priv;
+ struct qset_params *qsp = &adapter->params.sge.qset[0];
```

[PATCH 2/10] cxgb3 – main source file

```
+ struct sge_qset *qs = &adapter->sge.qs[0];
+
+ if (c->rx_coalesce_usecs * 10 > M_NEWTIMER)
+ return -EINVAL;
+
+ qsp->coalesce_usecs = c->rx_coalesce_usecs;
+ t3_update_qset_coalesce(qs, qsp);
+ return 0;
+}
+
+static int get_coalesce(struct net_device *dev, struct ethtool_coalesce *c)
+{
+ struct adapter *adapter = dev->priv;
+ struct qset_params *q = adapter->params.sge.qset;
+
+ c->rx_coalesce_usecs = q->coalesce_usecs;
+ return 0;
+}
+
+static int get_eeprom(struct net_device *dev, struct ethtool_eeprom *e,
+ u8 *data)
+{
+ int i, err = 0;
+ struct adapter *adapter = dev->priv;
+
+ u8 *buf = kmalloc(EEPROMSIZE, GFP_KERNEL);
+ if (!buf)
+ return -ENOMEM;
+
+ e->magic = EEPROM_MAGIC;
+ for (i = e->offset & ~3; !err && i < e->offset + e->len; i += 4)
+ err = t3_seeprom_read(adapter, i, (u32 *)&buf[i]);
+
+ if (!err)
+ memcpy(data, buf + e->offset, e->len);
+ kfree(buf);
+ return err;
+}
+
+static int set_eeprom(struct net_device *dev, struct ethtool_eeprom *eeprom,
+ u8 *data)
+{
+ u8 *buf;
+ int err = 0;
+ u32 aligned_offset, aligned_len, *p;
+ struct adapter *adapter = dev->priv;
+
+ if (eeprom->magic != EEPROM_MAGIC)
+ return -EINVAL;
+
+ aligned_offset = eeprom->offset & ~3;
```

[PATCH 2/10] cxgb3 – main source file

```
+ aligned_len = (eeprom->len + (eeprom->offset & 3) + 3) & ~3;
+
+ if (aligned_offset != eeprom->offset || aligned_len != eeprom->len) {
+ buf = kmalloc(aligned_len, GFP_KERNEL);
+ if (!buf)
+ return -ENOMEM;
+ err = t3_seeprom_read(adapter, aligned_offset, (u32 *)buf);
+ if (!err && aligned_len > 4)
+ err = t3_seeprom_read(adapter,
+ aligned_offset + aligned_len - 4,
+ (u32 *)&buf[aligned_len - 4]);
+ if (err)
+ goto out;
+ memcpy(buf + (eeprom->offset & 3), data, eeprom->len);
+ } else
+ buf = data;
+
+ err = t3_seeprom_wp(adapter, 0);
+ if (err)
+ goto out;
+
+ for (p = (u32 *)buf; !err && aligned_len; aligned_len -= 4, p++) {
+ err = t3_seeprom_write(adapter, aligned_offset, *p);
+ aligned_offset += 4;
+ }
+
+ if (!err)
+ err = t3_seeprom_wp(adapter, 1);
+out:
+ if (buf != data)
+ kfree(buf);
+ return err;
+}
+
+static void get_wol(struct net_device *dev, struct ethtool_wolinfo *wol)
+{
+ wol->supported = 0;
+ wol->wolopts = 0;
+ memset(&wol->sopass, 0, sizeof(wol->sopass));
+}
+
+static struct ethtool_ops cxgb_ethtool_ops = {
+ .get_settings = get_settings,
+ .set_settings = set_settings,
+ .get_drvinfo = get_drvinfo,
+ .get_msglevel = get_msglevel,
+ .set_msglevel = set_msglevel,
+ .get_ringparam = get_sge_param,
+ .set_ringparam = set_sge_param,
+ .get_coalesce = get_coalesce,
+ .set_coalesce = set_coalesce,
```

[PATCH 2/10] cxgb3 – main source file

```
+ .get_eeprom_len = get_eeprom_len,
+ .get_eeprom = get_eeprom,
+ .set_eeprom = set_eeprom,
+ .get_pauseparam = get_pauseparam,
+ .set_pauseparam = set_pauseparam,
+ .get_rx_csum = get_rx_csum,
+ .set_rx_csum = set_rx_csum,
+ .get_tx_csum = ethtool_op_get_tx_csum,
+ .set_tx_csum = ethtool_op_set_tx_csum,
+ .get_sg = ethtool_op_get_sg,
+ .set_sg = ethtool_op_set_sg,
+ .get_link = ethtool_op_get_link,
+ .get_strings = get_strings,
+ .phys_id = cxgb3_phys_id,
+ .nway_reset = restart_autoneg,
+ .get_stats_count = get_stats_count,
+ .get_ethtool_stats = get_stats,
+ .get_regs_len = get_regs_len,
+ .get_regs = get_regs,
+ .get_wol = get_wol,
+ .get_tso = ethtool_op_get_tso,
+ .set_tso = ethtool_op_set_tso,
+#ifdef ETHTOOL_GPERMADDR
+ .get_perm_addr = ethtool_op_get_perm_addr
+#endif
+};
+
+static int in_range(int val, int lo, int hi)
+{
+ return val < 0 || (val <= hi && val >= lo);
+}
+
+static int cxgb_extension_ioctl(struct net_device *dev, void __user *useraddr)
+{
+ int ret;
+ u32 cmd;
+ struct adapter *adapter = dev->priv;
+
+ if (copy_from_user(&cmd, useraddr, sizeof(cmd)))
+ return -EFAULT;
+
+ switch (cmd) {
+ case CHELSIO_SETREG: {
+ struct ch_reg edata;
+
+ if (!capable(CAP_NET_ADMIN))
+ return -EPERM;
+ if (copy_from_user(&edata, useraddr, sizeof(edata)))
+ return -EFAULT;
+ if ((edata.addr & 3) != 0 || edata.addr >= adapter->mmio_len)
+ return -EINVAL;
```

```

+ writel(edata.val, adapter->regs + edata.addr);
+ break;
+ }
+ case CHELSIO_GETREG: {
+ struct ch_reg edata;
+
+ if (copy_from_user(&edata, useraddr, sizeof(edata)))
+ return -EFAULT;
+ if ((edata.addr & 3) != 0 || edata.addr >= adapter->mmio_len)
+ return -EINVAL;
+ edata.val = readl(adapter->regs + edata.addr);
+ if (copy_to_user(useraddr, &edata, sizeof(edata)))
+ return -EFAULT;
+ break;
+ }
+ case CHELSIO_SET_QSET_PARAMS: {
+ int i;
+ struct qset_params *q;
+ struct ch_qset_params t;
+
+ if (!capable(CAP_NET_ADMIN))
+ return -EPERM;
+ if (copy_from_user(&t, useraddr, sizeof(t)))
+ return -EFAULT;
+ if (t.qset_idx >= SGE_QSETS)
+ return -EINVAL;
+ if (!in_range(t.intr_lat, 0, M_NEWTIMER) ||
+ !in_range(t.cong_thres, 0, 255) ||
+ !in_range(t.txq_size[0], MIN_TXQ_ENTRIES,
+ MAX_TXQ_ENTRIES) ||
+ !in_range(t.txq_size[1], MIN_TXQ_ENTRIES,
+ MAX_TXQ_ENTRIES) ||
+ !in_range(t.txq_size[2], MIN_CTRL_TXQ_ENTRIES,
+ MAX_CTRL_TXQ_ENTRIES) ||
+ !in_range(t.fl_size[0], MIN_FL_ENTRIES, MAX_RX_BUFFERS) ||
+ !in_range(t.fl_size[1], MIN_FL_ENTRIES,
+ MAX_RX_JUMBO_BUFFERS) ||
+ !in_range(t.rspq_size, MIN_RSPQ_ENTRIES, MAX_RSPQ_ENTRIES))
+ return -EINVAL;
+ if ((adapter->flags & FULL_INIT_DONE) &&
+ (t.rspq_size >= 0 || t.fl_size[0] >= 0 ||
+ t.fl_size[1] >= 0 || t.txq_size[0] >= 0 ||
+ t.txq_size[1] >= 0 || t.txq_size[2] >= 0 ||
+ t.polling >= 0 || t.cong_thres >= 0))
+ return -EBUSY;
+
+ q = &adapter->params.sge.qset[t.qset_idx];
+
+ if (t.rspq_size >= 0)
+ q->rspq_size = t.rspq_size;
+ if (t.fl_size[0] >= 0)

```

```

+ q->fl_size = t.fl_size[0];
+ if (t.fl_size[1] >= 0)
+ q->jumbo_size = t.fl_size[1];
+ if (t.txq_size[0] >= 0)
+ q->txq_size[0] = t.txq_size[0];
+ if (t.txq_size[1] >= 0)
+ q->txq_size[1] = t.txq_size[1];
+ if (t.txq_size[2] >= 0)
+ q->txq_size[2] = t.txq_size[2];
+ if (t.cong_thres >= 0)
+ q->cong_thres = t.cong_thres;
+ if (t.intr_lat >= 0) {
+ struct sge_qset *qs = &adapter->sge.qs[t.qset_idx];
+
+ q->coalesce_usecs = t.intr_lat;
+ t3_update_qset_coalesce(qs, q);
+ }
+ if (t.polling >= 0) {
+ if (adapter->flags & USING_MSIX)
+ q->polling = t.polling;
+ else {
+ /* No polling with INTx for T3A */
+ if (adapter->params.rev == 0 &&
+ !(adapter->flags & USING_MSI))
+ t.polling = 0;
+
+ for (i = 0; i < SGE_QSETS; i++) {
+ q = &adapter->params.sge.qset[i];
+ q->polling = t.polling;
+ }
+ }
+ }
+ break;
+ }
+ case CHELSIO_GET_QSET_PARAMS: {
+ struct qset_params *q;
+ struct ch_qset_params t;
+
+ if (copy_from_user(&t, useraddr, sizeof(t)))
+ return -EFAULT;
+ if (t.qset_idx >= SGE_QSETS)
+ return -EINVAL;
+
+ q = &adapter->params.sge.qset[t.qset_idx];
+ t.rspq_size = q->rspq_size;
+ t.txq_size[0] = q->txq_size[0];
+ t.txq_size[1] = q->txq_size[1];
+ t.txq_size[2] = q->txq_size[2];
+ t.fl_size[0] = q->fl_size;
+ t.fl_size[1] = q->jumbo_size;
+ t.polling = q->polling;

```

```

+ t.intr_lat = q->coalesce_usecs;
+ t.cong_thres = q->cong_thres;
+
+ if (copy_to_user(useraddr, &t, sizeof(t)))
+ return -EFAULT;
+ break;
+ }
+ case CHELSIO_SET_QSET_NUM: {
+ struct ch_reg edata;
+ unsigned int port_idx = dev->if_port;
+
+ if (!capable(CAP_NET_ADMIN))
+ return -EPERM;
+ if (adapter->flags & FULL_INIT_DONE)
+ return -EBUSY;
+ if (copy_from_user(&edata, useraddr, sizeof(edata)))
+ return -EFAULT;
+ if (edata.val < 1 ||
+ (edata.val > 1 && !(adapter->flags & USING_MSIX)))
+ return -EINVAL;
+ if (edata.val + adapter->port[port_idx].nqsets > SGE_QSETS)
+ return -EINVAL;
+ adapter->port[port_idx].nqsets = edata.val;
+ adapter->port[1].first_qset = adapter->port[0].nqsets;
+ break;
+ }
+ case CHELSIO_GET_QSET_NUM: {
+ struct ch_reg edata;
+
+ edata.cmd = CHELSIO_GET_QSET_NUM;
+ edata.val = adapter->port[dev->if_port].nqsets;
+ if (copy_to_user(useraddr, &edata, sizeof(edata)))
+ return -EFAULT;
+ break;
+ }
+ case CHELSIO_LOAD_FW: {
+ u8 *fw_data;
+ struct ch_mem_range t;
+
+ if (!capable(CAP_NET_ADMIN))
+ return -EPERM;
+ if (copy_from_user(&t, useraddr, sizeof(t)))
+ return -EFAULT;
+
+ fw_data = kmalloc(t.len, GFP_KERNEL);
+ if (!fw_data)
+ return -ENOMEM;
+
+ if (copy_from_user(fw_data, useraddr + sizeof(t), t.len)) {
+ kfree(fw_data);
+ return -EFAULT;

```

```

+ }
+
+ ret = t3_load_fw(adapter, fw_data, t.len);
+ kfree(fw_data);
+ if (ret)
+ return ret;
+ break;
+ }
+ case CHELSIO_SETMTUTAB: {
+ struct ch_mtus m;
+ int i;
+
+ if (!is_offload(adapter))
+ return -EOPNOTSUPP;
+ if (!capable(CAP_NET_ADMIN))
+ return -EPERM;
+ if (offload_running(adapter))
+ return -EBUSY;
+ if (copy_from_user(&m, useraddr, sizeof(m)))
+ return -EFAULT;
+ if (m.nmtus != NMTUS)
+ return -EINVAL;
+ if (m.mtus[0] < 81) /* accommodate SACK */
+ return -EINVAL;
+
+ // MTUs must be in ascending order
+ for (i = 1; i < NMTUS; ++i)
+ if (m.mtus[i] < m.mtus[i - 1])
+ return -EINVAL;
+
+ memcpy(adapter->params.mtus, m.mtus,
+ sizeof(adapter->params.mtus));
+ break;
+ }
+ case CHELSIO_GET_PM: {
+ struct tp_params *p = &adapter->params.tp;
+ struct ch_pm m = { .cmd = CHELSIO_GET_PM };
+
+ if (!is_offload(adapter))
+ return -EOPNOTSUPP;
+ m.tx_pg_sz = p->tx_pg_size;
+ m.tx_num_pg = p->tx_num_pgs;
+ m.rx_pg_sz = p->rx_pg_size;
+ m.rx_num_pg = p->rx_num_pgs;
+ m.pm_total = p->pmtx_size + p->chan_rx_size * p->nchan;
+ if (copy_to_user(useraddr, &m, sizeof(m)))
+ return -EFAULT;
+ break;
+ }
+ case CHELSIO_SET_PM: {
+ struct ch_pm m;

```

```

+ struct tp_params *p = &adapter->params.tp;
+
+ if (!lis_offload(adapter))
+ return -EOPNOTSUPP;
+ if (!capable(CAP_NET_ADMIN))
+ return -EPERM;
+ if (adapter->flags & FULL_INIT_DONE)
+ return -EBUSY;
+ if (copy_from_user(&m, useraddr, sizeof(m)))
+ return -EFAULT;
+ if (!m.rx_pg_sz || (m.rx_pg_sz & (m.rx_pg_sz - 1)) ||
+ !m.tx_pg_sz || (m.tx_pg_sz & (m.tx_pg_sz - 1)))
+ return -EINVAL; /* not power of 2 */
+ if (!(m.rx_pg_sz & 0x14000))
+ return -EINVAL; /* not 16KB or 64KB */
+ if (!(m.tx_pg_sz & 0x1554000))
+ return -EINVAL;
+ if (m.tx_num_pg == -1)
+ m.tx_num_pg = p->tx_num_pgs;
+ if (m.rx_num_pg == -1)
+ m.rx_num_pg = p->rx_num_pgs;
+ if (m.tx_num_pg % 24 || m.rx_num_pg % 24)
+ return -EINVAL;
+ if (m.rx_num_pg * m.rx_pg_sz > p->chan_rx_size ||
+ m.tx_num_pg * m.tx_pg_sz > p->chan_tx_size)
+ return -EINVAL;
+ p->rx_pg_size = m.rx_pg_sz;
+ p->tx_pg_size = m.tx_pg_sz;
+ p->rx_num_pgs = m.rx_num_pg;
+ p->tx_num_pgs = m.tx_num_pg;
+ break;
+ }
+ case CHELSIO_GET_MEM: {
+ struct ch_mem_range t;
+ struct mc7 *mem;
+ u64 buf[32];
+
+ if (!lis_offload(adapter))
+ return -EOPNOTSUPP;
+ if (!(adapter->flags & FULL_INIT_DONE))
+ return -EIO; /* need the memory controllers */
+ if (copy_from_user(&t, useraddr, sizeof(t)))
+ return -EFAULT;
+ if ((t.addr & 7) || (t.len & 7))
+ return -EINVAL;
+ if (t.mem_id == MEM_CM)
+ mem = &adapter->cm;
+ else if (t.mem_id == MEM_PMRX)
+ mem = &adapter->pmtx;
+ else if (t.mem_id == MEM_PMTX)
+ mem = &adapter->pmtx;

```

```

+ else
+ return -EINVAL;
+
+ /*
+ * Version scheme:
+ * bits 0..9: chip version
+ * bits 10..15: chip revision
+ */
+ t.version = 3 | (adapter->params.rev << 10);
+ if (copy_to_user(useraddr, &t, sizeof(t)))
+ return -EFAULT;
+
+ /*
+ * Read 256 bytes at a time as len can be large and we don't
+ * want to use huge intermediate buffers.
+ */
+ useraddr += sizeof(t); /* advance to start of buffer */
+ while (t.len) {
+ unsigned int chunk = min_t(unsigned int, t.len, sizeof(buf));
+
+ ret = t3_mc7_bd_read(mem, t.addr / 8, chunk / 8, buf);
+ if (ret)
+ return ret;
+ if (copy_to_user(useraddr, buf, chunk))
+ return -EFAULT;
+ useraddr += chunk;
+ t.addr += chunk;
+ t.len -= chunk;
+ }
+ break;
+ }
+ case CHELSIO_SET_TRACE_FILTER: {
+ struct ch_trace t;
+ const struct trace_params *tp;
+
+ if (!capable(CAP_NET_ADMIN))
+ return -EPERM;
+ if (!offload_running(adapter))
+ return -EAGAIN;
+ if (copy_from_user(&t, useraddr, sizeof(t)))
+ return -EFAULT;
+
+ tp = (const struct trace_params *)&t.sip;
+ if (t.config_tx)
+ t3_config_trace_filter(adapter, tp, 0, t.invert_match,
+ t.trace_tx);
+ if (t.config_rx)
+ t3_config_trace_filter(adapter, tp, 1, t.invert_match,
+ t.trace_rx);
+ break;
+ }

```

[PATCH 2/10] cxgb3 – main source file

```
+ case CHELSIO_SET_PKTSCHEDED: {
+ struct sk_buff *skb;
+ struct ch_pktsched_params p;
+ struct mngt_pktsched_wr *req;
+
+ if (!(adapter->flags & FULL_INIT_DONE))
+ return -EIO; /* uP must be up and running */
+ if (copy_from_user(&p, useraddr, sizeof(p)))
+ return -EFAULT;
+ skb = alloc_skb(sizeof(*req), GFP_KERNEL);
+ if (!skb)
+ return -ENOMEM;
+ req = (struct mngt_pktsched_wr *)skb_put(skb, sizeof(*req));
+ req->wr_hi = htonl(V_WR_OP(FW_WROPCODE_MNGT));
+ req->mngt_opcode = FW_MNGTOPCODE_PKTSCHEDED_SET;
+ req->sched = p.sched;
+ req->idx = p.idx;
+ req->min = p.min;
+ req->max = p.max;
+ req->binding = p.binding;
+ printk(KERN_INFO
+ "pktsched: sched %u idx %u min %u max %u binding %u\n",
+ req->sched, req->idx, req->min, req->max, req->binding);
+ skb->priority = 1;
+ offload_tx(&adapter->tdev, skb);
+ break;
+ }
+ default:
+ return -EOPNOTSUPP;
+ }
+ return 0;
+}
+
+static int cxgb_ioctl(struct net_device *dev, struct ifreq *req, int cmd)
+{
+ int ret, mmd;
+ struct adapter *adapter = dev->priv;
+ struct mii_ioctl_data *data = if_mii(req);
+
+ switch (cmd) {
+ case SIOCGMIIPHY:
+ data->phy_id = adapter->port[dev->if_port].phy.addr;
+ /* FALLTHRU */
+ case SIOCGMIIREG: {
+ u32 val;
+ struct cphy *phy = &adapter->port[dev->if_port].phy;
+
+ if (!phy->mdio_read)
+ return -EOPNOTSUPP;
+ if (is_10G(adapter)) {
+ mmd = data->phy_id >> 8;

```

```

+ if (!mmd)
+ mmd = MDIO_DEV_PCS;
+ else if (mmd > MDIO_DEV_XGXS)
+ return -EINVAL;
+
+ ret = phy->mdio_read(adapter, data->phy_id & 0x1f, mmd,
+ data->reg_num, &val);
+ } else
+ ret = phy->mdio_read(adapter, data->phy_id & 0x1f, 0,
+ data->reg_num & 0x1f, &val);
+ if (!ret)
+ data->val_out = val;
+ break;
+ }
+ case SIOCSMIIREG: {
+ struct cphy *phy = &adapter->port[dev->if_port].phy;
+
+ if (!capable(CAP_NET_ADMIN))
+ return -EPERM;
+ if (!phy->mdio_write)
+ return -EOPNOTSUPP;
+ if (is_10G(adapter)) {
+ mmd = data->phy_id >> 8;
+ if (!mmd)
+ mmd = MDIO_DEV_PCS;
+ else if (mmd > MDIO_DEV_XGXS)
+ return -EINVAL;
+
+ ret = phy->mdio_write(adapter, data->phy_id & 0x1f,
+ mmd, data->reg_num, data->val_in);
+ } else
+ ret = phy->mdio_write(adapter, data->phy_id & 0x1f, 0,
+ data->reg_num & 0x1f,
+ data->val_in);
+ break;
+ }
+ case SIOCCHIOCTL:
+ return cxgb_extension_ioctl(dev, req->ifr_data);
+ default:
+ return -EOPNOTSUPP;
+ }
+ return ret;
+}
+
+static int cxgb_change_mtu(struct net_device *dev, int new_mtu)
+{
+ int ret;
+ struct adapter *adapter = dev->priv;
+
+ if (new_mtu < 81) /* accommodate SACK */
+ return -EINVAL;

```

[PATCH 2/10] cxgb3 – main source file

```
+ if ((ret = t3_mac_set_mtu(&adapter->port[dev->if_port].mac, new_mtu)))
+ return ret;
+ dev->mtu = new_mtu;
+ init_port_mtus(adapter);
+ if (adapter->params.rev == 0 && offload_running(adapter))
+ t3_load_mtus(adapter, adapter->params.mtus,
+ adapter->params.a_wnd, adapter->params.b_wnd,
+ adapter->port[0].dev->mtu);
+ return 0;
+}
+
+static int cxgb_set_mac_addr(struct net_device *dev, void *p)
+{
+ struct adapter *adapter = dev->priv;
+ struct sockaddr *addr = p;
+
+ if (!is_valid_ether_addr(addr->sa_data))
+ return -EINVAL;
+
+ memcpy(dev->dev_addr, addr->sa_data, dev->addr_len);
+ t3_mac_set_address(&adapter->port[dev->if_port].mac, 0, dev->dev_addr);
+ if (offload_running(adapter))
+ write_smt_entry(adapter, dev->if_port);
+ return 0;
+}
+
+/**
+ * t3_synchronize_rx – wait for current Rx processing on a port to complete
+ * @adap: the adapter
+ * @port: the port index
+ *
+ * Ensures that current Rx processing on any of the queues associated with
+ * the given port completes before returning. We do this by acquiring and
+ * releasing the locks of the response queues associated with the port.
+ */
+static void t3_synchronize_rx(struct adapter *adap, int port)
+{
+ int i;
+ const struct port_info *p = &adap->port[port];
+
+ for (i = 0; i < p->nqsets; i++) {
+ struct sge_rspq *q = &adap->sge.qs[i + p->first_qset].rspq;
+
+ spin_lock_irq(&q->lock);
+ spin_unlock_irq(&q->lock);
+ }
+}
+
+static void vlan_rx_register(struct net_device *dev, struct vlan_group *grp)
+{
+ unsigned int i, have_vlans, idx = dev->if_port;
```

```

+ struct adapter *adapter = dev->priv;
+
+ adapter->port[idx].vlan_grp = grp;
+ if (adapter->params.rev > 0)
+ t3_set_vlan_accel(adapter, 1 << idx, grp != NULL);
+ else {
+ /* single control for all ports */
+ have_vlans = 0;
+ for_each_port(adapter, i)
+ have_vlans |= adapter->port[i].vlan_grp != NULL;
+
+ t3_set_vlan_accel(adapter, 1, have_vlans);
+ }
+ t3_synchronize_rx(adapter, idx);
+}
+
+static void vlan_rx_kill_vid(struct net_device *dev, unsigned short vid)
+{
+ /* nothing */
+}
+
+#ifdef CONFIG_NET_POLL_CONTROLLER
+static void cxgb_netpoll(struct net_device *dev)
+{
+ unsigned long flags;
+ struct adapter *adapter = dev->priv;
+ struct sge_qset *qs = dev2qset(dev);
+
+ local_irq_save(flags);
+ t3_intr_handler(adapter, qs->rspq.polling)(adapter->pdev->irq,
+ adapter);
+ local_irq_restore(flags);
+}
+#endif
+
+/*
+ * Periodic accumulation of MAC statistics.
+ */
+static void mac_stats_update(struct adapter *adapter)
+{
+ int i;
+
+ for_each_port(adapter, i) {
+ struct port_info *p = &adapter->port[i];
+
+ if (netif_running(p->dev)) {
+ spin_lock(&adapter->stats_lock);
+ t3_mac_update_stats(&p->mac);
+ spin_unlock(&adapter->stats_lock);
+ }
+ }
+}

```

```
+}
+
+static void check_link_status(struct adapter *adapter)
+{
+ int i;
+
+ for_each_port(adapter, i) {
+ struct port_info *p = &adapter->port[i];
+
+ if (!(p->port_type->caps & SUPPORTED_IRQ) &&
+ netif_running(p->dev))
+ t3_link_changed(adapter, i);
+ }
+}
+
+static void t3_adap_check_task(void *data)
+{
+ struct adapter *adapter = data;
+ const struct adapter_params *p = &adapter->params;
+
+ adapter->check_task_cnt++;
+
+ /* Check link status for PHYs without interrupts */
+ if (p->linkpoll_period)
+ check_link_status(adapter);
+
+ /* Accumulate MAC stats if needed */
+ if (!p->linkpoll_period ||
+ (adapter->check
```