

# Driver for SURECOM EP-320X-S 100/10M Ethernet PCI Adapter

**Source:** <http://unix.derkeiler.com/Mailing-Lists/FreeBSD/net/2004-09/0275.html>

---

**From:** Thomas (*news\_at\_undercover.dk*)

**Date:** 09/24/04

To: <freebsd-net@freebsd.org>

Date: Fri, 24 Sep 2004 15:41:28 +0200

Hi All

I have a Surecom Ethernet Adapter where there's a FreeBSD driver available for download at the manufactors website. However I think it is for FreeBSD 4.x because it won't compile on my FreeBSD 5.2.1-release system. Is anybody up for converting the driver to 5.x, or does anybody know if there's an alternate driver for this NIC?

Thanks.

Driver consists of 2 files: if\_fet.c & if\_fetreg.h (see below):

if\_fet.c

---

```
/*
 * fast ethernet PCI NIC driver
 */
#include "bpfiler.h"
#include <sys/param.h>
#include <sys/system.h>
#include <sys/sockio.h>
#include <sys/mbuf.h>
#include <sys/malloc.h>
#include <sys/kernel.h>
#include <sys/socket.h>
#include <net/if.h>
#include <net/if_arp.h>
#include <net/ethernet.h>
#include <net/if_dl.h>
#include <net/if_media.h>
#if NBPFILTER > 0
#include <net/bpf.h>
#endif
#include <vm/vm.h> /* for vtophys */
```

## freebsd-net: Driver for SURECOM EP-320X-S 100/10M Ethernet PCI Adapter

```
#include <vm/pmap.h> /* for vtophys */
#include <machine/clock.h> /* for DELAY */
#include <machine/bus_memio.h>
#include <machine/bus_pio.h>
#include <machine/bus.h>
#include <pci/pci.h>
#include <pci/pciivar.h>
/*
#define FET_USEIOWSPACE
*/
static int FET_USEIOWSPACE=1;
#include <pci/if_fetreg.h>
#ifdef lint
static const char rcsid[] =
"$Id: if_fet.c,v 1.30 2000/12/29 09:28:52 wpaul Exp $";
#endif
/*
* Various supported device vendors/types and their names.
*/
struct fet_type *fet_info_tmp;
static struct fet_type fet_devs[] = {
{ FETVENDORID, ID0, "100/10M Ethernet PCI Adapter" },
{ FETVENDORID, ID1, "100/10M Ethernet PCI Adapter" },
{ FETVENDORID, ID2, "1000/100/10M Ethernet PCI Adapter" },
{ 0, 0, NULL }
};
/*
* Various supported PHY vendors/types and their names. Note that
* this driver will work with pretty much any MII-compliant PHY,
* so failure to positively identify the chip is not a fatal error.
*/
static struct fet_type fet_phys[] = {
{ MysonPHYID0, MysonPHYID0, "<MYSON MTD981>" },
{ SeeqPHYID0, SeeqPHYID0, "<SEEQ 80225>" },
{ AhdDocPHYID0, AhdDocPHYID0, "<AHD0C 101>" },
{ MarvellPHYID0, MarvellPHYID0, "<MARVELL 88E1000>" },
{ LevelOnePHYID0, LevelOnePHYID0, "<LevelOne LXT1000>" },
{ 0, 0, "<MII-compliant physical interface>" }
};
static unsigned long fet_count = 0;
static const char *fet_probe __P((pcici_t, pcidi_t));
static void fet_attach __P((pcici_t, int));
static int fet_newbuf __P((struct fet_softc *, struct fet_chain_onefrag *));
static int fet_encap __P((struct fet_softc *, struct fet_chain *,
struct mbuf *));
static void fet_rxeof __P((struct fet_softc *));
static void fet_txeof __P((struct fet_softc *));
static void fet_txeoc __P((struct fet_softc *));
static void fet_intr __P((void *));
static void fet_start __P((struct ifnet *));
static int fet_ioctl __P((struct ifnet *, u_long, caddr_t));
```

```

static void fet_init __P((void *));
static void fet_stop __P((struct fet_softc *));
static void fet_watchdog __P((struct ifnet *));
static void fet_shutdown __P((int, void *));
static int fet_ifmedia_upd __P((struct ifnet *));
static void fet_ifmedia_sts __P((struct ifnet *, struct ifmediareq *));
static u_int16_t fet_phy_readreg __P((struct fet_softc *, int));
static void fet_phy_writereg __P((struct fet_softc *, int, int));
static void fet_autoneg_xmit __P((struct fet_softc *));
static void fet_autoneg_mii __P((struct fet_softc *, int, int));
static void fet_setmode_mii __P((struct fet_softc *, int));
static void fet_getmode_mii __P((struct fet_softc *));
static void fet_setcfg __P((struct fet_softc *, int));
static u_int8_t fet_calchash __P((caddr_t));
static void fet_setmulti __P((struct fet_softc *));
static void fet_reset __P((struct fet_softc *));
static int fet_list_rx_init __P((struct fet_softc *));
static int fet_list_tx_init __P((struct fet_softc *));
static long fet_send_cmd_to_phy __P((struct fet_softc *, int, int));
#define FET_SETBIT(sc, reg, x) CSR_WRITE_4(sc, reg, CSR_READ_4(sc, reg) | x)
#define FET_CLRBIT(sc, reg, x) CSR_WRITE_4(sc, reg, CSR_READ_4(sc, reg) &
~x)

static long fet_send_cmd_to_phy(sc, opcode, regad)
struct fet_softc *sc;
int opcode;
int regad;
{
long miir;
int i;
int mask, data;
/* enable MII output */
miir=CSR_READ_4(sc, FET_MANAGEMENT);
miir&=0xfffff0;
miir|=FET_MASK_MIIR_MII_WRITE+FET_MASK_MIIR_MII_MDO;
/* send 32 1's preamble */
for (i=0;i<32;i++)
{
/* low MDC; MDO is already high (miir) */
miir&=~FET_MASK_MIIR_MII_MDC;
CSR_WRITE_4(sc, FET_MANAGEMENT, miir);
/* high MDC */
miir|=FET_MASK_MIIR_MII_MDC;
CSR_WRITE_4(sc, FET_MANAGEMENT, miir);
}
/* calculate ST+OP+PHYAD+REGAD+TA */
data=opcode|(sc->fet_phy_addr<<7)|(regad<<2);
/* sent out */
mask=0x8000;
while (mask)
{

```

freebsd-net: Driver for SURECOM EP-320X-S 100/10M Ethernet PCI Adapter

```
/* low MDC, prepare MDO */
miir&=~(FET_MASK_MIIR_MII_MDC+FET_MASK_MIIR_MII_MDO);
if (mask & data)
miir|=FET_MASK_MIIR_MII_MDO;
CSR_WRITE_4(sc, FET_MANAGEMENT, miir);
/* high MDC */
miir|=FET_MASK_MIIR_MII_MDC;
CSR_WRITE_4(sc, FET_MANAGEMENT, miir);
DELAY(30);
/* next */
mask>>=1;
if (mask==0x2 && opcode==FET_OP_READ)
miir&=~FET_MASK_MIIR_MII_WRITE;
}
return miir;
}

static u_int16_t fet_phy_readreg(sc, reg)
struct fet_softc *sc;
int reg;
{
long miir;
int mask, data;
if (sc->fet_info->fet_did == ID1)
data = CSR_READ_2(sc, FET_PHYBASE+reg*2);
else
{
miir=fet_send_cmd_to_phy(sc, FET_OP_READ, reg);
/* read data */
mask=0x8000;
data=0;
while (mask)
{
/* low MDC */
miir&=~FET_MASK_MIIR_MII_MDC;
CSR_WRITE_4(sc, FET_MANAGEMENT, miir);
/* read MDI */
miir=CSR_READ_4(sc, FET_MANAGEMENT);
if (miir & FET_MASK_MIIR_MII_MDI)
data|=mask;
/* high MDC, and wait */
miir|=FET_MASK_MIIR_MII_MDC;
CSR_WRITE_4(sc, FET_MANAGEMENT, miir);
DELAY(30);
/* next */
mask>>=1;
}
/* low MDC */
miir&=~FET_MASK_MIIR_MII_MDC;
CSR_WRITE_4(sc, FET_MANAGEMENT, miir);
}
}
```

```

return (u_int16_t)data;
}

static void fet_phy_writereg(sc, reg, data)
struct fet_softc *sc;
int reg;
int data;
{
long miir;
int mask;
if (sc->fet_info->fet_did == ID1)
CSR_WRITE_2(sc, FET_PHYBASE+reg*2, data);
else
{
miir=fet_send_cmd_to_phy(sc, FET_OP_WRITE, reg);
/* write data */
mask=0x8000;
while (mask)
{
/* low MDC, prepare MDO */
miir&=~(FET_MASK_MIIR_MII_MDC+FET_MASK_MIIR_MII_MDO);
if (mask&data)
miir|=FET_MASK_MIIR_MII_MDO;
CSR_WRITE_4(sc, FET_MANAGEMENT, miir);
DELAY(1);
/* high MDC */
miir|=FET_MASK_MIIR_MII_MDC;
CSR_WRITE_4(sc, FET_MANAGEMENT, miir);
DELAY(1);
/* next */
mask>>=1;
}
/* low MDC */
miir&=~FET_MASK_MIIR_MII_MDC;
CSR_WRITE_4(sc, FET_MANAGEMENT, miir);
}
return;
}

static u_int8_t fet_calchash(addr)
caddr_t addr;
{
u_int32_t crc, carry;
int i, j;
u_int8_t c;
/* Compute CRC for the address value. */
crc = 0xFFFFFFFF; /* initial value */
for (i = 0; i < 6; i++)
{
c = *(addr + i);
for (j = 0; j < 8; j++)

```

```

{
carry = ((crc & 0x80000000) ? 1 : 0) ^ (c & 0x01);
crc <<= 1;
c >>= 1;
if (carry)
crc = (crc ^ 0x04c11db6) | carry;
}
}
/*
* return the filter bit position
* Note: I arrived at the following nonsense
* through experimentation. It's not the usual way to
* generate the bit position but it's the only thing
* I could come up with that works.
*/
return(~(crc >> 26) & 0x0000003F);
}

/*
* Program the 64-bit multicast hash filter.
*/
static void fet_setmulti(sc)
struct fet_softc *sc;
{
struct ifnet *ifp;
int h = 0;
u_int32_t hashes[2] = { 0, 0 };
struct ifmultiaddr *ifma;
u_int32_t rxfilt;
int mcnt = 0;
ifp = &sc->arpcom.ac_if;
rxfilt = CSR_READ_4(sc, FET_TCRRCR);
if (ifp->if_flags & IFF_ALLMULTI || ifp->if_flags & IFF_PROMISC)
{
rxfilt |= FET_AM;
CSR_WRITE_4(sc, FET_TCRRCR, rxfilt);
CSR_WRITE_4(sc, FET_MAR0, 0xFFFFFFFF);
CSR_WRITE_4(sc, FET_MAR1, 0xFFFFFFFF);
return;
}
/* first, zot all the existing hash bits */
CSR_WRITE_4(sc, FET_MAR0, 0);
CSR_WRITE_4(sc, FET_MAR1, 0);
/* now program new ones */
for (ifma = ifp->if_multiaddrs.lh_first; ifma != NULL;
ifma = ifma->ifma_link.le_next)
{
if (ifma->ifma_addr->sa_family != AF_LINK)
continue;
h = fet_calchash(LLADDR((struct sockaddr_dl *)ifma->ifma_addr));
if (h < 32)

```

```

hashes[0] |= (1 << h);
else
hashes[1] |= (1 << (h - 32));
mcnt++;
}
if (mcnt)
rxfilt |= FET_AM;
else
rxfilt &= ~FET_AM;
CSR_WRITE_4(sc, FET_MAR0, hashes[0]);
CSR_WRITE_4(sc, FET_MAR1, hashes[1]);
CSR_WRITE_4(sc, FET_TCRRCR, rxfilt);
return;
}

/*
 * Initiate an autonegotiation session.
 */
static void fet_autoneg_xmit(sc)
struct fet_softc *sc;
{
u_int16_t phy_sts;
fet_phy_writereg(sc, PHY_BMCR, PHY_BMCR_RESET);
DELAY(500);
while(fet_phy_readreg(sc, PHY_BMCR) & PHY_BMCR_RESET);
phy_sts = fet_phy_readreg(sc, PHY_BMCR);
phy_sts |= PHY_BMCR_AUTONEGENBL|PHY_BMCR_AUTONEGRSTR;
fet_phy_writereg(sc, PHY_BMCR, phy_sts);
return;
}

/*
 * Invoke autonegotiation on a PHY.
 */
static void fet_autoneg_mii(sc, flag, verbose)
struct fet_softc *sc;
int flag;
int verbose;
{
u_int16_t phy_sts = 0, media, advert, ability;
u_int16_t ability2 = 0;
struct ifnet *ifp;
struct ifmedia *ifm;
ifm = &sc->ifmedia;
ifp = &sc->arpcom.ac_if;
ifm->ifm_media = IFM_ETHER | IFM_AUTO;
#ifdef FORCE_AUTONEG_TFOUR
/*
 * First, see if autoneg is supported. If not, there's
 * no point in continuing.
 */

```

```

phy_sts = fet_phy_readreg(sc, PHY_BMSR);
if (!(phy_sts & PHY_BMSR_CANAUTONEG))
{
if (verbose)
printf("fet%d: autonegotiation not supported\n", sc->fet_unit);
ifm->ifm_media = IFM_ETHER|IFM_10_T|IFM_HDX;
return;
}
#endif
switch (flag)
{
case FET_FLAG_FORCEDELAY:
/*
* XXX Never use this option anywhere but in the probe
* routine: making the kernel stop dead in its tracks
* for three whole seconds after we've gone multi-user
* is really bad manners.
*/
fet_autoneg_xmit(sc);
DELAY(5000000);
break;
case FET_FLAG_SCHEDEDELAY:
/*
* Wait for the transmitter to go idle before starting
* an autoneg session, otherwise fet_start() may clobber
* our timeout, and we don't want to allow transmission
* during an autoneg session since that can screw it up.
*/
if (sc->fet_cdata.fet_tx_head != NULL)
{
sc->fet_want_auto = 1;
return;
}
fet_autoneg_xmit(sc);
ifp->if_timer = 5;
sc->fet_autoneg = 1;
sc->fet_want_auto = 0;
return;
case FET_FLAG_DELAYTIMEO:
ifp->if_timer = 0;
sc->fet_autoneg = 0;
break;
default:
printf("fet%d: invalid autoneg flag: %d\n", sc->fet_unit, flag);
return;
}
if (fet_phy_readreg(sc, PHY_BMSR) & PHY_BMSR_AUTONEGCOMP)
{
if (verbose)
printf("fet%d: autoneg complete, ", sc->fet_unit);
phy_sts = fet_phy_readreg(sc, PHY_BMSR);

```

```

}
else
{
if (verbose)
printf("fet%d: autoneg not complete, ", sc->fet_unit);
}
media = fet_phy_readreg(sc, PHY_BMCR);
/* Link is good. Report modes and set duplex mode. */
if (fet_phy_readreg(sc, PHY_BMSR) & PHY_BMSR_LINKSTAT)
{
if (verbose)
printf("fet%d: link status good. ", sc->fet_unit);
advert = fet_phy_readreg(sc, PHY_ANAR);
ability = fet_phy_readreg(sc, PHY_LPAR);
if ( (sc->fet_pinfo->fet_vid == MarvellPHYID0) ||
(sc->fet_pinfo->fet_vid == LevelOnePHYID0) )
{
ability2 = fet_phy_readreg(sc, PHY_1000SR);
if (ability2 & PHY_1000SR_1000BTXFULL)
{
advert = 0;
ability = 0;
/* this version did not support 1000M,
ifm->ifm_media = IFM_ETHER|IFM_1000_TX|IFM_FDX;
*/
ifm->ifm_media = IFM_ETHER|IFM_100_TX|IFM_FDX;
media &= ~PHY_BMCR_SPEEDSEL;
media |= PHY_BMCR_1000;
media |= PHY_BMCR_DUPLEX;
printf("(full-duplex, 1000Mbps)\n");
}
else if (ability2 & PHY_1000SR_1000BTXHALF)
{
advert = 0;
ability = 0;
/* this version did not support 1000M,
ifm->ifm_media = IFM_ETHER|IFM_1000_TX;
*/
ifm->ifm_media = IFM_ETHER|IFM_100_TX;
media &= ~PHY_BMCR_SPEEDSEL;
media &= ~PHY_BMCR_DUPLEX;
media |= PHY_BMCR_1000;
printf("(half-duplex, 1000Mbps)\n");
}
}
}
if (advert & PHY_ANAR_100BT4 && ability & PHY_ANAR_100BT4)
{
ifm->ifm_media = IFM_ETHER|IFM_100_T4;
media |= PHY_BMCR_SPEEDSEL;
media &= ~PHY_BMCR_DUPLEX;
printf("(100baseT4)\n");
}
}

```

```

}
else if (advert & PHY_ANAR_100BTXFULL && ability & PHY_ANAR_100BTXFULL)
{
ifm->ifm_media = IFM_ETHER|IFM_100_TX|IFM_FDX;
media |= PHY_BMCR_SPEEDSEL;
media |= PHY_BMCR_DUPLEX;
printf("(full-duplex, 100Mbps)\n");
}
else if (advert & PHY_ANAR_100BTXHALF && ability & PHY_ANAR_100BTXHALF)
{
ifm->ifm_media = IFM_ETHER|IFM_100_TX|IFM_HDX;
media |= PHY_BMCR_SPEEDSEL;
media &= ~PHY_BMCR_DUPLEX;
printf("(half-duplex, 100Mbps)\n");
}
else if (advert & PHY_ANAR_10BTFULL && ability & PHY_ANAR_10BTFULL)
{
ifm->ifm_media = IFM_ETHER|IFM_10_T|IFM_FDX;
media &= ~PHY_BMCR_SPEEDSEL;
media |= PHY_BMCR_DUPLEX;
printf("(full-duplex, 10Mbps)\n");
}
else if (advert)
{
ifm->ifm_media = IFM_ETHER|IFM_10_T|IFM_HDX;
media &= ~PHY_BMCR_SPEEDSEL;
media &= ~PHY_BMCR_DUPLEX;
printf("(half-duplex, 10Mbps)\n");
}
media &= ~PHY_BMCR_AUTONEGENBL;
/* Set ASIC's duplex mode to match the PHY. */
fet_phy_writereg(sc, PHY_BMCR, media);
fet_setcfg(sc, media);
}
else
{
if (verbose)
printf("fet%d: no carrier\n", sc->fet_unit);
}
fet_init(sc);
if (sc->fet_tx_pend)
{
sc->fet_autoneg = 0;
sc->fet_tx_pend = 0;
fet_start(ifp);
}
return;
}

/*
* To get PHY ability.

```

```

*/
static void fet_getmode_mii(sc)
struct fet_softc *sc;
{
    u_int16_t bmsr;
    struct ifnet *ifp;
    ifp = &sc->arpcom.ac_if;
    bmsr = fet_phy_readreg(sc, PHY_BMSR);
    if (bootverbose)
        printf("fet%d: PHY status word: %x\n", sc->fet_unit, bmsr);
    /* fallback */
    sc->ifmedia.ifm_media = IFM_ETHER|IFM_10_T|IFM_HDX;
    if (bmsr & PHY_BMSR_10BTHALF)
    {
        if (bootverbose)
            printf("fet%d: 10Mbps half-duplex mode supported\n", sc->fet_unit);
        ifmedia_add(&sc->ifmedia, IFM_ETHER|IFM_10_T|IFM_HDX, 0, NULL);
        ifmedia_add(&sc->ifmedia, IFM_ETHER|IFM_10_T, 0, NULL);
    }
    if (bmsr & PHY_BMSR_10BTFULL)
    {
        if (bootverbose)
            printf("fet%d: 10Mbps full-duplex mode supported\n", sc->fet_unit);
        ifmedia_add(&sc->ifmedia, IFM_ETHER|IFM_10_T|IFM_FDX, 0, NULL);
        sc->ifmedia.ifm_media = IFM_ETHER|IFM_10_T|IFM_FDX;
    }
    if (bmsr & PHY_BMSR_100BTXHALF)
    {
        if (bootverbose)
            printf("fet%d: 100Mbps half-duplex mode supported\n", sc->fet_unit);
        ifp->if_baudrate = 100000000;
        ifmedia_add(&sc->ifmedia, IFM_ETHER|IFM_100_TX, 0, NULL);
        ifmedia_add(&sc->ifmedia, IFM_ETHER|IFM_100_TX|IFM_HDX, 0, NULL);
        sc->ifmedia.ifm_media = IFM_ETHER|IFM_100_TX|IFM_HDX;
    }
    if (bmsr & PHY_BMSR_100BTXFULL)
    {
        if (bootverbose)
            printf("fet%d: 100Mbps full-duplex mode supported\n", sc->fet_unit);
        ifp->if_baudrate = 100000000;
        ifmedia_add(&sc->ifmedia, IFM_ETHER|IFM_100_TX|IFM_FDX, 0, NULL);
        sc->ifmedia.ifm_media = IFM_ETHER|IFM_100_TX|IFM_FDX;
    }
    /* Some also support 100BaseT4. */
    if (bmsr & PHY_BMSR_100BT4)
    {
        if (bootverbose)
            printf("fet%d: 100baseT4 mode supported\n", sc->fet_unit);
        ifp->if_baudrate = 100000000;
        ifmedia_add(&sc->ifmedia, IFM_ETHER|IFM_100_T4, 0, NULL);
        sc->ifmedia.ifm_media = IFM_ETHER|IFM_100_T4;
    }
}

```

freebsd-net: Driver for SURECOM EP-320X-S 100/10M Ethernet PCI Adapter

```

#ifdef FORCE_AUTONEG_TFOUR
if (bootverbose)
printf("fet%d: forcing on autoneg support for BT4\n", sc->fet_unit);
ifmedia_add(&sc->ifmedia, IFM_ETHER|IFM_AUTO, 0 NULL);
sc->ifmedia.ifm_media = IFM_ETHER|IFM_AUTO;
#endif
}
/* this version did not support 1000M,
if (sc->fet_pinfo->fet_vid == MarvellPHYID0)
{
if (bootverbose)
printf("fet%d: 1000Mbps half-duplex mode supported\n", sc->fet_unit);
ifp->if_baudrate = 1000000000;
ifmedia_add(&sc->ifmedia, IFM_ETHER|IFM_1000_TX, 0, NULL);
ifmedia_add(&sc->ifmedia, IFM_ETHER|IFM_1000_TX|IFM_HDX, 0, NULL);
if (bootverbose)
printf("fet%d: 1000Mbps full-duplex mode supported\n", sc->fet_unit);
ifp->if_baudrate = 1000000000;
ifmedia_add(&sc->ifmedia, IFM_ETHER|IFM_1000_TX|IFM_FDX, 0, NULL);
sc->ifmedia.ifm_media = IFM_ETHER|IFM_1000_TX|IFM_FDX;
}
*/
if (bmsr & PHY_BMSR_CANAUTONEG)
{
if (bootverbose)
printf("fet%d: autoneg supported\n", sc->fet_unit);
ifmedia_add(&sc->ifmedia, IFM_ETHER|IFM_AUTO, 0, NULL);
sc->ifmedia.ifm_media = IFM_ETHER|IFM_AUTO;
}
return;
}

/*
* Set speed and duplex mode.
*/
static void fet_setmode_mii(sc, media)
struct fet_softc *sc;
int media;
{
u_int16_t bmcr;
struct ifnet *ifp;
ifp = &sc->arpcom.ac_if;
printf("enter fet_setmode_mii()\n");
/*
* If an autoneg session is in progress, stop it.
*/
if (sc->fet_autoneg)
{
printf("fet%d: canceling autoneg session\n", sc->fet_unit);
ifp->if_timer = sc->fet_autoneg = sc->fet_want_auto = 0;
bmcr = fet_phy_readreg(sc, PHY_BMCR);
}
}

```

```

bmcr &= ~PHY_BMCR_AUTONEGENBL;
fet_phy_writereg(sc, PHY_BMCR, bmcr);
}
printf("fet%d: selecting MII, ", sc->fet_unit);
bmcr = fet_phy_readreg(sc, PHY_BMCR);
bmcr &= ~(PHY_BMCR_AUTONEGENBL|PHY_BMCR_SPEEDSEL|PHY_BMCR_1000|
PHY_BMCR_DUPLEX|PHY_BMCR_LOOPBK);
/* this version did not support 1000M,
if (IFM_SUBTYPE(media) == IFM_1000_TX)
{
printf("1000Mbps/T4, half-duplex\n");
bmcr &= ~PHY_BMCR_SPEEDSEL;
bmcr &= ~PHY_BMCR_DUPLEX;
bmcr |= PHY_BMCR_1000;
}
*/
if (IFM_SUBTYPE(media) == IFM_100_T4)
{
printf("100Mbps/T4, half-duplex\n");
bmcr |= PHY_BMCR_SPEEDSEL;
bmcr &= ~PHY_BMCR_DUPLEX;
}
if (IFM_SUBTYPE(media) == IFM_100_TX)
{
printf("100Mbps, ");
bmcr |= PHY_BMCR_SPEEDSEL;
}
if (IFM_SUBTYPE(media) == IFM_10_T)
{
printf("10Mbps, ");
bmcr &= ~PHY_BMCR_SPEEDSEL;
}
if ((media & IFM_GMASK) == IFM_FDX)
{
printf("full duplex\n");
bmcr |= PHY_BMCR_DUPLEX;
}
else
{
printf("half duplex\n");
bmcr &= ~PHY_BMCR_DUPLEX;
}
fet_phy_writereg(sc, PHY_BMCR, bmcr);
fet_setcfg(sc, bmcr);
return;
}

/*
* The Myson manual states that in order to fiddle with the
* 'full-duplex' and '100Mbps' bits in the netconfig register, we
* first have to put the transmit and/or receive logic in the idle state.

```

```

*/
static void fet_setcfg(sc, bmcr)
struct fet_softc *sc;
int bmcr;
{
int i, restart = 0;
if (CSR_READ_4(sc, FET_TCRRCR) & (FET_TE|FET_RE))
{
restart = 1;
FET_CLRBIT(sc, FET_TCRRCR, (FET_TE|FET_RE));
for (i = 0; i < FET_TIMEOUT; i++)
{
DELAY(10);
if (!(CSR_READ_4(sc, FET_TCRRCR)&(FET_TXRUN|FET_RXRUN)))
break;
}
if (i == FET_TIMEOUT)
printf("fet%d: failed to force tx and rx to idle state\n",
sc->fet_unit);
}
FET_CLRBIT(sc, FET_TCRRCR, FET_PS1000);
FET_CLRBIT(sc, FET_TCRRCR, FET_PS10);
if (bmcr & PHY_BMCR_1000)
FET_SETBIT(sc, FET_TCRRCR, FET_PS1000);
else if (!(bmcr & PHY_BMCR_SPEEDSEL))
FET_SETBIT(sc, FET_TCRRCR, FET_PS10);
if (bmcr & PHY_BMCR_DUPLEX)
FET_SETBIT(sc, FET_TCRRCR, FET_FD);
else
FET_CLRBIT(sc, FET_TCRRCR, FET_FD);
if (restart)
FET_SETBIT(sc, FET_TCRRCR, FET_TE|FET_RE);
return;
}

static void fet_reset(sc)
struct fet_softc *sc;
{
register int i;
FET_SETBIT(sc, FET_BCR, FET_SWR);
for (i = 0; i < FET_TIMEOUT; i++)
{
DELAY(10);
if (!(CSR_READ_4(sc, FET_BCR) & FET_SWR))
break;
}
if (i == FET_TIMEOUT)
printf("m0x%d: reset never completed!\n", sc->fet_unit);
/* Wait a little while for the chip to get its brains in order. */
DELAY(1000);
return;
}

```

```

}

/*
 * Probe for a Myson chip. Check the PCI vendor and device
 * IDs against our list and return a device name if we find a match.
 */
static const char *fet_probe(config_id, device_id)
pcici_t config_id;
pcidi_t device_id;
{
    struct fet_type *t;
    t = fet_devs;
    while(t->fet_name != NULL)
    {
        if (((device_id & 0xFFFF) == t->fet_vid &&
            ((device_id >> 16) & 0xFFFF) == t->fet_did)
            {
                fet_info_tmp = t;
                return(t->fet_name);
            }
        t++;
    }
    return(NULL);
}

/*
 * Attach the interface. Allocate softc structures, do ifmedia
 * setup and ethernet/BPF attach.
 */
static void fet_attach(config_id, unit)
pcici_t config_id;
int unit;
{
    int s, i;
    vm_offset_t pbase, vbase;
    u_char eaddr[ETHER_ADDR_LEN];
    u_int32_t command, iobase;
    struct fet_softc *sc;
    struct ifnet *ifp;
    int media = IFM_ETHER|IFM_100_TX|IFM_FDX;
    unsigned int round;
    caddr_t roundptr;
    struct fet_type *p;
    u_int16_t phy_vid, phy_did, phy_sts;
    s = splimp();
    sc = malloc(sizeof(struct fet_softc), M_DEVBUF, M_NOWAIT);
    if (sc == NULL)
    {
        printf("fet%d: no memory for softc struct!\n", unit);
        return;
    }
}

```

```

bzero(sc, sizeof(struct fet_softc));
/*
 * Map control/status registers.
 */
command = pci_conf_read(config_id, PCI_COMMAND_STATUS_REG);
command |= (PCIM_CMD_PORTEN|PCIM_CMD_MEMEN|PCIM_CMD_BUSMASTEREN);
pci_conf_write(config_id, PCI_COMMAND_STATUS_REG, command);
command = pci_conf_read(config_id, PCI_COMMAND_STATUS_REG);
if (fet_info_tmp->fet_id==ID0)
{
iobase = pci_conf_read(config_id, FET_PCI_LOIO);
if (iobase & 0x300)
FET_USEIOSPACE=0;
}
if (FET_USEIOSPACE)
{
if (!(command & PCIM_CMD_PORTEN))
{
printf("fet%d: failed to enable I/O ports!\n", unit);
free(sc, M_DEVBUF);
goto fail;
}
if (!pci_map_port(config_id, FET_PCI_LOIO, (u_int16_t *)&(sc->fet_bhandle)))
{
printf ("fet%d: couldn't map ports\n", unit);
goto fail;
}
sc->fet_btag = I386_BUS_SPACE_IO;
}
else
{
if (!(command & PCIM_CMD_MEMEN))
{
printf("fet%d: failed to enable memory mapping!\n", unit);
goto fail;
}
if (!pci_map_mem(config_id, FET_PCI_LOMEM, &vbase, &pbase))
{
printf ("fet%d: couldn't map memory\n", unit);
goto fail;
}
}
/*
sc->csr = (volatile caddr_t)vbase;
*/
sc->fet_btag = I386_BUS_SPACE_MEM;
sc->fet_bhandle = vbase;
}
/* Allocate interrupt */
if (!pci_map_int(config_id, fet_intr, sc, &net_imask))
{
printf("fet%d: couldn't map interrupt\n", unit);
}

```

```

goto fail;
}
sc->fet_info = fet_info_tmp;
/* Reset the adapter. */
fet_reset(sc);
/*
 * Get station address
 */
for (i = 0; i < ETHER_ADDR_LEN; ++i)
eaddr[i] = CSR_READ_1(sc, FET_PAR0+i);
/*
 * A Myson chip was detected. Inform the world.
 */
printf("fet%d: Ethernet address: %6D\n", unit, eaddr, ":");
sc->fet_unit = unit;
bcopy(eaddr, (char *)&sc->arpcom.ac_enaddr, ETHER_ADDR_LEN);
sc->fet_ldata_ptr = malloc(sizeof(struct fet_list_data) + 8,
M_DEVBUF, M_NOWAIT);
if (sc->fet_ldata_ptr == NULL)
{
free(sc, M_DEVBUF);
printf("fet%d: no memory for list buffers!\n", unit);
return;
}
sc->fet_ldata = (struct fet_list_data *)sc->fet_ldata_ptr;
round = (unsigned int)sc->fet_ldata_ptr & 0xF;
roundptr = sc->fet_ldata_ptr;
for (i = 0; i < 8; i++)
{
if (round % 8)
{
round++;
roundptr++;
}
else
break;
}
sc->fet_ldata = (struct fet_list_data *)roundptr;
bzero(sc->fet_ldata, sizeof(struct fet_list_data));
ifp = &sc->arpcom.ac_if;
ifp->if_softc = sc;
ifp->if_unit = unit;
ifp->if_name = "fet";
ifp->if_mtu = ETHERMTU;
ifp->if_flags = IFF_BROADCAST | IFF_SIMPLEX | IFF_MULTICAST;
ifp->if_ioctl = fet_ioctl;
ifp->if_output = ether_output;
ifp->if_start = fet_start;
ifp->if_watchdog = fet_watchdog;
ifp->if_init = fet_init;
ifp->if_baudrate = 10000000;

```

```

if (sc->fet_info->fet_did == ID1)
sc->fet_pinfo = fet_phys;
else
{
if (bootverbose)
printf("fet%d: probing for a PHY\n", sc->fet_unit);
for (i = FET_PHYADDR_MIN; i < FET_PHYADDR_MAX + 1; i++)
{
if (bootverbose)
printf("fet%d: checking address: %d\n", sc->fet_unit, i);
sc->fet_phy_addr = i;
phy_sts = fet_phy_readreg(sc, PHY_BMSR);
if ( (phy_sts!=0)&&(phy_sts!=0xffff) )
break;
else
phy_sts = 0;
}
if (phy_sts)
{
phy_vid = fet_phy_readreg(sc, PHY_VENID);
phy_did = fet_phy_readreg(sc, PHY_DEVID);
if (bootverbose)
{
printf("fet%d: found PHY at address %d, ",
sc->fet_unit, sc->fet_phy_addr);
printf("vendor id: %x device id: %x\n", phy_vid, phy_did);
}
p = fet_phys;
while(p->fet_vid)
{
if (phy_vid == p->fet_vid)
{
sc->fet_pinfo = p;
break;
}
p++;
}
if (sc->fet_pinfo == NULL)
sc->fet_pinfo = &fet_phys[PHY_UNKNOWN];
if (bootverbose)
printf("fet%d: PHY type: %s\n",
sc->fet_unit, sc->fet_pinfo->fet_name);
}
else
{
printf("fet%d: MII without any phy!\n", sc->fet_unit);
goto fail;
}
}
/*
* Do ifmedia setup.

```

```

*/
ifmedia_init(&sc->ifmedia, 0, fet_ifmedia_upd, fet_ifmedia_sts);
fet_getmode_mii(sc);
fet_autoneg_mii(sc, FET_FLAG_FORCEDELAY, 1);
media = sc->ifmedia.ifm_media;
fet_stop(sc);
ifmedia_set(&sc->ifmedia, media);
/*
* Call MI attach routines.
*/
if_attach(ifp);
ether_ifattach(ifp);
#if NBPFILTER > 0
bpfattach(ifp, DLT_EN10MB, sizeof(struct ether_header));
#endif
at_shutdown(fet_shutdown, sc, SHUTDOWN_POST_SYNC);
fail:
splx(s);
return;
}

/*
* Initialize the transmit descriptors.
*/
static int fet_list_tx_init(sc)
struct fet_softc *sc;
{
    struct fet_chain_data *cd;
    struct fet_list_data *ld;
    int i;
    cd = &sc->fet_cdata;
    ld = sc->fet_ldata;
    for (i = 0; i < FET_TX_LIST_CNT; i++)
    {
        cd->fet_tx_chain[i].fet_ptr = &ld->fet_tx_list[i];
        if (i == (FET_TX_LIST_CNT - 1))
            cd->fet_tx_chain[i].fet_nextdesc = &cd->fet_tx_chain[0];
        else
            cd->fet_tx_chain[i].fet_nextdesc = &cd->fet_tx_chain[i + 1];
    }
    cd->fet_tx_free = &cd->fet_tx_chain[0];
    cd->fet_tx_tail = cd->fet_tx_head = NULL;
    return(0);
}

/*
* Initialize the RX descriptors and allocate mbufs for them. Note that
* we arrange the descriptors in a closed ring, so that the last descriptor
* points back to the first.
*/
static int fet_list_rx_init(sc)

```

```

struct fet_softc *sc;
{
struct fet_chain_data *cd;
struct fet_list_data *ld;
int i;
cd = &sc->fet_cdata;
ld = sc->fet_ldata;
for (i = 0; i < FET_RX_LIST_CNT; i++)
{
cd->fet_rx_chain[i].fet_ptr = (struct fet_desc *)&ld->fet_rx_list[i];
if (fet_newbuf(sc, &cd->fet_rx_chain[i]) == ENOBUFS)
return(ENOBUFS);
if (i == (FET_RX_LIST_CNT - 1))
{
cd->fet_rx_chain[i].fet_nextdesc = &cd->fet_rx_chain[0];
ld->fet_rx_list[i].fet_next = vtophys(&ld->fet_rx_list[0]);
}
else
{
cd->fet_rx_chain[i].fet_nextdesc = &cd->fet_rx_chain[i + 1];
ld->fet_rx_list[i].fet_next = vtophys(&ld->fet_rx_list[i + 1]);
}
}
cd->fet_rx_head = &cd->fet_rx_chain[0];
return(0);
}

/*
 * Initialize an RX descriptor and attach an MBUF cluster.
 */
static int fet_newbuf(sc, c)
struct fet_softc *sc;
struct fet_chain_onefrag *c;
{
struct mbuf *m_new = NULL;
MGETHDR(m_new, M_DONTWAIT, MT_DATA);
if (m_new == NULL)
{
printf("fet%d: no memory for rx list -- packet dropped!\n",
sc->fet_unit);
return(ENOBUFS);
}
MCLGET(m_new, M_DONTWAIT);
if (!(m_new->m_flags & M_EXT))
{
printf("fet%d: no memory for rx list -- packet dropped!\n",
sc->fet_unit);
m_freem(m_new);
return(ENOBUFS);
}
c->fet_mbuf = m_new;

```

```

c->fet_ptr->fet_data = vtophys(mtod(m_new, caddr_t));
c->fet_ptr->fet_ctl = (MCLBYTES - 1) << FET_RBSShift;
c->fet_ptr->fet_status = FET_OWNBByNIC;
return(0);
}

/*
 * A frame has been uploaded: pass the resulting mbuf chain up to
 * the higher level protocols.
 */
static void fet_rxeof(sc)
struct fet_softc *sc;
{
    struct ether_header *eh;
    struct mbuf *m;
    struct ifnet *ifp;
    struct fet_chain_onefrag *cur_rx;
    int total_len = 0;
    u_int32_t rxstat;
    ifp = &sc->arpcom.ac_if;
    while (!((rxstat =
sc->fet_cdata.fet_rx_head->fet_ptr->fet_status)&FET_OWNBByNIC))
    {
        cur_rx = sc->fet_cdata.fet_rx_head;
        sc->fet_cdata.fet_rx_head = cur_rx->fet_nextdesc;
        if (rxstat & FET_ES) /* error summary */
            { /* give up this rx pkt */
                ifp->if_ierrors++;
                cur_rx->fet_ptr->fet_status = FET_OWNBByNIC;
                continue;
            }
        /* No errors; receive the packet. */
        total_len = (rxstat & FET_FLNGMASK) >> FET_FLNGShift;
        total_len -= ETHER_CRC_LEN;
        if (total_len < MINCLSIZE)
            {
                m = m_devget(mtod(cur_rx->fet_mbuf, char *), total_len, 0, ifp, NULL);
                cur_rx->fet_ptr->fet_status = FET_OWNBByNIC;
                if (m == NULL)
                    {
                        ifp->if_ierrors++;
                        continue;
                    }
            }
        else
            {
                m = cur_rx->fet_mbuf;
                /*
                 * Try to conjure up a new mbuf cluster. If that
                 * fails, it means we have an out of memory condition and
                 * should leave the buffer in place and continue. This will

```

```

* result in a lost packet, but there's little else we
* can do in this situation.
*/
if (fet_newbuf(sc, cur_rx) == ENOBUFS)
{
ifp->if_ierrors++;
cur_rx->fet_ptr->fet_status = FET_OWNBByNIC;
continue;
}
m->m_pkthdr.rcvif = ifp;
m->m_pkthdr.len = m->m_len = total_len;
}
ifp->if_ipackets++;
eh = mtod(m, struct ether_header *);
#if NBPFILTER > 0
/*
* Handle BPF listeners. Let the BPF user see the packet, but
* don't pass it up to the ether_input() layer unless it's
* a broadcast packet, multicast packet, matches our ethernet
* address or the interface is in promiscuous mode.
*/
if (ifp->if_bpf)
{
bpf_mtap(ifp, m);
if (ifp->if_flags & IFF_PROMISC &&
(bcmbp(eh->ether_dhost, sc->arpcom.ac_enaddr, ETHER_ADDR_LEN) &&
(eh->ether_dhost[0] & 1) == 0))
{
m_freem(m);
continue;
}
}
#endif
/* Remove header from mbuf and pass it on. */
m_adj(m, sizeof(struct ether_header));
ether_input(ifp, eh, m);
}
return;
}

/*
* A frame was downloaded to the chip. It's safe for us to clean up
* the list buffers.
*/
static void fet_txeof(sc)
struct fet_softc *sc;
{
struct fet_chain *cur_tx;
struct ifnet *ifp;
ifp = &sc->arpcom.ac_if;
/* Clear the timeout timer. */

```

```

ifp->if_timer = 0;
if (sc->fet_cdata.fet_tx_head == NULL)
return;
/*
 * Go through our tx list and free mbufs for those
 * frames that have been transmitted.
 */
while (sc->fet_cdata.fet_tx_head->fet_mbuf != NULL)
{
u_int32_t txstat;
cur_tx = sc->fet_cdata.fet_tx_head;
txstat = FET_TXSTATUS(cur_tx);
if ((txstat & FET_OWNBByNIC) || txstat == FET_UNSENT)
break;
if (!(CSR_READ_4(sc, FET_TCRRCR)&FET_Enhanced))
{
if (txstat & FET_TXERR)
{
ifp->if_oerrors++;
if (txstat & FET_EC) /* excessive collision */
ifp->if_collisions++;
if (txstat & FET_LC) /* late collision */
ifp->if_collisions++;
}
ifp->if_collisions += (txstat & FET_NCRMASK) >> FET_NCRShift;
}
ifp->if_opackets++;
m_freem(cur_tx->fet_mbuf);
cur_tx->fet_mbuf = NULL;
if (sc->fet_cdata.fet_tx_head == sc->fet_cdata.fet_tx_tail)
{
sc->fet_cdata.fet_tx_head = NULL;
sc->fet_cdata.fet_tx_tail = NULL;
break;
}
sc->fet_cdata.fet_tx_head = cur_tx->fet_nextdesc;
}
if (CSR_READ_4(sc, FET_TCRRCR)&FET_Enhanced)
{
ifp->if_collisions += (CSR_READ_4(sc, FET_TSR)&FET_NCRMMask);
}
return;
}

/*
 * TX 'end of channel' interrupt handler.
 */
static void fet_txeoc(sc)
struct fet_softc *sc;
{
struct ifnet *ifp;

```

```

ifp = &sc->arpcom.ac_if;
ifp->if_timer = 0;
if (sc->fet_cdata.fet_tx_head == NULL)
{
ifp->if_flags &= ~IFF_OACTIVE;
sc->fet_cdata.fet_tx_tail = NULL;
if (sc->fet_want_auto)
fet_autoneg_mii(sc, FET_FLAG_SCHDELAY, 1);
}
else
{
if (FET_TXOWN(sc->fet_cdata.fet_tx_head) == FET_UNSENT)
{
FET_TXOWN(sc->fet_cdata.fet_tx_head) = FET_OWNByNIC;
ifp->if_timer = 5;
CSR_WRITE_4(sc, FET_TXPDR, 0xFFFFFFFF);
}
}
return;
}

```

```

static void fet_intr(arg)
void *arg;
{
struct fet_softc *sc;
struct ifnet *ifp;
u_int32_t status;
sc = arg;
ifp = &sc->arpcom.ac_if;
if (!(ifp->if_flags & IFF_UP))
return;
/* Disable interrupts. */
CSR_WRITE_4(sc, FET_IMR, 0x00000000);
for (;;)
{
status = CSR_READ_4(sc, FET_ISR);
status &= FET_INTRS;
if (status)
CSR_WRITE_4(sc, FET_ISR, status);
else
break;
if (status & FET_RI) /* receive interrupt */
fet_rxeof(sc);
if ((status & FET_RBU) || (status & FET_RxErr))
{ /* rx buffer unavailable or rx error */
ifp->if_ierrors++;
#ifdef foo
fet_stop(sc);
fet_reset(sc);
fet_init(sc);
#endif
}
}
}

```

```

}
if (status & FET_TI) /* tx interrupt */
fet_txeof(sc);
if (status & FET_ETI) /* tx early interrupt */
fet_txeof(sc);
if (status & FET_TBU) /* tx buffer unavailable */
fet_txeoc(sc);
/* 90/1/18 delete
if (status & FET_FBE)
{
fet_reset(sc);
fet_init(sc);
}
*/
}
/* Re-enable interrupts. */
CSR_WRITE_4(sc, FET_IMR, FET_INTRS);
if (ifp->if_snd.ifq_head != NULL)
fet_start(ifp);
return;
}

/*
* Encapsulate an mbuf chain in a descriptor by coupling the mbuf data
* pointers to the fragment pointers.
*/
static int fet_encap(sc, c, m_head)
struct fet_softc *sc;
struct fet_chain *c;
struct mbuf *m_head;
{
struct fet_desc *f = NULL;
int total_len;
struct mbuf *m, *m_new=NULL;
/* calculate the total tx pkt length */
total_len = 0;
for (m = m_head; m != NULL; m = m->m_next)
total_len += m->m_len;
/*
* Start packing the mbufs in this chain into
* the fragment pointers. Stop when we run out
* of fragments or hit the end of the mbuf chain.
*/
m = m_head;
MGETHDR(m_new, M_DONTWAIT, MT_DATA);
if (m_new == NULL)
{
printf("fet%d: no memory for tx list", sc->fet_unit);
return(1);
}
if (m_head->m_pkthdr.len > MHLEN)

```

```

{
MCLGET(m_new, M_DONTWAIT);
if (!(m_new->m_flags & M_EXT))
{
m_freem(m_new);
printf("fet%d: no memory for tx list", sc->fet_unit);
return(1);
}
}
m_copydata(m_head, 0, m_head->m_pkthdr.len, mtod(m_new, caddr_t));
m_new->m_pkthdr.len = m_new->m_len = m_head->m_pkthdr.len;
m_freem(m_head);
m_head = m_new;
f = &c->fet_ptr->fet_frag[0];
f->fet_status = 0;
f->fet_data = vtophys(mtod(m_new, caddr_t));
total_len = m_new->m_len;
f->fet_ctl = FET_TXFD|FET_TXLD|FET_CRCEnable|FET_PADEnable;
f->fet_ctl |= total_len<<FET_PKTShift; /* pkt size */
f->fet_ctl |= total_len; /* buffer size */
/* 89/12/29 add, for mtd891 */
if (sc->fet_info->fet_did==ID2)
f->fet_ctl |= FET_ETIControl|FET_RetryTxLC;
c->fet_mbuf = m_head;
c->fet_lastdesc = 0;
FET_TXNEXT(c) = vtophys(&c->fet_nextdesc->fet_ptr->fet_frag[0]);
return(0);
}

/*
* Main transmit routine. To avoid having to do mbuf copies, we put pointers
* to the mbuf data regions directly in the transmit lists. We also save a
* copy of the pointers since the transmit list fragment pointers are
* physical addresses.
*/
static void fet_start(ifp)
struct ifnet *ifp;
{
struct fet_softc *sc;
struct mbuf *m_head = NULL;
struct fet_chain *cur_tx = NULL, *start_tx;
sc = ifp->if_softc;
if (sc->fet_autoneg)
{
sc->fet_tx_pend = 1;
return;
}
}
/*
* Check for an available queue slot. If there are none,
* punt.
*/

```

```

if (sc->fet_cdata.fet_tx_free->fet_mbuf != NULL)
{
ifp->if_flags |= IFF_OACTIVE;
return;
}
start_tx = sc->fet_cdata.fet_tx_free;
while (sc->fet_cdata.fet_tx_free->fet_mbuf == NULL)
{
IF_DEQUEUE(&ifp->if_snd, m_head);
if (m_head == NULL)
break;
/* Pick a descriptor off the free list. */
cur_tx = sc->fet_cdata.fet_tx_free;
sc->fet_cdata.fet_tx_free = cur_tx->fet_nextdesc;
/* Pack the data into the descriptor. */
fet_encap(sc, cur_tx, m_head);
if (cur_tx != start_tx)
FET_TXOWN(cur_tx) = FET_OWNBByNIC;
#if NBPFILTER > 0
/*
* If there's a BPF listener, bounce a copy of this frame
* to him.
*/
if (ifp->if_bpf)
bpf_mtap(ifp, cur_tx->fet_mbuf);
#endif
}
/*
* If there are no packets queued, bail.
*/
if (cur_tx == NULL)
return;
/*
* Place the request for the upload interrupt
* in the last descriptor in the chain. This way, if
* we're chaining several packets at once, we'll only
* get an interrupt once for the whole chain rather than
* once for each packet.
*/
FET_TXCTL(cur_tx) |= FET_TXIC;
cur_tx->fet_ptr->fet_frag[0].fet_ctl |= FET_TXIC;
sc->fet_cdata.fet_tx_tail = cur_tx;
if (sc->fet_cdata.fet_tx_head == NULL)
sc->fet_cdata.fet_tx_head = start_tx;
FET_TXOWN(start_tx) = FET_OWNBByNIC;
CSR_WRITE_4(sc, FET_TXPDR, 0xFFFFFFFF); /* tx polling demand */
/*
* Set a timeout in case the chip goes out to lunch.
*/
ifp->if_timer = 5;
return;

```

```

}

static void fet_init(xsc)
void *xsc;
{
    struct fet_softc *sc = xsc;
    struct ifnet *ifp = &sc->arpcom.ac_if;
    int s, i;
    u_int16_t phy_bmcr = 0;
    if (sc->fet_autoneg)
        return;
    s = splimp();
    if (sc->fet_pinfo != NULL)
        phy_bmcr = fet_phy_readreg(sc, PHY_BMCR);
    /*
     * Cancel pending I/O and free all RX/TX buffers.
     */
    fet_stop(sc);
    fet_reset(sc);
    /*
     * Set cache alignment and burst length.
     */
    /* 89/9/1 modify,
    CSR_WRITE_4(sc, FET_BCR, FET_RPBLE512);
    CSR_WRITE_4(sc, FET_TCRRCR, FET_TFTSF);
    */
    CSR_WRITE_4(sc, FET_BCR, FET_PBL8);
    CSR_WRITE_4(sc, FET_TCRRCR, FET_TFTSF|FET_RBLEN|FET_RPBLE512);
    /*
    89/12/29 add, for mtd891,
    */
    if (sc->fet_info->fet_did==ID2)
    {
        FET_SETBIT(sc, FET_BCR, FET_PROG);
        FET_SETBIT(sc, FET_TCRRCR, FET_Enhanced);
    }
    fet_setcfg(sc, phy_bmcr);
    /* Init circular RX list. */
    if (fet_list_rx_init(sc) == ENOBUFS)
    {
        printf("fet%d: initialization failed: no memory for rx buffers\n",
            sc->fet_unit);
        fet_stop(sc);
        (void)splx(s);
        return;
    }
    /* Init TX descriptors. */
    fet_list_tx_init(sc);
    /* If we want promiscuous mode, set the allframes bit. */
    if (ifp->if_flags & IFF_PROMISC)
        FET_SETBIT(sc, FET_TCRRCR, FET_PROM);
}

```

```

else
FET_CLRBIT(sc, FET_TCRRCR, FET_PROM);
/*
* Set capture broadcast bit to capture broadcast frames.
*/
if (ifp->if_flags & IFF_BROADCAST)
FET_SETBIT(sc, FET_TCRRCR, FET_AB);
else
FET_CLRBIT(sc, FET_TCRRCR, FET_AB);
/*
* Program the multicast filter, if necessary.
*/
fet_setmulti(sc);
/*
* Load the address of the RX list.
*/
FET_CLRBIT(sc, FET_TCRRCR, FET_RE);
CSR_WRITE_4(sc, FET_RXLBA, vtophys(&sc->fet_ldata->fet_rx_list[0]));
/*
* Enable interrupts.
*/
CSR_WRITE_4(sc, FET_IMR, FET_INTRS);
CSR_WRITE_4(sc, FET_ISR, 0xFFFFFFFF);
/* Enable receiver and transmitter. */
FET_SETBIT(sc, FET_TCRRCR, FET_RE);
FET_CLRBIT(sc, FET_TCRRCR, FET_TE);
CSR_WRITE_4(sc, FET_TXLBA, vtophys(&sc->fet_ldata->fet_tx_list[0]));
FET_SETBIT(sc, FET_TCRRCR, FET_TE);
/* Restore state of BMCR */
if (sc->fet_pinfo != NULL)
fet_phy_writereg(sc, PHY_BMCR, phy_bmcr);
ifp->if_flags |= IFF_RUNNING;
ifp->if_flags &= ~IFF_OACTIVE;
(void)splx(s);
return;
}

/*
* Set media options.
*/
static int fet_ifmedia_upd(ifp)
struct ifnet *ifp;
{
struct fet_softc *sc;
struct ifmedia *ifm;
sc = ifp->if_softc;
ifm = &sc->ifmedia;
if (IFM_TYPE(ifm->ifm_media) != IFM_ETHER)
return(EINVAL);
if (IFM_SUBTYPE(ifm->ifm_media) == IFM_AUTO)
fet_autoneg_mii(sc, FET_FLAG_SCHEDDELAY, 1);

```

```

else
fet_setmode_mii(sc, ifm->ifm_media);
return(0);
}

/*
 * Report current media status.
 */
static void fet_ifmedia_sts(ifp, ifmr)
struct ifnet *ifp;
struct ifmediareq *ifmr;
{
struct fet_softc *sc;
u_int16_t advert = 0, ability = 0, ability2 = 0;
sc = ifp->if_softc;
ifmr->ifm_active = IFM_ETHER;
if (!(fet_phy_readreg(sc, PHY_BMCR) & PHY_BMCR_AUTONEGENBL))
{
/* this version did not support 1000M,
if (fet_phy_readreg(sc, PHY_BMCR) & PHY_BMCR_1000)
ifmr->ifm_active = IFM_ETHER|IFM_1000TX;
*/
if (fet_phy_readreg(sc, PHY_BMCR) & PHY_BMCR_SPEEDSEL)
ifmr->ifm_active = IFM_ETHER|IFM_100_TX;
else
ifmr->ifm_active = IFM_ETHER|IFM_10_T;
if (fet_phy_readreg(sc, PHY_BMCR) & PHY_BMCR_DUPLEX)
ifmr->ifm_active |= IFM_FDX;
else
ifmr->ifm_active |= IFM_HDX;
return;
}
ability = fet_phy_readreg(sc, PHY_LPAR);
advert = fet_phy_readreg(sc, PHY_ANAR);
/* this version did not support 1000M,
if (sc->fet_pinfo->fet_vid = MarvellPHYID0)
{
ability2 = fet_phy_readreg(sc, PHY_1000SR);
if (ability2 & PHY_1000SR_1000BTXFULL)
{
advert = 0;
ability = 0;
ifmr->ifm_active = IFM_ETHER|IFM_1000_TX|IFM_FDX;
}
else if (ability & PHY_1000SR_1000BTXHALF)
{
advert = 0;
ability = 0;
ifmr->ifm_active = IFM_ETHER|IFM_1000_TX|IFM_HDX;
}
}
}

```

```

*/
if (advert & PHY_ANAR_100BT4 && ability & PHY_ANAR_100BT4)
ifmr->ifm_active = IFM_ETHER|IFM_100_T4;
else if (advert & PHY_ANAR_100BTXFULL && ability & PHY_ANAR_100BTXFULL)
ifmr->ifm_active = IFM_ETHER|IFM_100_TX|IFM_FDX;
else if (advert & PHY_ANAR_100BTXHALF && ability & PHY_ANAR_100BTXHALF)
ifmr->ifm_active = IFM_ETHER|IFM_100_TX|IFM_HDX;
else if (advert & PHY_ANAR_10BTFFULL && ability & PHY_ANAR_10BTFFULL)
ifmr->ifm_active = IFM_ETHER|IFM_10_T|IFM_FDX;
else if (advert & PHY_ANAR_10BTXHALF && ability & PHY_ANAR_10BTXHALF)
ifmr->ifm_active = IFM_ETHER|IFM_10_T|IFM_HDX;
return;
}

```

```

static int fet_ioctl(ifp, command, data)
struct ifnet *ifp;
u_long command;
caddr_t data;
{
struct fet_softc *sc = ifp->if_softc;
struct ifreq *ifr = (struct ifreq *)data;
int s, error = 0;
s = splimp();
switch(command)
{
case SIOCSIFADDR:
case SIOCGIFADDR:
case SIOCSIFMTU:
error = ether_ioctl(ifp, command, data);
break;
case SIOCSIFFLAGS:
if (ifp->if_flags & IFF_UP)
fet_init(sc);
else if (ifp->if_flags & IFF_RUNNING)
fet_stop(sc);
error = 0;
break;
case SIOCADDMULTI:
case SIOCDELMULTI:
fet_setmulti(sc);
error = 0;
break;
case SIOCGIFMEDIA:
case SIOCSIFMEDIA:
error = ifmedia_ioctl(ifp, ifr, &sc->ifmedia, command);
break;
default:
error = EINVAL;
break;
}
(void)splx(s);

```

```

return(error);
}

static void fet_watchdog(ifp)
struct ifnet *ifp;
{
    struct fet_softc *sc;
    sc = ifp->if_softc;
    if (sc->fet_autoneg)
    {
        fet_autoneg_mii(sc, FET_FLAG_DELAYTIMEO, 1);
        return;
    }
    ifp->if_oerrors++;
    printf("fet%d: watchdog timeout\n", sc->fet_unit);
    if (!(fet_phy_readreg(sc, PHY_BMSR) & PHY_BMSR_LINKSTAT))
        printf("fet%d: no carrier - transceiver cable problem?\n", sc->fet_unit);
    fet_stop(sc);
    fet_reset(sc);
    fet_init(sc);
    if (ifp->if_snd.ifq_head != NULL)
        fet_start(ifp);
    return;
}

/*
 * Stop the adapter and free any mbufs allocated to the
 * RX and TX lists.
 */
static void fet_stop(sc)
struct fet_softc *sc;
{
    register int i;
    struct ifnet *ifp;
    ifp = &sc->arpcom.ac_if;
    ifp->if_timer = 0;
    FET_CLRBIT(sc, FET_TCRRCR, (FET_RE|FET_TE));
    CSR_WRITE_4(sc, FET_IMR, 0x00000000);
    CSR_WRITE_4(sc, FET_TXLBA, 0x00000000);
    CSR_WRITE_4(sc, FET_RXLBA, 0x00000000);
}
/*
 * Free data in the RX lists.
 */
for (i = 0; i < FET_RX_LIST_CNT; i++)
{
    if (sc->fet_cdata.fet_rx_chain[i].fet_mbuf != NULL)
    {
        m_freem(sc->fet_cdata.fet_rx_chain[i].fet_mbuf);
        sc->fet_cdata.fet_rx_chain[i].fet_mbuf = NULL;
    }
}

```

```

bzero((char *)&sc->fet_ldata->fet_rx_list,
sizeof(sc->fet_ldata->fet_rx_list));
/*
 * Free the TX list buffers.
 */
for (i = 0; i < FET_TX_LIST_CNT; i++)
{
if (sc->fet_cdata.fet_tx_chain[i].fet_mbuf != NULL)
{
m_freem(sc->fet_cdata.fet_tx_chain[i].fet_mbuf);
sc->fet_cdata.fet_tx_chain[i].fet_mbuf = NULL;
}
}
bzero((char *)&sc->fet_ldata->fet_tx_list,
sizeof(sc->fet_ldata->fet_tx_list));
ifp->if_flags &= ~(IFF_RUNNING | IFF_OACTIVE);
return;
}

/*
 * Stop all chip I/O so that the kernel's probe routines don't
 * get confused by errant DMAs when rebooting.
 */
static void fet_shutdown(howto, arg)
int howto;
void *arg;
{
struct fet_softc *sc = (struct fet_softc *)arg;
fet_stop(sc);
return;
}

static struct pci_device fet_device = {
"fet",
fet_probe,
fet_attach,
&fet_count,
NULL
};
DATA_SET(pcidevice_set, fet_device);

```

---

if\_fetreg.h

---

```

/*
 * register definitions.
 */
#define FET_PAR0 0x0 /* physical address 0-3 */
#define FET_PAR1 0x04 /* physical address 4-5 */
#define FET_MAR0 0x08 /* multicast address 0-3 */

```

## freebsd-net: Driver for SURECOM EP-320X-S 100/10M Ethernet PCI Adapter

```
#define FET_MAR1 0x0C /* multicast address 4-7 */
#define FET_FAR0 0x10 /* flow-control address 0-3 */
#define FET_FAR1 0x14 /* flow-control address 4-5 */
#define FET_TCRRCR 0x18 /* receive & transmit configuration */
#define FET_BCR 0x1C /* bus command */
#define FET_TXPDR 0x20 /* transmit polling demand */
#define FET_RXPDR 0x24 /* receive polling demand */
#define FET_RXCWP 0x28 /* receive current word pointer */
#define FET_TXLBA 0x2C /* transmit list base address */
#define FET_RXLBA 0x30 /* receive list base address */
#define FET_ISR 0x34 /* interrupt status */
#define FET_IMR 0x38 /* interrupt mask */
#define FET_FTH 0x3C /* flow control high/low threshold */
#define FET_MANAGEMENT 0x40 /* bootrom/eeprom and mii management */
#define FET_TALLY 0x44 /* tally counters for crc and mpa */
#define FET_TSR 0x48 /* tally counter for transmit status */
#define FET_PHYBASE 0x4c
/*
 * Receive Configuration Register
 */
#define FET_RXRUN 0x00008000 /* receive running status */
#define FET_EIEN 0x00004000 /* early interrupt enable */
#define FET_RFCEN 0x00002000 /* receive flow control packet enable */
#define FET_NDFA 0x00001000 /* not defined flow control address */
#define FET_RBLLEN 0x00000800 /* receive burst length enable */
#define FET_RPBLE1 0x00000000 /* 1 word */
#define FET_RPBLE4 0x00000100 /* 4 words */
#define FET_RPBLE8 0x00000200 /* 8 words */
#define FET_RPBLE16 0x00000300 /* 16 words */
#define FET_RPBLE32 0x00000400 /* 32 words */
#define FET_RPBLE64 0x00000500 /* 64 words */
#define FET_RPBLE128 0x00000600 /* 128 words */
#define FET_RPBLE512 0x00000700 /* 512 words */
#define FET_PROM 0x00000080 /* promiscuous mode */
#define FET_AB 0x00000040 /* accept broadcast */
#define FET_AM 0x00000020 /* accept mutlicast */
#define FET_ARP 0x00000008 /* receive runt pkt */
#define FET_ALP 0x00000004 /* receive long pkt */
#define FET_SEP 0x00000002 /* receive error pkt */
#define FET_RE 0x00000001 /* receive enable */
/*
 * Transmit Configuration Register
 */
#define FET_TXRUN 0x04000000 /* transmit running status */
#define FET_Enhanced 0x02000000 /* transmit enhanced mode */
#define FET_TFCEN 0x01000000 /* tx flow control packet enable */
#define FET_TFT64 0x00000000 /* 64 bytes */
#define FET_TFT32 0x00200000 /* 32 bytes */
#define FET_TFT128 0x00400000 /* 128 bytes */
#define FET_TFT256 0x00600000 /* 256 bytes */
#define FET_TFT512 0x00800000 /* 512 bytes */
```

## freebsd-net: Driver for SURECOM EP-320X-S 100/10M Ethernet PCI Adapter

```
#define FET_TFT768 0x00A00000 /* 768 bytes */
#define FET_TFT1024 0x00C00000 /* 1024 bytes */
#define FET_TFTSF 0x00E00000 /* store and forward */
#define FET_FD 0x00100000 /* full duplex mode */
#define FET_PS10 0x00080000 /* port speed is 10M */
#define FET_TE 0x00040000 /* transmit enable */
#define FET_PS1000 0x00010000 /* port speed is 1000M */
/*
 * Bus Command Register
 */
#define FET_PROG 0x00000200 /* programming */
#define FET_RLE 0x00000100 /* read line command enable */
#define FET_RME 0x00000080 /* read multiple command enable */
#define FET_WIE 0x00000040 /* write and invalidate cmd enable */
#define FET_PBL1 0x00000000 /* 1 dword */
#define FET_PBL4 0x00000008 /* 4 dwords */
#define FET_PBL8 0x00000010 /* 8 dwords */
#define FET_PBL16 0x00000018 /* 16 dwords */
#define FET_PBL32 0x00000020 /* 32 dwords */
#define FET_PBL64 0x00000028 /* 64 dwords */
#define FET_PBL128 0x00000030 /* 128 dwords */
#define FET_PBL512 0x00000038 /* 512 dwords */
#define FET_ABR 0x00000004 /* arbitration rule */
#define FET_BLS 0x00000002 /* big/little endian select */
#define FET_SWR 0x00000001 /* software reset */
/*
 * Transmit Poll Demand Register
 */
#define FET_TxPollDemand 0x1
/*
 * Receive Poll Demand Register
 */
#define FET_RxPollDemand 0x01
/*
 * Interrupt Status Register
 */
#define FET_RFCON 0x00020000 /* receive flow control xon packet */
#define FET_RFCOFF 0x00010000 /* receive flow control xoff packet */
#define FET_LSCStatus 0x00008000 /* link status change */
#define FET_ANCStatus 0x00004000 /* autonegotiation completed */
#define FET_FBE 0x00002000 /* fatal bus error */
#define FET_FBEMask 0x00001800
#define FET_ParityErr 0x00000000 /* parity error */
#define FET_MasterErr 0x00000800 /* master error */
#define FET_TargetErr 0x00001000 /* target abort */
#define FET_TUNF 0x00000400 /* transmit underflow */
#define FET_ROVF 0x00000200 /* receive overflow */
#define FET_ETI 0x00000100 /* transmit early int */
#define FET_ERI 0x00000080 /* receive early int */
#define FET_CNTOVF 0x00000040 /* counter overflow */
#define FET_RBU 0x00000020 /* receive buffer unavailable */
```

freebsd-net: Driver for SURECOM EP-320X-S 100/10M Ethernet PCI Adapter

```
#define FET_TBU 0x00000010 /* transmit buffer unavailable */
#define FET_TI 0x00000008 /* transmit interrupt */
#define FET_RI 0x00000004 /* receive interrupt */
#define FET_RxErr 0x00000002 /* receive error */
/*
 * Interrupt Mask Register
 */
#define FET_MRFCON 0x00020000 /* receive flow control xon packet */
#define FET_MRFCOFF 0x00010000 /* receive flow control xoff packet */
#define FET_MLSCStatus 0x00008000 /* link status change */
#define FET_MANCStatus 0x00004000 /* autonegotiation completed */
#define FET_MFBE 0x00002000 /* fatal bus error */
#define FET_MFBEMask 0x00001800
#define FET_MTUNF 0x00000400 /* transmit underflow */
#define FET_MROVF 0x00000200 /* receive overflow */
#define FET_METI 0x00000100 /* transmit early int */
#define FET_MERI 0x00000080 /* receive early int */
#define FET_MCNTOVF 0x00000040 /* counter overflow */
#define FET_MRBU 0x00000020 /* receive buffer unavailable */
#define FET_MTBU 0x00000010 /* transmit buffer unavailable */
#define FET_MTI 0x00000008 /* transmit interrupt */
#define FET_MRI 0x00000004 /* receive interrupt */
#define FET_MRxEr 0x00000002 /* receive error */
/* 90/1/18 delete */
/* #define FET_INTRS FET_FBE|FET_MRBU|FET_TBU|FET_MTI|FET_MRI|FET_METI */
#define FET_INTRS FET_MRBU|FET_TBU|FET_MTI|FET_MRI|FET_METI
/*
 * Flow Control High/Low Threshold Register
 */
#define FET_FCHTShift 16 /* flow control high threshold */
#define FET_FLCTShift 0 /* flow control low threshold */
/*
 * BootROM/EEPROM/MII Management Register
 */
#define FET_MASK_MIIR_MII_READ 0x00000000
#define FET_MASK_MIIR_MII_WRITE 0x00000008
#define FET_MASK_MIIR_MII_MDO 0x00000004
#define FET_MASK_MIIR_MII_MDI 0x00000002
#define FET_MASK_MIIR_MII_MDC 0x00000001
/*
 * Tally Counter for CRC and MPA
 */
#define FET_TCOVF 0x80000000 /* crc tally counter overflow */
#define FET_CRCMask 0x7ff0000 /* crc number: bit 16-30 */
#define FET_CRCShift 16
#define FET_TMOV 0x00008000 /* mpa tally counter overflow */
#define FET_MPAMask 0x00007fff /* mpa number: bit 0-14 */
#define FET_MPAShift 0
/*
 * Tally Counters for transmit status
 */
```

freebsd-net: Driver for SURECOM EP-320X-S 100/10M Ethernet PCI Adapter

```
#define FET_AbortMask 0xff000000 /* transmit abort number */
#define FET_AbortShift 24
#define FET_LCoIMask 0x00ff0000 /* transmit late collisions */
#define FET_LCoIShift 16
#define FET_NCRMMask 0x0000ffff /* transmit retry number */
#define FET_NCRShift 0
/*
 * Myson TX/RX descriptor structure.
 */
struct fet_desc {
    u_int32_t fet_status;
    u_int32_t fet_ctl;
    u_int32_t fet_data;
    u_int32_t fet_next;
};
/*
 * for tx/rx descriptors
 */
#define FET_OWNByNIC 0x80000000
#define FET_OWNByDriver 0x0
/*
 * receive descriptor 0
 */
#define FET_RXOWN 0x80000000 /* own bit */
#define FET_FLNGMASK 0x0fff0000 /* frame length */
#define FET_FLNGShift 16
#define FET_MARSTATUS 0x00004000 /* multicast address received */
#define FET_BARSTATUS 0x00002000 /* broadcast address received */
#define FET_PHYSTATUS 0x00001000 /* physical address received */
#define FET_RXFSD 0x00000800 /* first descriptor */
#define FET_RXLSD 0x00000400 /* last descriptor */
#define FET_ES 0x00000080 /* error summary */
#define FET_RUNT 0x00000040 /* runt packet received */
#define FET_LONG 0x00000020 /* long packet received */
#define FET_FAE 0x00000010 /* frame align error */
#define FET_CRC 0x00000008 /* crc error */
#define FET_RXER 0x00000004 /* receive error */
#define FET_RDES0CHECK 0x000078fc /* only check MAR, BAR, PHY, ES, RUNT,
LONG, FAE, CRC and RXER bits */
/*
 * receive descriptor 1
 */
#define FET_RXIC 0x00800000 /* interrupt control */
#define FET_RBSMASK 0x000007ff /* receive buffer size */
#define FET_RBSShift 0
/*
 * transmit descriptor 0
 */
#define FET_TXERR 0x00008000 /* transmit error */
#define FET_JABTO 0x00004000 /* jabber timeout */
#define FET_CSL 0x00002000 /* carrier sense lost */
```

## freebsd-net: Driver for SURECOM EP-320X-S 100/10M Ethernet PCI Adapter

```
#define FET_LC 0x00001000 /* late collision */
#define FET_EC 0x00000800 /* excessive collision */
#define FET_UDF 0x00000400 /* fifo underflow */
#define FET_DFR 0x00000200 /* deferred */
#define FET_HF 0x00000100 /* heartbeat fail */
#define FET_NCRMASK 0x000000ff /* collision retry count */
#define FET_NCRShift 0
/*
 * tx descriptor 1
 */
#define FET_TXIC 0x80000000 /* interrupt control */
#define FET_ETIControl 0x40000000 /* early transmit interrupt */
#define FET_TXLD 0x20000000 /* last descriptor */
#define FET_TXFD 0x10000000 /* first descriptor */
#define FET_CRCDisable 0x00000000 /* crc control */
#define FET_CRCEnable 0x08000000
#define FET_PADDisable 0x00000000 /* padding control */
#define FET_PADEnable 0x04000000
#define FET_RetryTxLC 0x02000000 /* retry late collision */
#define FET_PKTShift 11 /* transmit pkt size */
#define FET_TBSMASK 0x000007ff
#define FET_TBSShift 0 /* transmit buffer size */
#define FET_MAXFRAGS 1
#define FET_RX_LIST_CNT 64
#define FET_TX_LIST_CNT 64
#define FET_MIN_FRAMELEN 60
/*
 * A transmit 'super descriptor' is actually FET_MAXFRAGS regular
 * descriptors clumped together. The idea here is to emulate the
 * multi-fragment descriptor layout found in devices such as the
 * Texas Instruments ThunderLAN and 3Com boomerang and cylone chips.
 * The advantage to using this scheme is that it avoids buffer copies.
 * The disadvantage is that there's a certain amount of overhead due
 * to the fact that each 'fragment' is 16 bytes long. In fet tests,
 * this limits top speed to about 10.5MB/sec. It should be more like
 * 11.5MB/sec. However, the upshot is that you can achieve better
 * results on slower machines: a Pentium 200 can pump out packets at
 * same speed as a PII 400.
 */
struct fet_txdesc {
struct fet_desc fet_frag[FET_MAXFRAGS];
};
#define FET_TXSTATUS(x) x->fet_ptr->fet_frag[x->fet_lastdesc].fet_status
#define FET_TXCTL(x) x->fet_ptr->fet_frag[x->fet_lastdesc].fet_ctl
#define FET_TXDATA(x) x->fet_ptr->fet_frag[x->fet_lastdesc].fet_data
#define FET_TXNEXT(x) x->fet_ptr->fet_frag[x->fet_lastdesc].fet_next
#define FET_TXOWN(x) x->fet_ptr->fet_frag[0].fet_status
#define FET_UNSENT 0x1234
struct fet_list_data {
struct fet_desc fet_rx_list[FET_RX_LIST_CNT];
struct fet_txdesc fet_tx_list[FET_TX_LIST_CNT];
```

```

};
struct fet_chain {
struct fet_txdesc *fet_ptr;
struct mbuf *fet_mbuf;
struct fet_chain *fet_nextdesc;
u_int8_t fet_lastdesc;
};
struct fet_chain_onefrag {
struct fet_desc *fet_ptr;
struct mbuf *fet_mbuf;
struct fet_chain_onefrag *fet_nextdesc;
u_int8_t fet_rlast;
};
struct fet_chain_data {
struct fet_chain_onefrag fet_rx_chain[FET_RX_LIST_CNT];
struct fet_chain fet_tx_chain[FET_TX_LIST_CNT];
struct fet_chain_onefrag *fet_rx_head;
struct fet_chain *fet_tx_head;
struct fet_chain *fet_tx_tail;
struct fet_chain *fet_tx_free;
};
struct fet_type {
u_int16_t fet_vid;
u_int16_t fet_did;
char *fet_name;
};
#define FET_FLAG_FORCEDELAY 1
#define FET_FLAG_SCHEDDELAY 2
#define FET_FLAG_DELAYTIMEO 3
struct fet_softc {
struct arpcom arpcom; /* interface info */
struct ifmedia ifmedia; /* media info */
bus_space_handle_t fet_bhandle;
bus_space_tag_t fet_btag;
struct fet_type *fet_info; /* adapter info */
struct fet_type *fet_pinfo; /* phy info */
u_int8_t fet_unit; /* interface number */
u_int8_t fet_type;
u_int8_t fet_phy_addr; /* PHY address */
u_int8_t fet_tx_pend; /* TX pending */
u_int8_t fet_want_auto;
u_int8_t fet_autoneg;
u_int16_t fet_txthresh;
caddr_t fet_ldata_ptr;
struct fet_list_data *fet_ldata;
struct fet_chain_data fet_cdata;
};
/*
 * register space access macros
 */
#define CSR_WRITE_4(sc, reg, val) \

```

## freebsd-net: Driver for SURECOM EP-320X-S 100/10M Ethernet PCI Adapter

```
bus_space_write_4(sc->fet_btag, sc->fet_bhandle, reg, val)
#define CSR_WRITE_2(sc, reg, val) \
bus_space_write_2(sc->fet_btag, sc->fet_bhandle, reg, val)
#define CSR_WRITE_1(sc, reg, val) \
bus_space_write_1(sc->fet_btag, sc->fet_bhandle, reg, val)
#define CSR_READ_4(sc, reg) \
bus_space_read_4(sc->fet_btag, sc->fet_bhandle, reg)
#define CSR_READ_2(sc, reg) \
bus_space_read_2(sc->fet_btag, sc->fet_bhandle, reg)
#define CSR_READ_1(sc, reg) \
bus_space_read_1(sc->fet_btag, sc->fet_bhandle, reg)
#define FET_TIMEOUT 1000
/*
 * General constants that are fun to know.
 */
 * PCI vendor ID
 */
#define FETVENDORID 0x1516
/*
 * FETSON device IDs.
 */
#define ID0 0x0800
#define ID1 0x0803
#define ID2 0x0891
/*
 * ST+OP+PHYAD+REGAD+TA
 */
#define FET_OP_READ 0x6000 /* ST:01+OP:10+PHYAD+REGAD+TA:Z0 */
#define FET_OP_WRITE 0x5002 /* ST:01+OP:01+PHYAD+REGAD+TA:10 */
/*
 * Constants for Myson PHY
 */
#define MysonPHYID0 0x0300
/*
 * Constants for Seeq 80225 PHY
 */
#define SeeqPHYID0 0x0016
#define SEEQ_MIIRegister18 18
#define SEEQ_SPD_DET_100 0x80
#define SEEQ_DPLX_DET_FULL 0x40
/*
 * Constants for Ahdoc 101 PHY
 */
#define AhdocPHYID0 0x0022
#define AHD0C_DiagnosticReg 18
#define AHD0C_DPLX_FULL 0x0800
#define AHD0C_Speed_100 0x0400
/*
 * Constants for Marvell 88E1000/88E1000S PHY and LevelOne PHY
 */
#define MarvellPHYID0 0x0141
```

freebsd-net: Driver for SURECOM EP-320X-S 100/10M Ethernet PCI Adapter

```
#define LevelOnePHYID0 0x0013
#define Marvell_SpecificStatus 17
#define Marvell_Speed1000 0x8000
#define Marvell_Speed100 0x4000
#define Marvell_FullDuplex 0x2000
/*
 * PCI low memory base and low I/O base register, and
 * other PCI registers. Note: some are only available on
 * the 3c905B, in particular those that related to power management.
 */
#define FET_PCI_VENDOR_ID 0x00
#define FET_PCI_DEVICE_ID 0x02
#define FET_PCI_COMMAND 0x04
#define FET_PCI_STATUS 0x06
#define FET_PCI_CLASSCODE 0x09
#define FET_PCI_LATENCY_TIMER 0x0D
#define FET_PCI_HEADER_TYPE 0x0E
#define FET_PCI_LOIO 0x10
#define FET_PCI_LOMEM 0x14
#define FET_PCI_BIOSROM 0x30
#define FET_PCI_INTLINE 0x3C
#define FET_PCI_INTPIN 0x3D
#define FET_PCI_MINGNT 0x3E
#define FET_PCI_MINLAT 0x0F
#define FET_PCI_RESETOPT 0x48
#define FET_PCI_EEPROM_DATA 0x4C
#define PHY_UNKNOWN 3
#define FET_PHYADDR_MIN 0x00
#define FET_PHYADDR_MAX 0x1F
#define PHY_BMCR 0x00
#define PHY_BMSR 0x01
#define PHY_VENID 0x02
#define PHY_DEVID 0x03
#define PHY_ANAR 0x04
#define PHY_LPAR 0x05
#define PHY_ANEXP 0x06
#define PHY_NPTR 0x07
#define PHY_LPNPR 0x08
#define PHY_1000CR 0x09
#define PHY_1000SR 0x0a
#define PHY_ANAR_NEXTPAGE 0x8000
#define PHY_ANAR_RSVD0 0x4000
#define PHY_ANAR_TLRFLT 0x2000
#define PHY_ANAR_RSVD1 0x1000
#define PHY_ANAR_RSVD2 0x0800
#define PHY_ANAR_RSVD3 0x0400
#define PHY_ANAR_100BT4 0x0200L
#define PHY_ANAR_100BTXFULL 0x0100
#define PHY_ANAR_100BTXHALF 0x0080
#define PHY_ANAR_10BTFULL 0x0040
#d
```