FUSION RESEARCH CENTER

Received by OSTI

FEB 04 1991

DOE/ER-53266-37

FRC #383

A UNIX device driver for a TransLink II Transputer Board

James C. Wiley

**Fusion Research Center
University of Texas at Austin
Austin, Texas 78712**

January 1991

THE UNIVERSITY OF TEXAS

Austin, Texas

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

ps

A UNIX device driver for a TransLink II Transputer Board

James C. Wiley

**Fusion Research Center
University of Texas at Austin
Austin, Texas 78712**

DOE/ER/53266--37

DE91 006969

Abstract

A UNIX device driver for a TransLink II Transputer board is described. A complete listing of the code is presented. The device driver allows a transputer array to be used with the A/UX operating system.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

I. Introduction

In the UNIX operating system, the actual control of physical devices is isolated from user programs. Only routines that are part of the operating system kernel are allowed to directly manipulate physical devices. These routines are called device drivers. The device driver described in this report controls a TransLink II NuBus card in an Apple Macintosh II computer running Apple's version of the UNIX operating system, A/UX. Due to the memory mapped I/O structure of the NuBus, the device's control registers appear as a few absolute memory locations in the host processor's address space. The simple nature of the physical interface and the rather straightforward operating requirements produce a model device driver that exhibits many of the essential device driver techniques. As such it serves as an intermediate example between the trivial sample driver and the more obscure extended examples supplied by Apple in their device driver kit.¹

Usually a device driver is supplied by the hardware manufacturer. In the case of the TransLink II card a driver was supplied for the Macintosh operating system but not for A/UX. The TransLink II card controls an array of transputer modules. From one to eight transputers can be supported by each TransLink II card and several cards can be placed in each Macintosh II. A transputer is a 32-bit RISC microcomputer manufactured by Inmos Ltd. with its own local memory and with high speed communication links for connecting one transputer to another. These processors were designed as building blocks for parallel processing systems. Arrays of transputers provide, at moderate cost, a powerful tool for exploring distributed memory architecture parallel computing.² The purpose of moving the transputer array to the A/UX operating system was to gain the networking and multi-tasking capabilities. Since the tasks that the transputer network was designed to execute typically run for long periods of time, a multi-tasking operating system allows other work to be accomplished on the host while the transputers operate in the background. Further, with the network connection, many users can have access to the transputer array without having to physically move the cards. The only piece missing from the A/UX-TransLink system was the A/UX device driver.

This report is organized as follows. Section II gives a general description of the TransLink II card and its control registers. Section III gives a general description of the requirements for the device driver and discusses the strategy used to implement them. The next section discusses the code whose complete listing is given in the appendix. Finally, some comments are made about the functions not supported by this driver.

II. The Device

A complete description of the TransLink II card and the transputer modules is beyond the scope of this report. The hardware manuals for each should be consulted.^{3,4} This section describes a programmer's model of the card and may not accurately reflect the actual hardware.

The TransLink II card consists of three types of devices, NuBus link adapters, a C004 link exchange, and the transputer modules. The link adapters are the devices that couple the transputer's communication links to the Macintosh's NuBus. The C004 link exchange is a crossbar switch that provides program controlled interconnections between any of the transputer links. The card has space for four transputer modules, each of which contains one or two T800 transputers along with the processor's memory. The NuBus link adapters appear as fourteen memory locations in the card's address space. Four of these locations are used to read and write to the link exchange, four are used to read and write to a transputer module, and the remaining six are used to control the card. In terms of the driver software, the device is considered to consist of two minor devices, a configuration device, and a link device. The configuration device is used to program the link exchange. The link device is used to communicate through the link exchange to one link on one of the transputer modules. The particular connection is determined by the exchange's configuration.

The names, primary direction, offsets and active bits for each of the registers are given in Table 1. The base address of the card is determined by the slot in which it is located. Each register can then be addressed as an offset to the card base address. When the system is started, the TransLink II card should be placed in a known state. This is accomplished by writing a zero to the SysReset, ConfigReset, and Analyse registers. The LinkSpeed should also be set at this time. Writing a one to the LinkSpeed register configures the link speed to 20 Mbits/sec while writing a zero sets the speed at 10 Mbits/sec. Only the 20 Mbits/sec option is used in this device driver. The whole card or just the C004 can be reset by writing a one to the reset register and then writing a zero after an appropriate delay of typically a few hundred clock cycles. The reset is accomplished on the high to low transition and becomes stable approximately 720 clock periods later.⁴

Table 1: Registers

Name	Direction	Offset	Active bits
ConfigIn	read	0x00	0-7
ConfigOut	write	0x04	0-7
ConfigInStatus	read	0x08	0-1
ConfigOutStatus	read	0x0C	0-1
LinkIn	read	0x10	0-7
LinkOut	write	0x14	0-7
LinkInStatus	read	0x18	0-1
LinkOutStatus	read	0x1C	0-1
SysReset	write	0x40	0
ConfigReset	write	0x48	0
Analyse	write	0x50	0
LinkSpeed	write	0x58	0
InterruptEnable	write	0x60	0
Error	read	0x70	0

Reading from and writing to the card is accomplished by using the In and Out registers. These registers are one byte wide. Flow control to the I/O registers is controlled through the corresponding status register. The status registers have the two lowest bits as active bits. The low bit is high when the corresponding I/O register is available. The second bit is used to control interrupts. If the status interrupt bit is set high and the InterruptEnable register has been set, then an interrupt will be generated when the I/O register is ready. Whether interrupts are enabled or not the status interrupt bit must be set before reading or writing the I/O register or a bus error will occur. The I/O can be controlled either by polling or by interrupts. The device driver presented here uses interrupts and a major portion of the logic is concerned with the interaction between the status interrupt bits and the InterruptEnable register.

III. The Driver

This section describes the general requirements for the TransLink device driver. A complete discussion of UNIX device drivers is beyond the scope of this report. The driver construction kit from Apple provides the best information and requirements for A/UX drivers.

Certain assumptions have been made about the software system that will use this driver. The system will need to place the card in an operational mode, to configure the link exchange, to load the transputer object code into the transputer modules, and then to mediate the data transfers between the host computer (the Macintosh) and the transputer array. It is assumed that either the host or the transputer can initiate a data transfer once the transputer code begins executing. (The transputer is loaded by first issuing a reset. The transputer then scans its links waiting for input. The first input to arrive is assumed to be a transputer message containing the length of the object code followed by the code itself. This code is then loaded into memory and a jump to the first location is executed. The exact structure of this message should be handled by the transputer tool set.)

Since the transputer can initiate transfers, the device driver should be capable of receiving data and storing it into a buffer before a read request is made. As long as the buffer is adequate, the transputer does not have to idle while waiting for the host to perform a read. The device driver is to operate synchronously; i.e., the system will complete the requested operation before returning to the calling program. There may be a number of identical cards in a single host, therefore the driver should be reentrant.

A UNIX driver is required to provide a certain set of routines with specific names, calling arguments, and return codes. These routines correspond to the system routines open, close, read, write, and ioctl. In the device driver, these routine names are prefixed with a unique identifying code, in this case "tl_". In addition an interrupt routine is required for card devices, and an initialization routine may be provided to perform device specific operations at start up time. The driver consists of the six routines, tl_init(), tl_open(), tl_close(), tl_read(), tl_write(), tl_int().

Device drivers are often logically divided into two parts, known as the front end and back end, or the top half and bottom half.⁵ The front or top part is responsible for accepting the operation request from the operating system and scheduling it. The back end or bottom half handles the actual interaction with the device. The communication between these two halves of the driver is usually through a set of queues or buffers and associated flags. The two halves typically operate on different time sequences and in different address spaces. The coordination between the two address spaces and the two time sequences is an essential function of the device driver.

IV. The Code

The initialization routine puts the card registers into a known state and initializes the driver state variables for each card. The open routine resets the device, allocates space for the input and output queues, and sets the in use flag. The device can only be used by a single user at a time. If a second program tries to open the device, it will receive a busy error. The close routine releases the queues and clears the in use flag. The ioctl routine can return the state of the status and error flags.

In this driver, the top half is contained in the read and write routines. The write routine copies the data from the user buffer into an output queue that is in the kernel address space. It then sets a flag and enters a sleep state until the interrupt routine, the bottom half, completes the transfer and issues a wake up. The write routine then returns to the system. The read routine operates in a similar fashion. It first checks to see if the input queue already contains enough data to satisfy the read request. If so, the data is copied from the kernel space to the user buffer. If there is not enough data, a flag is set and the sleep state is entered. When the interrupt routine has received enough data, the read flag is cleared and the read routine is awakened.

The bottom half is contained in the interrupt routine. Read interrupts are always enabled except when data is being transferred between the queues and the user buffers. When the transputer sends a byte an interrupt is triggered. The interrupt routine then accepts data until either the read buffer is full, or there is no more data, or a specified maximum number of bytes have been transferred. If the interrupt routine has the processor for too long, then performance will suffer. If there is a read request pending and there are now enough bytes in the queue to satisfy it, the read routine is awakened. The interrupt routine then checks for the write pending flag and transfers any data that is ready to be sent.

V. Comments

Several capabilities that were found in the Macintosh version of the device driver were not included in this A/UX driver. The transputer and the Macintosh have opposite byte orderings. The Macintosh driver had the ability in the read and write routines to swap byte orders depending on the data type. This function is assumed to move to the higher level software and is not supported by this driver. The Macintosh version also has the ability to query the card to find the number and type of active transputers and their memory configuration. The ability is again left to a higher software level.

Acknowledgments

The author acknowledges many useful discussions with Craig Davidson and Keith Ostrum of Quantum Leap Systems about the TransLink II card, and a useful discussion about A/UX device drivers with Rick Auricchio of Apple Computer Corporation.

Apple, Macintosh, and A/UX are registered trademarks of Apple Computer, Inc., NuBus is a trademark of Texas Instruments, TransLink II is a trademark of Levco Corp., and transputer is a trademark of Inmos Ltd.

Work supported by USDOE Grant #DE-FG05-88ER53266

References

1. Building A/UX Device Drivers, version 2.0, Apple Computer, Inc., Cupertino (1990).
2. C. Davidson, Parallel programming for Transputers, APDAlog Jan. (1988).
3. Transputer Toolset, version 88.3, Logical Systems and Levco Sales (1988).
4. IMS T800 Transputer Reference Manual, Inmos Ltd., Bristol England (1987).
5. J. Egan, T. Teixeira, Writing a UNIX Device Driver, John Wiley and Sons, Inc. New York (1988).

Appendix - Program Listings

Listing 1

```

/*****
-
- File: tl.h
-
- Purpose: A/UX device driver header for TransLink board.
-
- Author: J. C. Wiley, Oct 1990.
-
*****/

/* ioctl commands */
#include <sys/ioctl.h>
#define ERRORCK _IOR('L', 1, unsigned char)
#define CONFIGINSTATUS _IOR('L', 2, unsigned char)
#define CONFIGOUTSTATUS _IOR('L', 3, unsigned char)
#define LINKINSTATUS _IOR('L', 4, unsigned char)
#define LINKOUTSTATUS _IOR('L', 5, unsigned char)

/* additional error messages */
#define EINUREADC 129
#define EINUWTIREC 130

```

Listing 2

```

/*****
-
- File: tl.c
-
- Purpose: A/UX device driver for TransLink board.
-
- Author: J. C. Wiley, Oct 1990.
-
- Notes:
-   The translink board is considered to hold two devices a
-   C004 link adapter used for configuration of the link switch
-   and a link (24) which can be directed to any of the transputer
-   modules. The driver should only provide the mechanism for
-   reading, writing and resetting these two devices. Configuration
-   of the link adapter and communication with the transputers
-   is done at a higher software level.
-   This is a character device driver.
-   The information and sample code for writing directly to the
-   translink hardware was provided by Keith Ostrom at Quantum
-   Leap Systems. Thanks Keith.
-
- Master file:
-   id 312
-   if . include slots
-   acvs 1   tl_   -   2   1
-
*****/
#undef DEBUG
#include <sys/types.h>
#include <sys/sysmacros.h>
#include <sys/reg.h>
#include <sys/uio.h>
#include <sys/errno.h>
#include <stdio.h>
#include "tl.h"
extern int tl_addr[];

#define MAXSLOTS 6
#define SLOTOFFSET 9

/***** queue structure *****/
struct aqueue
{
    int size; /* number of bytes in queue */
    int front; /* index of front of queue, last entry */
    int rear; /* index of end of queue, first data entered */
    int avail; /* number of bytes free in queue, 0 indicates full */
    unsigned char *data; /* pointer to data */
};

typedef struct aqueue *QUEUE;

static void putQ(QUEUE q, unsigned char ch)
{
    q->data[q->front] = ch;
    q->front++;
    if(q->front >= q->size) q->front = 0;
    q->avail--;
}

```

```

    }

static unsigned char getQ(Queue q)
{
    unsigned char ch;

    ch = q->data[q->rear];
    q->rear++;
    if(q->rear >= q->size) q->rear = 0;
    q->avail++;
    return(ch);
}

static void initQ(Queue *q, int n)
{
    *q = (Queue)kmem_alloc( sizeof(**q));
    (*q)->size = n;
    (*q)->rear = 0;
    (*q)->front = 0;
    (*q)->avail = n;
    (*q)->data = (unsigned char *)getmem(n);
}

static void freeQ(Queue q)
{
    int n, m;

    n = q->size;
    releasemem(q->data, n);
    m = sizeof(*q);
    kmem_free(q, m);
}

static int availQ(Queue q)
{
    return(q->avail);
}

static int levelQ(Queue q)
{
    return(q->size - q->avail);
}

static int sizeQ(Queue q)
{
    return(q->size);
}

/*****

/* offsets to hardware registers from board base address, which depends
   on the slot */
/* Note that bit 1 of the status registers is used for interrupt enables */

/* General Offsets */
#define InReg      0x00
#define OutReg     0x04
#define InStatus   0x08
#define OutStatus  0x0C
#define ConfigOffset 0x00
#define LinkOffset 0x10

/* C004 registers */

```

```

#define ConfigIn      0x00
#define ConfigOut     0x04
#define ConfigInStatus 0x08
#define ConfigOutStatus 0x0C
#define ConfigReset   0x48

/* Link registers - link 24*/
#define LinkIn        0x10
#define LinkOut       0x14
#define LinkInStatus  0x18
#define LinkOutStatus 0x1C
#define LinkReset     0x40

/* Board registers */
#define LinkSpeed      0x58
#define Analyse        0x50
#define InterruptEnable 0x60
#define ErrorRegister  0x70

/* Offsets */
#define BoardOffset    0xF0000003
#define LINKMULT       24
#define PHYSBASE(slot) (unsigned char *)((unsigned int)BoardOffset+\
                                           (slot<<LINKMULT))

/* Queue sizes and transfer size */
#define INQUEUESIZE 1024
#define OUTQUEUESIZE 1024
#define MAXTRANSFER 50

/* status structure */
struct astatus{
    int sConfigDevOpen;
    int sLinkDevOpen;
    int sReadPending;
    int sReadSize;
    int sWritePending;
    int sWriteSize;
    int sWriteOffset;
    QUEUE sQin;
    QUEUE sQout;
};
typedef struct astatus STATUS;

/* static variables */
static STATUS ss[MAXSLOTS];

enum {
    Config=0,
    Link=1
};

/*****
-
- init - initializes the board. Called at start up.
-
*****/
int tl_init()
{
    int slot;
    unsigned char *boardbase;
    STATUS *ssp;

```

```

    slot = tl_addr[0];
    boardbase = PHYSBASE(slot);
    ssp = &ss[slot - SLOTOFFSET];

    printf("tl_init: v3.0 TransLink board in slot %d \n", slot);

    /* put board in known state */
    *(boardbase + ConfigReset) = 0;
    *(boardbase + LinkReset) = 0;
    *(boardbase + Analyse) = 0;
    *(boardbase + InterruptEnable) = 0;
    *(boardbase + LinkSpeed) = 1; /* set links to 20 mbps */
    delay(5);

    /* set static variables */
    ssp->sConfigDevOpen=0;
    ssp->sLinkDevOpen = 0;
    ssp->sReadPending = 0;
    ssp->sWritePending = 0;
    ssp->sQin = NULL;
    ssp->sQout = NULL;
    return(0);
}

/*****
-
- open
-     dev - device number, encodes both major and minor
-     flag - corresponds to oflag in open(2) call.
-     ndevp - used by streams.
-
-     Sets sDevOpen flag;
-
*****/
int tl_open(dev_t dev, int flag, dev_t *ndevp)
{
    int min, slot;
    unsigned char *boardbase;
    STATUS *ssp;

    min = minor(dev);
    slot = tl_addr[0];
    boardbase = PHYSBASE(slot);
    ssp = &ss[slot - SLOTOFFSET];

#ifdef DEBUG
    printf(" in tl_open minor %d, slot = %d\n", min, slot);
#endif
    switch(min)
    {
        case Config:
            if(ssp->sConfigDevOpen == 1) return(EBUSY);
            *(boardbase + ConfigReset) = 1; /* hi to low transition resets */
            delay(5);
            *(boardbase + ConfigReset) = 0;
            delay(5);
            *(boardbase + ConfigInStatus) = 2; /* set bit 2 for interrupt */
            *(boardbase + ConfigOutStatus) = 0;
            ssp->sConfigDevOpen = 1;
            if(ssp->sQin == NULL) initQ(&ssp->sQin, INQUEUESIZE);
            if(ssp->sQout == NULL) initQ(&ssp->sQout, OUTQUEUESIZE);
            break;
    }

```

```

        case Link:
            if(ssp->sLinkDevOpen == 1) return(EBUSY);
            *(boardbase + LinkReset) = 1; /* hi to low transition resets */
            delay(5);
            *(boardbase + LinkReset) = 0;
            delay(5);
            *(boardbase + LinkInStatus) = 2; /* set bit 2 for interrupt */
            *(boardbase + LinkOutStatus) = 0;
            ssp->sLinkDevOpen = 1;
            if(ssp->sQin == NULL) initQ(&ssp->sQin, INQUEUEESIZE);
            if(ssp->sQout == NULL) initQ(&ssp->sQout, OUTQUEUEESIZE);
            break;
        default:
            return(ENODEV);
    }
    *(boardbase + InterruptEnable) = 1;
    return(0);
}

/*****
-
- close
-     dev - device number.
-     flag - from oflag in
-
-     Clears sDevOpen flag.
-
*****/
int tl_close(dev_t dev, int flag)
{
    int min, slot;
    STATUS *ssp;

    min = minor(dev);
    slot = tl_addr[0];
    ssp = &ss[slot - SLOTOFFSET];
#ifdef DEBUG
    printf("in tl_close minor %d, slot = %d\n", min, slot);
#endif
    switch(min)
    {
        case Config:
            ssp->sConfigDevOpen = 0;
            if( ssp->sLinkDevOpen == 0 && ssp->sQin != NULL)
            {
                freeQ(ssp->sQin);
                ssp->sQin = NULL;
            }
            if( ssp->sLinkDevOpen == 0 && ssp->sQout != NULL)
            {
                freeQ(ssp->sQout);
                ssp->sQout = NULL;
            }
            break;
        case Link:
            ssp->sLinkDevOpen = 0;
            if( ssp->sConfigDevOpen == 0 && ssp->sQin != NULL)
            {
                freeQ(ssp->sQin);
                ssp->sQin = NULL;
            }
            if( ssp->sConfigDevOpen == 0 && ssp->sQout != NULL)
            {

```

```

        freeQ(ssp->sQout);
        ssp->sQout = NULL;
    }
    break;
default:
    return(ENODEV);
}
return(0);
}

/*****
-
- int - handles interrupts from slots
-     args described in <sys/reg.h>, a_dev field has slot number.
-
*****/
void tl_int(struct args *args)
{
    int slot, s, i;
    unsigned char *base, *boardbase, ch;
    STATUS *ssp;

    slot = args->a_dev;
    boardbase = PHYSBASE(slot);
    ssp = &ss[slot - SLOTOFFSET];

#ifdef DEBUG
    printf("tl_int: Begin slot = %d\n", slot);
#endif

    /* check for input characters */
    base = 0;
    if( (*(boardbase + LinkOffset + InStatus) & 0x01) == 1)
    {
        base = boardbase + LinkOffset;
    }
    else if( (*(boardbase + ConfigOffset + InStatus) & 0x01) == 1)
    {
        base = boardbase + ConfigOffset;
    }

    if(base != 0) /* characters to read */
    {
        i = 0;
        while( (*(base + InStatus) & 0x01) == 1 /* character ready */
            && availQ(ssp->sQin) > 0 /* queue not full */
            && i++ < MAXTRANSFER) /* not too many now */
        {
            putQ(ssp->sQin, *(base+InReg)); /* get byte from board */
        }

        if( (ssp->sReadPending == 1) && (levelQ(ssp->sQin) >= ssp->sReadSize))
        {
            *(boardbase + InterruptEnable) = 0; /* turn off interrupts */
            ssp->sReadPending = 0;
            wakeup(&ssp->sReadPending); /* restart front end */
        }
    }

    /***** write section *****/

    if(ssp->sWritePending == 1)
    {

```

```

        base = (unsigned char *) (boardbase + ssp->sWriteOffset);
        i = 0;
        while( ((* (base+OutStatus) & 0x01) ==1) /* board ready */
                && levelQ(ssp->sQout) > 0 /* characters to send */
                && ssp->sWriteSize > 0 /* in this block */
                && i++ < MAXTRANSFER) /* not too many now */
        {
            *(base + OutReg) = getQ(ssp->sQout); /* send byte to board */
            ssp->sWriteSize--;
        }

        if(ssp->sWritePending == 1 && ssp->sWriteSize == 0) /* done? */
        {
            ssp->sWritePending = 0;
            *(boardbase + InterruptEnable) = 0; /* turn off interrupts */
            wakeup(&ssp->sWritePending); /* restart front end */
        }

/*
- if input buffer is full and no writes scheduled, then turn off
- interrupts or driver will continue to interrupt and
- do nothing.
*/
        if( (ssp->sWritePending == 0) && (availQ(ssp->sQin) == 0))
            *(boardbase + InterruptEnable) = 0;
#ifdef DEBUG
        printf("tl_int: end\n");
#endif
    }

/*****
-
- read
- dev - device number
- uio - pointer to uio structure see <sys/uio.h>
-
*****/
int tl_read(dev_t dev, struct uio *uio)
{
    int slot, s, qsize;
    int i, n;
    unsigned char *base, *boardbase;
    STATUS *ssp;

#ifdef DEBUG
    printf("tl_read: begin \n");
#endif

    slot = tl_addr[0];
    boardbase = PHYSBASE(slot);
    ssp = &ss[slot - SLOTOFFSET];

    n = (uio->uio_iov)->iiov_len; /* length of data */
    qsize = sizeQ(ssp->sQin);
    do
    {
        s = spl2();
        *(boardbase+InterruptEnable) = 0; /* turn off board interrupts */
        ssp->sReadSize = (n < qsize)? n: qsize;
        n -= ssp->sReadSize;
        while( levelQ(ssp->sQin) < ssp->sReadSize)
        {
            ssp->sReadPending=1; /* set flag for bottom half */

```



```

        *(boardbase+InterruptEnable) = 1; /* turn on board interrupts */
        sleep(&ssp->sReadPending);
    }
    *(boardbase+InterruptEnable) = 0;
    splx(s);

    /* transfer data from local buffer to user */
    for(i=0; i<ssp->sReadSize; i++)
    {
        if( "readc( getQ(ssp->sQin), uio) == -1) return(EINUREADC);
    }
    while( n>0 );

    *(boardbase + InterruptEnable) = 1; /* turn on board interrupts */

#ifdef DEBUG
    printf("tl_read: end\n");
#endif
    return(0);
}

/*****
-
- write
-     dev - device number
-     uio - pointer to uio structure see <sys/uio.h>
-
*****/
int tl_write(dev_t dev, struct uio *uio)
{
    int min, slot, s, qsize;
    unsigned char *base, *boardbase;
    int i, n, ch;
    STATUS *ssp;

#ifdef DEBUG
    printf("tl_write: begin \n");
#endif
    min = minor(dev);

    /* hardware base */
    slot = tl_addr[0];
    boardbase = PHYSBASE(slot);
    ssp = &ss[slot - SLOTOFFSET];
    switch(min) /* which device? */
    {
        case Config:
            base = (unsigned char *) (boardbase+ConfigOffset);
            ssp->sWriteOffset = ConfigOffset;
            break;

        case Link:
            base = (unsigned char *) (boardbase+LinkOffset);
            ssp->sWriteOffset = LinkOffset;
            break;

        default:
            return(ENODEV); /* Error */
            break;
    }

    n = (uio->uio_iov)->iiov_len; /* size of transfer */
    do
    {
        *(boardbase+InterruptEnable) = 0; /* turn off board interrupts */

```

```

        qsize = availQ(ssp->sQout);
        ssp->sWriteSize = (n < qsize)? n : qsize;
        n -= ssp->sWriteSize;
        for(i=0; i< ssp->sWriteSize; i++) /* transfer data to output queue */
        {
            if( (ch = uwritec(uio)) == -1 ) return(EINUWTIREC);
            putQ(ssp->sQout, (unsigned char)ch);
        }

        s = spl2();
        while(ssp->sWriteSize > 0)
        {
            *(base + OutStatus) = 2;
            ssp->sWritePending = 1;
            *(boardbase+InterruptEnable) = 1;
            sleep(&ssp->sWritePending);
        }
        *(base+OutStatus) = 0;
        splx(s);
    } while( n > 0 );

    *(boardbase+InterruptEnable) = 1; /* turn on board interrupts */

#ifdef DEBUG
    printf("tl_write: end, length = %d \n",n);
#endif
    return(0);
}

/*****
-
- ioctl
-     dev - device number
-     cmd - corresponds to request parameter in ioctl(2)
-     addr - address of arguments copied from ioctl(2)
-     mode - contains values set when device open (reading or writing)
-
*****/
int tl_ioctl(dev_t dev, int cmd, caddr_t addr, int mode)
{
    int slot;
    unsigned char *boardbase;
    STATUS *ssp;

    slot = tl_addr[0];
    boardbase = PHYSBASE(slot);
    ssp = &ss[slot - SLOTOFFSET];
    switch(cmd)
    {
        case ERRORCK:
            *addr = *(boardbase + ErrorRegister) & 0x01;
            break;
        case CONFIGINSTATUS:
            if( (*(boardbase + ConfigInStatus) & 0x01) == 1 ||
                levelQ(ssp->sQin) > 0)
            {
                *addr = 1;
            }
            else *addr = 0;
            break;
        case CONFIGOUTSTATUS:
            *addr = *(boardbase + ConfigOutStatus) & 0x01;
            break;
    }
}

```

```

case LINKINSTATUS:
    if( (*(boardbase + LinkInStatus) & 0x01) == 1 ||
        levelQ(ssp->sQin) > 0)
    {
        *addr = 1;
    }
    else *addr = 0;
    break;
case LINKOUTSTATUS:
    *addr = *(boardbase + LinkOutStatus) & 0x01;
    break;
default:
    return(ENOMSG);
    break;
}
return(0);
}

```

END

DATE FILMED

02 / 14 / 91