```
#       @(#)70.mk       2.15

VERSION = 70

LIB = lib2.$(VERSION).a
COMPOOL=

LIB2OBJS = \
        $(LIB)(mx1.o) \
        $(LIB)(mx2.o) \
        $(LIB)(bio.o) \
        $(LIB)(tty.o) \
        $(LIB)(malloc.o) \
        $(LIB)(pipe.o) \
        $(LIB)(dhdm.o) \
        $(LIB)(dh.o) \
        $(LIB)(dhfdm.o) \
        $(LIB)(dj.o) \
        $(LIB)(dn.o) \
        $(LIB)(ds40.o) \
        $(LIB)(dz.o) \
        $(LIB)(err.o) \
        $(LIB)(alarm.o) \
        $(LIB)(hf.o) \
        $(LIB)(hps.o) \
        $(LIB)(hpmap.o) \
        $(LIB)(hp45.o) \
        $(LIB)(hs.o) \
        $(LIB)(ht.o) \
        $(LIB)(jy.o) \
        $(LIB)(kl.o) \
        $(LIB)(lfh.o) \
        $(LIB)(lp.o) \
        $(LIB)(mem.o) \
        $(LIB)(nmpipe.o) \
        $(LIB)(rf.o) \
        $(LIB)(rk.o) \
        $(LIB)(rp.o) \
        $(LIB)(rx.o) \
        $(LIB)(sys.o) \
        $(LIB)(trans.o) \
        $(LIB)(ttdma.o) \
        $(LIB)(tec.o) \
        $(LIB)(tex.o) \
        $(LIB)(tm.o) \
        $(LIB)(vp.o) \
        $(LIB)(vs.o) \
        $(LIB)(vtlp.o) \
        $(LIB)(vtll.o) \
        $(LIB)(fakevtlp.o) \
        $(LIB)(vt61.o) \
        $(LIB)(vt100.o) \
        $(LIB)(vtmon.o) \
        $(LIB)(vtdbg.o) \
        $(LIB)(vtutil.o) \
```

```
        $(LIB)(vtast.o) \
        $(LIB)(partab.o) \
        $(LIB)(rh.o) \
        $(LIB)(devstart.o) \
        $(LIB)(dmc11.o) \
        $(LIB)(rop.o) \
        $(LIB)(ioctl.o) \
        $(LIB)(fakemx.o)

all:    $(LIB)
        @echo $(LIB) is now up-to-date.

$(LIB)::        $(LIB2OBJS)

$(LIB2OBJS):    $(FRC)

FRC:
        rm -f $(LIB)

clobber: cleanup
        -rm -f $(LIB) *.o

clean cleanup:

install:        all

.PRECIOUS:      $(LIB)

.s.a:
        $(AS) $(ASFLAGS) -o $*.o $<
        ar rcv $@ $*.o
        rm $*.o
```

```
#       @(#)Makefile    2.7

70;
all;:   70

.DEFAULT:
        make -f $<.mk
```

```
/*      @(#)alarm.c     2.6     */

#
/*.n'alarm'Includes, Defines, and Data Declarations'*/

/*
 *      Alarm Panel Driver (BD04).
 *
 *      written by:     j. c. kaufeld
 *                      CB 2C249 x4522
 *
 *      initial version 04/13/77
 *      update: t. j. cook
 *              CB 2C223 x4409
 *              05/24/78
 *              : fix to prevent race condition in setting major alarm
 */

#include "sys/param.h"
#include "sys/conf.h"
#include "sys/confx.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/systm.h"
#include "sys/reg.h"

#define ALMADDR         0164410
#define ALPERIOD        60      /* alarm time increment in clock ticks */
#define ALDELAY         1       /* delay between reset and major alarm set */

#define ALM_RESET       0
#define ALM_MINOR       1
#define ALM_MAJOR       2

/*
 *      BD04 Hardware register layout.
 */

struct alarm {
        int majr;               /* major alarm */
        int minr;               /* minor alarm */
        int reset;              /* reset major/minor alarms (turn them off) */
        int retrigger;          /* watchdog trigger */
};

/*
 *      Software control flags.
 */

struct alarms {
        int almflag;            /* flags (see defines below) */
        int mintime;            /* minor alarm time */
} all;

#define ALMWDOG 1               /* watchdog timer running */
```

```
#define ALMOPEN 2                /* alarms opened */
#define ALMAJOR 4                /* major alarm flag */
/*.s'almopen'Open alarm system (enables watchdog)'*/
almopen(dev,flag)
{
        if((all.almflag&ALMWDOG) == 0) {
                all.almflag =| ALMWDOG;
                spl5();
                almreset(0);
                spl0();
        }
        all.almflag =| ALMOPEN;
}


/*.t'almclose'Close alarm system (turns off alarms)'*/
almclose(dev,flag)
{
        all.almflag =& ~(ALMOPEN|ALMAJOR);
        all.mintime =  0;
}


/*.t'almreset'Alarm watchdog'*/
static int almreset(flag)
                                 /* flag = 1: set major alarm    */
{
        ALMADDR->retrigger = 0;
        if(flag)
                {
                ALMADDR->majr = 0;
                goto alreturn;
                }
        if(all.mintime == 0 || all.mintime-- == 0)
                ALMADDR->reset = 0;
        else
                ALMADDR->minr = 0;

        if(all.almflag&ALMAJOR)
                timeout(almreset,1,ALDELAY);
        else
alreturn:
                timeout(almreset,0,ALPERIOD);
}
/*.s'almioctl'Alarm Functions'*/
almioctl(dev, cmd, addr, flg)
register cmd, addr;
{

        switch(cmd) {

        case ALM_RESET:
                all.almflag =& ~ALMAJOR;
                all.minr    =  0;
                return;
```

```
        case ALM_MINOR:
                all.minr = addr;
                return;

        case ALM_MAJOR:
                all.almflag =| ALMAJOR;
                return;

        default:
                u.u_error = EINVAL;
                return;
        }
}
```

```
/*      @(#)bio.c       2.10    */

/*
 *      Copyright 1973 Bell Telephone Laboratories Inc
 *
 *      UNIX SUPPORT GROUP MODIFICATION 2.33
 *
 */

#include "sys/param.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/buf.h"
#include "sys/bufx.h"
#include "sys/conf.h"
#include "sys/confx.h"
#include "sys/systm.h"
#include "sys/proc.h"
#include "sys/procx.h"
#include "sys/seg.h"
#include "sys/sysmes.h"
#include "sys/sysmesx.h"
#include "sys/vtmn.h"
#include "sys/maus.h"
#include "sys/elog.h"
#include "sys/iobuf.h"

struct  buf     swbuf1;
struct  buf     swbuf2;

/*
 * Allocation of buffers proper.
 */
char    sabuf[NBUF][BSIZE+BSLOP];               /* Addressable buffers */

#define b_acount        b_resid

char    rathresh        1;      /*  Must have more than this
                                 *  number of buffers on freelist
                                 *  before read-ahead will occur.
                                 */

/*
 * The following several routines allocate and free
 * buffers with various side effects.  In general the
 * arguments to an allocate routine are a device and
 * a block number, and the value is a pointer to
 * to the buffer header; the buffer is marked "busy"
 * so that no on else can touch it.  If the block was
 * already in core, no I/O need be done; if it is
 * already busy, the process waits until it becomes free.
 * The following routines allocate a buffer:
 *      getblk  (macro which expands to call to aagetblk)
 *      agetblk (macro which expands to call to aagetblk)
 *      aagetblk
```

```
 *      getablk
 *      bread   (macro which expands to call to aabread)
 *      abread  (macro which expands to call to aabread)
 *      aabread
 *      breada
 * Eventually the buffer must be released, possibly with the
 * side effect of writing it out, by using one of
 *      bwrite
 *      bdwrite
 *      bawrite
 *      brelse
 */

/*
 * Read in (if necessary) the block and return a buffer pointer
 * to a buffer which will be directly addressable if flag is nonzero.
 */
struct buf *
aabread(dev, blkno, flag)
dev_t dev;
daddr_t blkno;
int flag;
{
        register struct buf *bp;

        bp = aagetblk(dev, blkno, flag);
        if (bp->b_flags&B_DONE)
                return(bp);
        bp->b_flags =| B_READ;
        bp->b_bcount = BSIZE;
        (*bdevsw[major(dev)].d_strategy)(bp);
        meas.m_dread++;
        u.u_dread++;
        iowait(bp);
        return(bp);
}

/*
 * Read in the block, like bread, but also start I/O on the
 * read-ahead block (which is not allocated to the caller)
 */
struct buf *
breada(dev, blkno, rablkno)
register dev_t dev;
daddr_t blkno, rablkno;
{
        register struct buf *bp, *rabp;

        bp = NULL;
        if (!incore(dev, blkno)) {
                bp = getblk(dev, blkno);
                if ((bp->b_flags&B_DONE) == 0) {
                        bp->b_flags =| B_READ;
                        bp->b_bcount = BSIZE;
                        (*bdevsw[major(dev)].d_strategy)(bp);
                        meas.m_dread++;
```

```
                        u.u_dread++;
                }
        }
        if (rablkno && bfreelist.b_bcount>rathresh && !incore(dev, rablkno)) {
                rabp = getblk(dev, rablkno);
                if (rabp->b_flags & B_DONE)
                        brelse(rabp);
                else {
                        rabp->b_flags |= B_READ|B_ASYNC;
                        rabp->b_bcount = BSIZE;
                        (*bdevsw[major(dev)].d_strategy)(rabp);
                        meas.m_dread++;
                        u.u_dread++;
                }
        }
        if (bp == NULL)
                return(bread(dev, blkno));
        iowait(bp);
        return(bp);
}

/*
 * Write the buffer, waiting for completion.
 * Then release the buffer.
 */
bwrite(bp)
register struct buf *bp;
{
        register flag;

        flag = bp->b_flags;
        if((flag & B_DELWRI) == 0)
                u.u_dwrit++;
        bp->b_flags &= ~(B_READ | B_DONE | B_ERROR | B_DELWRI | B_AGE);
        bp->b_bcount = BSIZE;
        (*bdevsw[major(bp->b_dev)].d_strategy)(bp);
        meas.m_dwrit++;
        if ((flag&B_ASYNC) == 0) {
                iowait(bp);
                brelse(bp);
        } else if (flag&B_DELWRI)
                bp->b_flags |= B_AGE;
        else
                geterror(bp);
}

/*
 * Release the buffer, marking it so that if it is grabbed
 * for another purpose it will be written out before being
 * given up (e.g. when writing a partial block where it is
 * assumed that another write for the same block will soon follow).
 * This can't be done for magtape, since writes must be done
 * in the same order as requested.
 */
bdwrite(bp)
register struct buf *bp;
```

```
{
        register struct iobuf *dp;

        dp = bdevsw[major(bp->b_dev)].d_tab;
        if (dp->b_flags & B_TAPE)
                bawrite(bp);
        else {
                if((bp->b_flags&B_DELWRI) == 0)
                        u.u_dwrit++;
                bp->b_flags |= B_DELWRI | B_DONE;
                bp->b_resid = 0;
                brelse(bp);
        }
}

/*
 * Release the buffer, start I/O on it, but don't wait for completion.
 */
bawrite(bp)
register struct buf *bp;
{

        if (bfreelist.b_bcount>4)
                bp->b_flags |= B_ASYNC;
        bwrite(bp);
}

/*
 * release the buffer, with no I/O implied.
 */
brelse(bp)
register struct buf *bp;
{
        register struct buf **backp;
        register s;

        if (bp->b_flags&B_WANTED)
                wakeup((caddr_t)bp);
        if (bfreelist.b_flags&B_WANTED) {
                bfreelist.b_flags &= ~B_WANTED;
                wakeup((caddr_t)&bfreelist);
        }
        if (bp->b_flags&B_ERROR) {
                bp->b_flags |= B_STALE|B_AGE;
                bp->b_flags &= ~(B_ERROR|B_DELWRI);
                bp->b_error = 0;
        }
        s = spl6();
        if(bp->b_flags & B_AGE) {
                backp = &bfreelist.av_forw;
                (*backp)->av_back = bp;
                bp->av_forw = *backp;
                *backp = bp;
                bp->av_back = &bfreelist;
        } else {
                backp = &bfreelist.av_back;
```

```
                (*backp)->av_forw = bp;
                bp->av_back = *backp;
                *backp = bp;
                bp->av_forw = &bfreelist;
        }
        bp->b_flags &= ~(B_WANTED|B_BUSY|B_ASYNC|B_AGE);
        bfreelist.b_bcount++;
        if (bp->b_paddr < (paddr_t)(unsigned)&sabuf[NBUF][0])
                bfreelist.b_acount++;
        splx(s);
}

/*
 * See if the block is associated with some buffer
 * (mainly to avoid getting hung up on a wait in breada)
 */
incore(dev, blkno)
dev_t dev;
register daddr_t blkno;
{
        register struct buf *bp;
        register struct iobuf *dp;

        dp = bdevsw[major(dev)].d_tab;
        if (dp->b_flags&B_QHASH)
                dp = &(((qhsh_t *)(dp->b_forw))[blkno&LB2QSZ]);
        for (bp=dp->b_forw; bp != dp; bp = bp->b_forw)
                if (bp->b_blkno==blkno && bp->b_dev==dev && (bp->b_flags&B_STAL
*  E)==0)
                        return(1);
        return(0);
}

/*
 * Assign a buffer for the given block.  If the appropriate
 * block is already associated, return it; otherwise search
 * for the oldest non-busy buffer and reassign it.
 * If flag is nonzero the buffer will be directly addressable.
 */
struct buf *
aagetblk(dev, blkno, flag)
dev_t dev;
register daddr_t blkno;
int flag;
{
        register struct buf *bp;
        register struct iobuf *dp;

        meas.m_gbcnt++;
        if (flag)
                meas.m_agbcnt++;
        u.u_gbcnt++;
        if(major(dev) >= nblkdev)
                panic("blkdev");

    loop:
```

```
        spl0();
        dp = bdevsw[major(dev)].d_tab;
        if(dp == NULL)
                panic("devtab");
        if (dp->b_flags&B_QHASH)
                dp = &(((qhsh_t *)(dp->b_forw))[blkno&LB2QSZ]);
        for (bp=dp->b_forw; bp != dp; bp = bp->b_forw) {
                if (bp->b_blkno!=blkno || bp->b_dev!=dev ||
                   (bp->b_flags&B_STALE))
                        continue;
                spl6();
                if (bp->b_flags&B_BUSY) {
                        bp->b_flags |= B_WANTED;
                        sleep((caddr_t)bp, PRIBIO+1);
                        goto loop;
                }
                notavail(bp);
                spl0();
                if (flag && bp->b_paddr >= (paddr_t)(unsigned)&sabuf[NBUF][0]){
                        register struct buf *abp;
                        register paddr_t save;

                        abp = getablk(1);
                        copyio(bp->b_paddr,(caddr_t)abp->b_paddr,BSIZE,U_RKD);
                        save = bp->b_paddr;
                        bp->b_paddr = abp->b_paddr;
                        abp->b_paddr = save;
                        brelse(abp);
                }
                meas.m_fndblk++;
                return(bp);
        }
        spl6();
        if (bfreelist.av_forw == &bfreelist ||
            (flag && bfreelist.b_acount == 0)) {
                bfreelist.b_flags |= B_WANTED;
                sleep((caddr_t)&bfreelist, PRIBIO+1);
                meas.m_noblk++;
                if (flag)
                        meas.m_anoblk++;
                goto loop;
        }
        for (bp=bfreelist.av_forw; flag; bp=bp->av_forw)
                if (bp->b_paddr < (paddr_t)(unsigned)&sabuf[NBUF][0])
                        break;
        notavail(bp);
        spl0();
        if (bp->b_flags & B_DELWRI) {
                bp->b_flags |= B_ASYNC;
                bwrite(bp);
                goto loop;
        }
        bp->b_flags = B_BUSY;
        bp->b_back->b_forw = bp->b_forw;
        bp->b_forw->b_back = bp->b_back;
        bp->b_forw = dp->b_forw;
```

```
        bp->b_back = dp;
        dp->b_forw->b_back = bp;
        dp->b_forw = bp;
        bp->b_dev = dev;
        bp->b_blkno = blkno;
        return(bp);
}

/*
 * get an empty block,
 * not assigned to any particular device
 * and if flag non-zero, directly addressable
 */
struct buf *
getablk(flag)
{
        register struct buf *bp;
        register struct buf *dp;

        meas.m_gbcnt++;
        if(flag)
                meas.m_agbcnt++;
        u.u_gbcnt++;
loop:
        dp = &bfreelist;
        spl6();
        while (dp->av_forw == dp ||
                (flag && dp->b_acount == 0)) {
                dp->b_flags |= B_WANTED;
                sleep((caddr_t)dp, PRIBIO+1);
                meas.m_noblk++;
                if (flag)
                        meas.m_anoblk++;
        }
        for (bp = dp->av_forw; flag; bp = bp->av_forw)
                if (bp->b_paddr < (paddr_t)(unsigned)&sabuf[NBUF][0])
                        break;
        notavail(bp);
        spl0();
        if (bp->b_flags & B_DELWRI) {
                bp->b_flags |= B_ASYNC;
                bwrite(bp);
                goto loop;
        }
        bp->b_flags = B_BUSY|B_AGE;
        bp->b_back->b_forw = bp->b_forw;
        bp->b_forw->b_back = bp->b_back;
        bp->b_forw = dp->b_forw;
        bp->b_back = dp;
        dp->b_forw->b_back = bp;
        dp->b_forw = bp;
        bp->b_dev = (dev_t)NODEV;
        return(bp);
}

/*
```

```
 * Wait for I/O completion on the buffer; return errors
 * to the user.
 */
iowait(bp)
register struct buf *bp;
{

        spl6();
        while ((bp->b_flags&B_DONE)==0)
                sleep((caddr_t)bp, PRIBIO);
        spl0();
        geterror(bp);
}

/*
 * Unlink a buffer from the available list and mark it busy.
 * (internal interface)
 */
notavail(bp)
register struct buf *bp;
{

        bp->av_back->av_forw = bp->av_forw;
        bp->av_forw->av_back = bp->av_back;
        bp->b_flags |= B_BUSY;
        bfreelist.b_bcount--;
        if (bp->b_paddr < (paddr_t)(unsigned)&sabuf[NBUF][0])
                bfreelist.b_acount--;
}

/*
 * Mark I/O complete on a buffer, release it if I/O is asynchronous,
 * and wake up anyone waiting for it.
 */
iodone(bp)
register struct buf *bp;
{

        if(bp->b_flags&B_MAP)
                mapfree(bp);
        bp->b_flags |= B_DONE;
        if (bp->b_flags&B_ASYNC)
                brelse(bp);
        else {
                bp->b_flags &= ~B_WANTED;
                wakeup(bp);
        }
}

/*
 * Initialize the buffer I/O system by freeing
 * all buffers and setting all device buffer lists to empty.
 */
binit()
{
        register struct buf *bp;
```

```
        register struct buf *dp;
        register int i;
        int j;
        struct bdevsw *bdp;
        paddr_t nbase;

        dp = &bfreelist;
        dp->b_forw = dp->b_back =
            dp->av_forw = dp->av_back = dp;
        nbase = bufbase;
        for (i=0, bp=buf; i<NBUF+NXBUF; i++, bp++) {
                bp->b_dev = NODEV;
                if (i<NBUF)
                        bp->b_paddr = (paddr_t)sabuf[i];
                else {
                        bp->b_paddr = nbase;
                        nbase += BSIZE;
                }
                bp->b_back = dp;
                bp->b_forw = dp->b_forw;
                dp->b_forw->b_back = bp;
                dp->b_forw = bp;
                bp->b_flags = B_BUSY;
                bp->b_bcount = BSIZE;
                brelse(bp);
        }
        for (i=0, bdp=bdevsw; i<nblkdev; bdp++, i++) {
                dp = bdp->d_tab;
                if(dp) {
                        if ((dp->b_flags&B_OHASH) == 0) {
                                dp->b_forw = dp;
                                dp->b_back = dp;
                        }
                }
        }
        hinit();
}

/*
 * swap I/O
 */
swap(blkno, coreaddr, count, rdflg)
unsigned coreaddr;
{
        register struct buf *bp;

        meas.m_swap++;
        bp = &swbuf1;
        if(bp->b_flags & B_BUSY)
                if((swbuf2.b_flags&B_WANTED) == 0)
                        bp = &swbuf2;
        spl6();
        while (bp->b_flags&B_BUSY) {
                bp->b_flags |= B_WANTED;
                sleep((caddr_t)bp, PSWP+1);
        }
```

```
        bp->b_flags = B_BUSY | B_PHYS | rdflg;
        bp->b_dev = swapdev;
        bp->b_bcount = ctob(count);
        bp->b_blkno = blkno;
        bp->b_paddr = (paddr_t)coreaddr << 6;
        (*bdevsw[swapdev>>8].d_strategy)(bp);
        if(rdflg == 0) {
                meas.m_dwrit++;
                u.u_dwrit++;
        } else {
                meas.m_dread++;
                u.u_dread++;
        }
        spl6();
        while((bp->b_flags&B_DONE)==0)
                sleep((caddr_t)bp, PSWP);
        if (bp->b_flags&B_WANTED)
                wakeup((caddr_t)bp);
        spl0();
        bp->b_flags &= ~(B_BUSY|B_WANTED);
        if (bp->b_flags & B_ERROR)
                panic("IO err in swap");
}

/*
 * make sure all write-behind blocks
 * on dev (or NODEV for all)
 * are flushed out.
 * (from umount and update)
 */
bflush(dev)
dev_t dev;
{
        register struct buf *bp;

loop:
        spl6();
        for (bp = bfreelist.av_forw; bp != &bfreelist; bp = bp->av_forw) {
                if (bp->b_flags&B_DELWRI && (dev == NODEV||dev==bp->b_dev)) {
                        bp->b_flags |= B_ASYNC;
                        notavail(bp);
                        bwrite(bp);
                        goto loop;
                }
        }
        spl0();
}

/*
 * Raw I/O. The arguments are
 *      The strategy routine for the device
 *      The device number
 *      Read/write flag
 *      The number of blocks on the logical device.
 * Essentially all the work is computing physical addresses and
 * validating them.
```

```
 */
physio(strat, dev, rw, nblocks)
int (*strat)();
unsigned nblocks;
{
        register struct buf *bp;
        register unsigned base;
        register int nb;
        int ts;
        unsigned off;

        /*
         * Check to see if a partial read should be made for end of a
         * logical device;
         */
        off = u.u_offset>>BSHIFT;
        if(nblocks) {
                if(off >= nblocks)
                        return;
                base = (((long)u.u_count)+511)/512;
                nblocks -= off;                 /* blocks left this filesys */
                if(base > nblocks) {
                        if(rw == B_WRITE) {
                                u.u_error = ENXIO;
                                return;
                        }
                        base = (base-nblocks)*512;
                        u.u_count -= base;
                        u.u_arg[1] -= base;
                }
        }

        base = u.u_base;
        /*
         * Check odd base, odd count, and address wraparound
         */
        if (base&01 || u.u_count&01 || base>=base+u.u_count)
                goto bad;
        ts = (u.u_tsize+127) & ~0177;
        if (u.u_sep)
                ts = 0;
        nb = (base>>6) & 01777;
        /*
         * Check overlap with text. (ts and nb now
         * in 64-byte clicks)
         */
        if (nb < ts)
                goto bad;
        /*
         * Check that transfer is either entirely in the
         * data or in the stack: that is, either
         * the end is in the data or the start is in the stack
         * (remember wraparound was already checked).
         */
        if ((((base+u.u_count)>>6)&01777) >= ts+u.u_dsize
            && nb < 1024-u.u_ssize)
```

```
#ifdef MAKEMAUS
        /* Check if within MAUS section.
         * Since a process' MAUS references may not physically
         * abutt but may be disjoint and often overlap, it is a waste of time
         * and space to calculate the most generally legal move. Therefore,
         * it is only legal to raw I/O from one maus section at a
         * time with no spill over.
         */
                if(((((ts=u.u_uisd[(nb)>>7)+8])&RW)==RO) && rw==B_READ)
                        || ( ((ts)>>8) &0177) + 1
                                < (((base&017777)+u.u_count+63)>>6)&01777))
#endif
                        goto bad;

        bp = getbfh();
        bp->b_flags =| B_BUSY | B_PHYS | rw;
        bp->b_dev = dev;
        /*
         * Compute physical address by simulating
         * the segmentation hardware.
         */
        ts = (u.u_sep ? UDSA : UISA)->r[nb>>7] + (nb&0177);
        bp->b_paddr = ((paddr_t)ts << 6) + (base & 077);
        bp->b_blkno = off;
        bp->b_bcount = u.u_count;
        bp->b_error = 0;
        u.u_procp->p_flag =| SLOCK;
        VTPROCENT(u.u_procp,PR_FLAG);
        (*strat)(bp);
        if(rw&B_READ) {
                meas.m_dread++;
                u.u_dread++;
        } else {
                meas.m_dwrit++;
                u.u_dwrit++;
        }
        spl6();
        while ((bp->b_flags&B_DONE) == 0)
                sleep(bp, PRIBIO);
        u.u_procp->p_flag =& ~SLOCK;
        VTPROCENT(u.u_procp,PR_FLAG);
        if(runin) {
                runin = 0;
                wakeup(&runin);
        }
        spl0();
        u.u_count = bp->b_resid;
        geterror(bp);
        hrelse(bp);
        return;
    bad:
        u.u_error = EFAULT;
}

/*
 * Pick up the device's error number and pass it to the user;
```

```
 * if there is an error but the number is 0 set a generalized
 * code.
 */
geterror(bp)
register struct buf *bp;
{

        if (bp->b_flags&B_ERROR)
                if ((u.u_error = bp->b_error)==0)
                        u.u_error = EIO;
}

/*
 * The following routines allocate and free buf headers.
 * Buf headers are used by physical I/O routines whenever
 * I/O must be done to some random core area.
 * The following routine is used to allocate headers:
 *              getbfh
 * The following routine is used to free headers:
 *              hrelse
 */

/*
 * Initialize free header list (called from main).
 */
hinit()
{
        register struct buf *bp;

        hfreelist = bp = heads;
        for(; bp<&heads[NHEAD-1]; bp++)
                bp->av_forw = bp+1;
        bp->av_forw = 0;
}

/*
 * Get a header.
 */
getbfh()
{
        register int sps;
        register struct buf *bp;

        meas.m_physio++;
        sps = spl6();
        while((bp=hfreelist) == 0) {
                h_flags |= B_WANTED;
                meas.m_nphys++;
                sleep((caddr_t)&hfreelist,PRIBIO+1);
        }
        hfreelist = hfreelist->av_forw;
        splx(sps);
        bp->b_flags |= B_HEAD;
        return(bp);
}
```

```
/*
 * Free a header with the side effect of clearing all flags.
 */
hrelse(bp)
register struct buf *bp;
{
        register int sps;

        sps = spl6();
        bp->av_forw = hfreelist;
        hfreelist   = bp;
        bp->b_flags = 0;
        if(h_flags&B_WANTED) {
                h_flags &= ~B_WANTED;
                wakeup((caddr_t)&hfreelist);
        }
        splx(sps);
}
```

```
/*      @(#)devstart.c  2.1     */

#include "sys/param.h"
#include "sys/buf.h"

/*
 * Device start routine for disks
 * and other devices that have the register
 * layout of the older DEC controllers (RF, RK, RP, TM)
 */
#define IENABLE 0100
#define WCOM    02
#define RCOM    04
#define GO      01
devstart(bp, devloc, devblk, hbcom)
register struct buf *bp;
caddr_t devloc;
daddr_t devblk;
{
        register int *dp;
        register int com;

#ifdef PWR_FAIL
        extern pwr_fail;
#endif

        dp = devloc;
        *dp = devblk;                   /* block address */
        *--dp = bp->b_paddr.loword;     /* buffer address */
        *--dp = -(bp->b_bcount>>1);     /* word count */
        com = (hbcom<<8) | IENABLE | GO |
                ((bp->b_paddr.hiword & 03) << 4);
        if (bp->b_flags&B_READ) /* command + x-mem */
                com |= RCOM;
        else
                com |= WCOM;
#ifdef PWR_FAIL
        *--dp = com ^ pwr_fail;
#endif
#ifndef PWR_FAIL
        *--dp = com;
#endif
}
```

```
/*      @(#)dh.c        2.10    */

#
/*.n'dh11'Includes and Defines'*/
/*
 *      Copyright 1973 Bell Telephone Laboratories Inc
 */

/*
 *      DH-11 driver
 *      This driver calls on the DHDM driver.
 */

#include "sys/param.h"
#include "sys/conf.h"
#include "sys/confx.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/ioctl.h"
#include "sys/tty.h"
#include "sys/ttyx.h"
#include "sys/proc.h"
#include "sys/procx.h"
#include "sys/dm11.h"

/*
 * If SILO64 is defined then dh xmit interrupts will occur only when
 * the silo is full. If SILO64 is not defined then xmit interrupts
 * will occur whenever more than DHSILOLV characters are in the silo.
 */

#define SILO64


/*
 * Local Variables:
 *      1. NDH11        - # of DH11s * 16.
 *      2. DHSCNRAT     - # of clock ticks between silo scans.
 *      3. DHSILOLV     - silo alarm level (0,1,2,4,8,16,32).
 */

#ifndef NDH11
#define NDH11    16
#endif
#ifndef DHSCNRAT
#define DHSCNRAT 2
#endif
#ifndef DHSILOLV
#define DHSILOLV 32
#endif

/*
 * Hardware addresses / control bit definitions.
 */
```

```
#ifndef DHADDR
#define DHADDR  0160020         /* Normal starting address for first DH11 */
#endif

#define B7BITS  02
#define B8BITS  03
#define STOP2   04
#define PENABLE 020
#define OPAR    040             /* DEC manuals incorrectly say this bit
                                   causes generation of even parity */
#define HDUPLX  040000
#ifdef  SILO64
#define DHENABL 030000          /* enable interrupts on xmit, store */
#endif
#ifndef SILO64
#define DHENABL 030100          /* enable int on xmit, store, and rcv */
#endif
#define SSPEED  7               /* standard speed: 300 baud */

/*.s'dh11'Structures and Global declarations'*/
/*
 * Used to communicate the number of lines to the DM
 */

int ndh11 NDH11;

/*
 * TTY structure reservations for DH11 lines.
 */

struct  tty dh11[NDH11];

/*
 * flag word indicating which lines have been opened:
 * bit 0: any line on dh0.
 * bit 1: any line on dh1.
 * etc.
 */

int     dhopenf;

/*
 * Software copy of last dhbar
 */

int     dhsar[(NDH11+15)>>4];

/*
 * Data space for DMA output.
 */

struct tty_dma dh_dma[NDH11];

/*
 * Layout of DH11 hardware registers.
 */
```

```
struct dhregs {
        int dhcsr;              /* system control */
        int dhnxch;             /* next received character */
        int dhlpr;              /* line parameter */
        int dhcar;              /* current address */
        int dhbcr;              /* byte count */
        int dhbar;              /* buffer active */
        int dhbreak;            /* break control */
        int dhsilo;             /* silo status */
};
/*.s'dhopen'Open a DH11 line'*/
dhopen(dev, flag) {

        register struct tty *tp;
        register int    line;

#ifdef PWR_FAIL
        extern unsigned pwr_fail;

        if (dev == NODEV) {
                if (pwr_fail)
                        return;
                pwr_init(dh11, NDH11, 0);
                for (line=0; line < (NDH11+15)>>4; line++)
                        if (dhopenf & (1<<line))
                                dhxint(line);
                return;
        }
#endif

        if((line=dev.d_minor) >= NDH11) {
                u.u_error = ENXIO;
                return;
        }
        tp = &dh11[line];

        dhcntrl(dev, 1);
        if ((tp->t_state&ISOPEN) == 0) {
                if((tp->t_state&EVEROPEN) == 0) {
                        dhopenf =| (1 << (dev.d_minor>>4));
                        tp->t_speeds = SSPEED | (SSPEED<<8);
                        tp->t_flags  = ANYP|CRMOD|XTABS|ECHO|NCDELAY|NLDELAY;
                        dh_dma[line].dma_blk = &dh_dma[line].dma_char;
                }
                dhparam(dev);
        }
        (*linesw[tp->t_ltype].l_open)(tp);
}


/*.t'dhclose'Close DH11 line'*/
dhclose(dev) {

        register struct tty *tp;
```

```
        tp = &dh11[dev.d_minor];
        (*linesw[tp->t_ltype].l_close)(dev, tp);
        if (!(tp->t_flags&NOHUP))
                (*cdevsw[major(dev)].d_mctl)(tp,'c',(RQSEND|CDLEAD|LENABLE));
}
/*.s'dhcntrl'Control DH11.'*/

dhcntrl(dev, action) {

        register struct tty *tp;
        register int lsr;
        register struct dhregs *dhadr;
        extern dhstart(),dhscan();
        static scan;

        tp          = &dh11[dev.d_minor];
        tp->t_dev   = dev;
        tp->t_state =| SSTART;
        tp->t_addr  = dhstart;

        /*
         * Set up DH11 control registers for interrupts and SILO.
         */

        dhadr = DHADDR + (dev.d_minor>>4) * sizeof(*dhadr);
        dhadr->dhcsr  =| DHENABL;
        dhadr->dhsilo =  DHSILOLV;

        /*
         * Do the proper type of DM11 action depending on
         * the type of line being initialized and the action parameter.
         */

        lsr = CDLEAD|LENABLE;
        if((tp->t_flags&HDPLX) == 0)
                lsr =| RQSEND;
        if(action)
                (*cdevsw[major(dev)].d_mctl)(tp,'s',lsr);
        else
                (*cdevsw[major(dev)].d_mctl)(tp,'c',lsr);

        /*
         * Test state of DM11 leads and turn on appropriate bits.
         */

        spl5();
        lsr = (*cdevsw[major(dev)].d_mctl)(tp,'t',0377);
        if(lsr&(CARRIER|SUPRD))
                tp->t_state =| CARR_ON;
        else
                tp->t_state =& ~CARR_ON;
        spl0();

        if(scan == 0) {
                timeout(&dhscan,0,DHSCNRAT);
                scan++;
```

```
        }
}
/*.s'dhread'Read a DH11 line'*/
dhread(dev) {

        register struct tty *tp;

        tp = &dh11[dev.d_minor];
        (*lineswitp->t_ltype].l_read) (tp);

}

/*.t'dhrint'Handle DH11 receiver interrupt'*/
dhrint() {

        register struct dhregs *dhadr;
        register int    dev,c;
        struct   tty    *tp;

        dhadr = DHADDR;
        for(dev = 0; dev < (NDH11+15)>>4; dev++) {
                if(dhopenf & (1<<dev))
                        while((c=dhadr->dhnxch) < 0) {  /* char. present */
                                tp = &dh11[((c>>8)&017) + (dev*16)];
                                if(tp >= &dh11[NDH11])
                                        continue;
                                if(((c&0177)==XON)||((c&0177)==XOFF)) {
                                        if (tp->t_flags&TANDEMI) {
                                                dhxoff(tp, c&0177);
                                                continue;
                                        }
                                }
                                (*lineswitp->t_ltype].l_rcvd) (c, tp);
                        }
                dhadr++;
        }
}

/*.t'dhscan'Activate DH11 read routine at intervals'*/
dhscan() {

        dhrint();
        timeout(&dhscan, 0, DHSCNRAT);

}

/*.s'dhxoff'Suspend DMA on XOFF'*/

dhxoff(tp, c)
register struct tty *tp;
{
        register struct dhregs *dhadr;
        register linebit;

        dhadr = DHADDR + (tp->t_dev.d_minor>>4) * sizeof(*dhadr);
        linebit = tp->t_dev & 017;
```

```
        dhadr->dhcsr.lobyte = linebit | DHENABL;
        linebit = 1<<linebit;
        if(c==XON) {
                tp->t_state &= ~XMTXOFF;
                if(dhadr->dhbcr < 0) {
                        dhadr->dhbar |= linebit;
                        dhsar[tp->t_dev.d_minor>>4] |= linebit;
                } else {
                        dhadr->dhbcr = 0;
                        tp->t_state &= ~BUSY;
                        dhstart(tp);
                }
        } else {
                dhadr->dhbar &= ~linebit;
                dhsar[tp->t_dev.d_minor>>4] &= ~linebit;
                tp->t_state |= (BUSY|XMTXOFF);
        }
}

/*.s'dhwrite'Write a DH11 line'*/
dhwrite(dev) {

        register struct tty *tp;

        tp = &dh11[dev.d_minor];
        (*linesw[tp->t_ltype].l_write) (tp);

}

/*.t'dhxint'Handle DH11 transmitter interrupts'*/
dhxint(dev) {

        register struct tty *tp;
        register ttybit, bar;
        struct dhregs *dhadr;

        dhadr = DHADDR + dev.d_minor * sizeof(*dhadr);
        bar = dhadr->dhbar;
        dhadr->dhcsr =& ~0101060;
        bar =^ bar|dhsar[dev.d_minor];
        ttybit = 1;
        tp = &dh11[dev.d_minor*16];
        while(bar) {
                if(bar&ttybit) {
                        bar =& ~ttybit;
                        dhsar[dev.d_minor] =& ~ttybit;
                        tp->t_state =& ~BUSY;
                        dhstart(tp);
                }
                ttybit =<< 1;
                tp++;
        }
}
/*.s'dhstart'Start output on DH11 line'*/
dhstart(atp)  struct tty *atp;  {
```

```
        register struct tty_dma *dp;
        register struct tty     *tp;
        register struct dhregs  *dhadr;
        int cnt, lineno;
        extern dhrstrt();

        tp = atp;
        if (tp->t_state&BUSY)
                return;

        dp = &dh_dma[tp->t_dev.d_minor];
        lineno = tp->t_dev.d_minor&017;
        dhadr = DHADDR + (tp->t_dev.d_minor>>4) * sizeof(*dhadr);

        if((cnt = (*linesw[tp->t_ltype].l_xdma)(tp,dp)) < 0) {
                dhadr->dhcsr.lobyte = lineno | DHENABL | ((dp->dma_xmem&3)<<4);
                dhadr->dhcar = dp->dma_blk;
                dhadr->dhbcr = -(cnt&0177);
                lineno = 1<<lineno;
                dhadr->dhbar =| lineno;
                dhsar[tp->t_dev.d_minor>>4] =| lineno;
                tp->t_state =| BUSY;
        } else if(cnt & CBREAK) {
              dhadr->dhbreak =| 1 << lineno;
              timeout(dhrstrt,tp,(cnt&0177)+chrdelay[tp->t_speeds.hibyte&017]);
              tp->t_state =| TIMEOUT;
        }
}


/*,t'dhrstrt'Restart DH11 after a time delay'*/
dhrstrt(atp)  struct tty *atp;  {

        register lineno;
        register struct dhregs *dhadr;
        register struct tty    *tp;

        tp = atp;
        lineno = tp - dh11;
        dhadr = DHADDR + (lineno>>4) * sizeof(*dhadr);
        dhadr->dhbreak =& ~(1 << (lineno&017));
        ttrstrt(tp);
}
/*.s'dhioctl'Set new DH11 parameters'*/
dhioctl(dev, cmd, addr, flag)
caddr_t addr;
{
        register flg;
        register struct tty *tp;

        tp = &dh11[dev.d_minor];
        flg = 0;
        if (cmd == OLDSGTTY)
                flg = lsgtty(addr, tp);
        else if(ttioccomm(cmd, tp, addr, dev)) {
                if (cmd==TIOCSETP || cmd==TIOCSETN || cmd==TIOCSETO)
```

```
                        flg++;
        } else
                u.u_error = ENOTTY;
        if (flg)
                dhparam(dev);
}
/*,s'dhparam'Set DH11 line parameter register'*/
dhparam(dev) {

        struct {
                int     low12:12;
                int     top4:4;
        } ;
        struct {
                int     low12:12;
                int     sdbits:3;
        } ;
        register struct tty *tp;
        register int lpr;
        register struct dhregs *dhadr;

        tp = &dh11[dev.d_minor];
        dhadr = DHADDR + (dev.d_minor>>4) * sizeof(*dhadr);
        spl5();
        dhadr->dhcsr.lobyte = (dev.d_minor%16) | DHENABL;
        lpr = ((tp->t_speeds.hibyte&017)<<10) | (tp->t_speeds.lobyte<<6);
        if(tp->t_speeds.lobyte != 2)
                if((tp->t_flags&(EVENP|ODDP))==EVENP) {
                        lpr =| B7BITS|PENABLE;
                } else {
                        if((tp->t_flags&(EVENP|ODDP))==ODDP) {
                                lpr =| B7BITS|OPAR|PENABLE;
                        } else {
                                lpr =| B8BITS;
                        }
                }

        if(tp->t_speeds.lobyte <= 3)            /* 110 baud */
                lpr =| STOP2;

        if(tp->t_flags & HDPLX)                 /* half duplex */
                lpr =| HDUPLX;

        if(tp->t_speeds < 0) {  /* non default stop or data bits */
                lpr =& ~(B8BITS | STOP2);
                lpr =| tp->t_speeds.sdbits;
        }
        tp->t_speeds.top4 = (lpr & (B8BITS | STOP2));

        dhadr->dhlpr = lpr;
        spl0();
        /*
         * Kludge to set high speed bit for 212 data sets.
         * Requires special cable with pin 11 connected to pin 23
         * at the data set end.
         */
```

```
        if (tp->t_speeds.lobyte == 9 && tp->t_ltype == 0)
                (*cdevsw[major(dev)].d_mctl)(tp, 's', SUPTD);
        else
                (*cdevsw[major(dev)].d_mctl)(tp, 'c', SUPTD);
}
```

```
/*      @(#)dhdm.c      2.4     */

#
/*.n'dm11'Includes, Defines, Structs, Globals'*/
/*
 *      Copyright 1973 Bell Telephone Laboratories Inc
 */

/*
 *      DM-BB driver
 */

#include "sys/param.h"
#include "sys/tty.h"
#include "sys/ttyx.h"
#include "sys/conf.h"
#include "sys/confx.h"
#include "sys/dm11.h"

#ifndef DMADDR
#define DMADDR  0170500
#endif

struct  tty dh11[];
int     ndh11;

struct dmregs {
        int     dmcsr;
        int     dmlstat;
        int     dmfill1;
        int     dmfill2;
};
/*.s'dmcntrl'Control DM11'*/
dmcntrl(tp, flag, bits)
struct tty *tp;
{
        register struct dmregs *dmadr;
        register int    minord,savcsr;
        int      sps;

        minord = minor(tp->t_dev);
        dmadr = DMADDR + (minord/16) * sizeof(*dmadr);

        sps = spl5();

        /*
         * Turn off the DM11 scanner before attempting to change
         * any of the csr values.  You 'must' wait for the BUSY bit
         * to go off before changing the csr.
         */

        while((savcsr=dmadr->dmcsr)&DMBUSY)
                dmadr->dmcsr =& ~SCENABL;

        /*
```

```
         * Now set the csr to the line to be changed.
         */

        dmadr->dmcsr = minord%16;
        switch(flag) {

                case 's':
                        minord = (dmadr->dmlstat =| bits);
                        break;

                case 'c':
                        minord = (dmadr->dmlstat =& ~bits);
                        break;

                case 't':
                        minord = dmadr->dmlstat&bits;
                        break;

        }

        dmadr->dmcsr = savcsr|(IENABLE|SCENABL);
        splx(sps);
        return(minord);

}                                       /* dmcntrl */
/*.s'dmint'Handle DM11 interrupts'*/
dmint(dev) {

        register struct tty *tp;
        register struct dmregs *dmadr;

        dmadr = DMADDR + dev.d_minor * sizeof(*dmadr);
        if (dmadr->dmcsr&DONE) {

                tp = &dh11[(dmadr->dmcsr&017) + dev.d_minor*16];
                dmadr->dmcsr =& ~(DONE|SCENABL);
                if (tp < &dh11[ndh11])
                        (*linesw[tp->t_ltype].l_dst)(tp, dmadr->dmcsr,
                                                     dmadr->dmlstat);

                dmadr->dmcsr =| SCENABL;
        }
}                                       /* dmint */
```

```
/*      @(#)dhfdm.c     2.4     */

/*
 *      DM-BB fake driver
 */
#include "sys/param.h"
#include "sys/tty.h"

fmcntrl(tp, action)
register struct tty *tp;
{

        switch(action) {
                case 0:
                        tp->t_state =& ~CARR_ON;
                        break;

                case 1:
                        tp->t_state =| CARR_ON;
                        break;
        }
        return(CARRIER);
}
```

```
/*      @(#)dj.c        2.5     */

#
/*
 *   DJ-11 driver
 */

#include "sys/param.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/ioctl.h"
#include "sys/tty.h"
#include "sys/ttyx.h"
#include "sys/conf.h"
#include "sys/confx.h"
#include "sys/proc.h"
#include "sys/procx.h"

#define DJADDR 0160010

#define NDJ11 32
int ndj11 NDJ11;

struct  tty dj11[NDJ11];

int     djopenf;

#define DJENABL 050401
#define BRKSEL 02000            /* break register select */
#define DJRDONE 0200            /* receiver done */

struct djregs {
        int djcsr;
        int djrbuf;
        int djtcr;
        int djtbuf;
};


struct {
        char    lbyte;
        char    hbyte;
};

djcntrl(dev, action)
{
        register struct tty *tp;
        extern djstart();
        static scan;
        extern djscan();

        tp = &dj11[dev.d_minor];
        tp->t_dev = dev;
        tp->t_state =| CARR_ON | SSTART;
        tp->t_addr = djstart;
```

```
        (DJADDR + (dev.d_minor/16)*8)->djcsr =| DJENABL;
        if (scan == 0) {
                timeout(djscan, 0, 4);
                scan++;
        }
}

djopen(dev, flag)
{
        register struct tty *tp;
        register line;
        register struct djregs *djaddr;

#ifdef PWR_FAIL
        extern unsigned pwr_fail;

        if (dev == NODEV) {
                if (pwr_fail)
                        return;
                pwr_init(dj11, NDJ11, 0);
                djaddr = DJADDR;
                for (line=0; line < (NDJ11+15)/16; line++) {
                        if (djopenf & (1<<line))
                                djaddr->djtcr = -1;
                        djaddr++;
                }
                return;
        }
#endif

        if (dev.d_minor >= NDJ11) {
                u.u_error = ENXIO;
                return;
        }
        djcntrl(dev, 1);
        tp = &dj11[dev.d_minor];
        if ((tp->t_state&ISOPEN) == 0) {
                if((tp->t_state&EVEROPEN) == 0) {
                        djopenf =| (1 << (dev.d_minor/16));
                        tp->t_flags = ODDP|EVENP|CRMOD;
                }
        }
        (*linesw[tp->t_ltype].l_open)(tp);
}

djclose(dev)
{
        register struct tty *tp;

        tp = &dj11[dev.d_minor];
        (*linesw[tp->t_ltype].l_close)(dev, tp);
}

djread(dev)
{
        register struct tty *tp;
```

```
        tp = &djll[dev.d_minor];
        (*linesw[tp->t_ltype].l_read)(tp);
}

djwrite(dev)
{
        register struct tty *tp;

        tp = &djll[dev.d_minor];
        (*linesw[tp->t_ltype].l_write)(tp);
}

djscan()
{
        djrint();
        timeout(djscan, 0, 4);
}


djrint()
{
        register struct djregs *djadr;
        register int    dev,c;
        struct   tty    *tp;

        djadr = DJADDR;
        for(dev = 0; dev < (NDJ11+15)/16; dev++) {
                if(djopenf&(1<<dev))
                        while ((c = djadr->djrbuf) < 0) { /* char. present */
                                tp = &djll[((c>>8)&017) + (dev*16)];
                                if (tp >= &djll[NDJ11])
                                        continue;
                                (*linesw[tp->t_ltype].l_rcvd)(c, tp);
                        }
                djadr++;
        }
}

djioctl(dev, cmd, addr, flag)
caddr_t addr;
{
        struct {
                int     low12:12;
                int     dbits:2;
                int     sbit:1;
        } ;
        register struct tty *tp;

        tp = &djll[dev.d_minor];
        if (cmd == OLDSGTTY)
                lsgtty(addr, tp);
        else if (ttioccomm(cmd, tp, addr, dev) == 0) {
                u.u_error = ENOTTY;
                return;
        }
```

```
        if(tp->t_speeds.lobyte <= 3)
                tp->t_speeds.sbit = 1;
        if(tp->t_speeds.lobyte == 2)
                tp->t_speeds.dbits = 0;
        else
                tp->t_speeds.dbits = 3;
}

djxint(dev)
{
        extern ttrstrt();
        struct tty *tpa;
        register struct tty *tp;
        register c, djadr;
                                        /* disable interupt */
        (djadr = DJADDR + dev.d_minor*8)->djcsr =& ~040000;
        tpa = &dj11[dev.d_minor*16];
        while (djadr->djcsr < 0 ) {             /* xmtr done */
                tp = &tpa[(djadr->djtbuf)>>8)&017];
                c = (*linesw[tp->t_ltype].l_xmtd)(tp, 0);
                if(c < 0) {
                        djadr->djtbuf.lbyte = c;
                        continue;
                }
                if(c&CTOUT)
                        goto tmeout;
                if(c&CBREAK) {
                        djadr->djcsr =| BRKSEL;
                        djadr->djtcr =| 1 << (tp-tpa);
                        djadr->djcsr =& ~BRKSEL;
                        c--;
                    tmeout:
                        timeout(ttrstrt, tp, (c&0177)+1);
                        tp->t_state =| TIMEOUT;
                }
                if(c == 0)
                        (*linesw[tp->t_ltype].l_xmtd)(tp, 1);

                djadr->djtcr =& ~(1 << (tp-tpa));
        }
        djadr->djcsr =| 040000;         /* allow interrupt */
}

djstart(atp)
struct tty *atp;
{
        register struct tty *tp;
        register djno;
        register djadr;

        tp = atp;
        if ((tp->t_state&TIMEOUT) == 0) {
                djno = tp - dj11;
                djadr = DJADDR + djno/16*8;
                djno = 1 << djno%16;
                djadr->djcsr =| BRKSEL;
```

```
                djadr->djtcr =& ~djno;
                djadr->djcsr =& ~BRKSEL;
                djadr->djtcr =| djno;
        }
}
```

```
/*      @(#)dmc11.c     2.6.1.1 */

#
/*,n'dmc11'Defines, Structures, and Globals'*/
#include "sys/param.h"
#include "sys/buf.h"
#include "sys/bufx.h"
#include "sys/conf.h"
#include "sys/reg.h"
#include "sys/syserr.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/vtmn.h"

/*
 * DMC11 Register Layout -- by byte and word.
 */

struct dmcregb {
        char    bsel0;
        char    bsel1;
        char    bsel2;
        char    bsel3;
        char    bsel4;
        char    bsel5;
        char    bsel6;
        char    bsel7;
};
struct dmcregw {
        int     sel0;
        int     sel2;
        int     sel4;
        int     sel6;
};

/*
 * DMC11 Parameters.
 */

#ifndef DMCADDR
#define DMCADDR 0160210
#endif
#ifndef NDMC11
#define NDMC11  1
#endif
/* the define symbol NDMCMB is a bit map of which of the
 * DMC11 are of the one megabaud variety.  This map is
 * used to insure that only one such device is open at
 * a time in accordance with DECs support policy.
 * NDMCMB should not be defined in there are less than 2
 * megabaud DMCs on the system.  An example use of this variable
 * would be:
 * suppose 4 DMCs were provided with the first , second and fourth
 * operating at one megabaud, then NDMCMB would be defined as 013
 */
```

```
#define DMCPSW  1               /* satellite DMC password */
#define DELAY   120             /* down-stream boot delay time */

#define DMCXFER 512             /* transfer size limit (bytes) */
#define DMCRDMX 7               /* max # of reads queued by DMC11 */
#define DMCXMMX 7               /* max # of xmts  queued by DMC11 */

#define DMCSPRI -20             /* DMC11 sleep priority */
#define DMCWPRI 5               /* DMC11 wait  priority */

#define TRUE    1
#define FALSE   0
#define RESET   1

#define DMCRESTART      1       /* stty reset command */
#define DMCBOOT         2       /* stty down-stream boot */
#define DMCEOF          3       /* eof */
#define DMCGTTY         4       /* gtty */
#define DMCCGTTY        5       /* gtty with clear */
/*.e*/
/*
 * Defines for bsel0 and bsel2.
 */

#define TYPEI   3               /* request type mask */
#define BACCI   0
#define CNTLI   1
#define BASEI   3
#define INIO    4               /* direction mask */
#define INX     0               /* transmit block */
#define INR     4               /* read     block */
#define RQI     040             /* port request bit */
#define IEI     0100            /* enable input interrupts */
#define RDYI    0200            /* port ready */

#define TYPEO   3               /* request type mask */
#define BACCO   0
#define CNTLO   1
#define OUTIO   4               /* direction mask */
#define OUX     0               /* transmit block */
#define OUR     4               /* read     block */
#define IEO     0100            /* enable output interrupts */
#define RDYO    0200            /* port available */

#define MCLR    0100            /* DMC11 Master Clear */

/*
 * Defines for CNTLI mode (set into sel6).
 */

#define HDPLX   02000           /* half duplex DDCMP operation */
#define SEC     04000           /* half duplex secondary station */
#define MAINT   0400            /* enter maintenance mode */

/*
 * Defines for BACCI/O and BASEI mode (set into sel6).
```

```
 */

#define DMXMEM  0140000         /* xmem bit position */
#define CCOUNT  037777          /* character count mask */
#define RESUME  02000           /* resume (BASEI only) */

/*
 * Defines for CNTLO (set in sel6).
 */

#define DATACK  1               /* re-xmit threshold exceeded */
#define TIMEOUT 2               /* link time out */
#define ORUN    4               /* data recvd, no buffer */
#define MAINTR  010             /* maintenance message received */
#define LSTDATA 020             /* FATAL: data lost, buffer too short */
#define DISC    0100            /* disconnect */
#define STRTRC  0200            /* FATAL: start message received */
#define NXMEM   0400            /* FATAL: very bad something */
#define PRCERR  01000           /* FATAL: pdp11 procedure error */
/*.e*/
/*
 * DMC11 control structure (see devtab in buf.h).
 */

struct dmccntrl {
        char    dm_flags;               /* DMC11 flags */
        char    dm_fill0;
        struct  buf *b_forw;            /* buffers held by DMC11 */
        struct  buf *b_back;            /* buffers held by DMC11 */
        struct  buf *dm_actq;           /* active buffers */
        struct  buf *dm_watq;           /* waiting buffers */
        struct  buf *dm_cmd;            /* delayed command (at most one) */
        char    dm_rcvn;                /* # of reads held by DMC11 */
        char    dm_xmtn;                /* # of xmts  held by DMC11 */
};

#define DMCBUSY 1
#define DMCOPEN 2
#define DMCMANT 4
#define DMCRDYI 010
#define DMCBASI 020
#define DMCCNTI 040
#define DMCINIT 060
#define NEITHER 0

/*
 * DMC11 error log structure.
 */

struct dmcerrs {
        int     dmc_tmsg;               /* # of messages transmitted */
        int     dmc_rmsg;               /* # of messages recevied */
        int     dmc_rtry;               /* # of retry thresholds */
        int     dmc_timo;               /* # of DMC11 timeouts */
        int     dmc_orun;               /* # of data overruns */
        int     dmc_lost;               /* # of times data was lost */
```

```
        int     dmc_disc;               /* # of disconnects */
        int     dmc_strt;               /* # of DMC11 restarts */
        int     dmc_xmem;               /* # of NXMEM errors */
        int     dmc_prce;               /* # of procedure errors */
        int     dmc_mant;               /* # of times in maint. mode */
};

/*
 * per DMC11 global data.
 */

struct  dmccntrl dmc11[NDMC11];         /* Q headers and flags */
struct  dmcerrs  dmcer[NDMC11];         /* error information */
int     dmcarea[NDMC11][128];           /* Base table */
char    dmcmopd[NDMC11][10];            /* Maintenance receive area */

char    dmcmop[] {                      /* Maintenance message */
        6,DMCPSW,DMCPSW,DMCPSW,DMCPSW
};
char    dmcmopr[] {                     /* Maintenance message response */
        8,12,1
};
#ifdef NDMCMB
int opendmc;
#endif
/*DEBUG*/ int dmccnt,dmcpos;
/*.s'dmcopen'Open DMC11'*/
dmcopen(dev, flag)
{
        register struct dmccntrl *dmp;
        register x;

#ifdef PWR_FAIL
        if (dev == NODEV)
                return;
#endif

        dmp = &dmc11[dev.d_minor];
#ifdef NDMCMB
        x = 1 << dev.d_minor;
        if ((opendmc & NDMCMB) && (NDMCMB & x) && !(opendmc & x)) {
                u.u_error = ENOALOC;
                return;
        }
#endif
        spl5();
        if((dmp->dm_flags&DMCOPEN) == 0) {
                dmp->dm_flags = DMCOPEN;
                dmcinit(dev);
#ifdef NDMCMB
                opendmc |= x;
#endif
        }
        spl0();
        VTMISCENT(3,"dmcopen",dmccnt++);
        VTMISCENT(4,"flags",dmp->dm_flags);
```

```
}

/*.t'dmcclose'Close DMC11'*/
dmcclose(dev)
{
        register struct dmccntrl *dmp;

        dmp = &dmc11[dev.d_minor];
        spl5();
        dmcclr(dev);
        dmp->dm_flags =& ~DMCOPEN;
#ifdef NDMCMB
        opendmc &= ~(1 << dev.d_minor);
#endif
        spl0();
        VTMISCENT(8,"dmcclose",dmccnt++);
        VTMISCENT(9,"flags",dmp->dm_flags);
}
/*.s'dmcinit'Initialize DMC11'*/
dmcinit(dev)
{
        register int    sps;
        register struct dmccntrl *dmp;
        register struct dmcregb  *dmcadr;

        dmp = &dmc11[dev.d_minor];
        sps = PS->integ;
        spl5();
        switch(dmp->dm_flags&DMCINIT) {

        case NEITHER:
                dmcclr(dev);
                dmp->dm_flags =| DMCBASI;

        case DMCBASI:
                if(dmcload(dmp,BASEI,dmcarea[dev.d_minor],0) == FALSE)
                        break;
                dmp->dm_flags =& ~DMCBASI;
                dmp->dm_flags =| DMCCNTI;

        case DMCCNTI:
                if(dmcload(dmp,CNTLI,0,0) == FALSE)
                        break;
                dmp->dm_flags =& ~DMCCNTI;
                wakeup(dmp);
                dmcadr = DMCADDR + (dev.d_minor*sizeof(*dmcadr));
                dmcadr->bsel2 =| IEO;

        }
        PS->integ = sps;
}
/*.s'dmcclr'Abort DMC11'*/
dmcclr(dev)
{
        int sps;
```

```
        register struct dmcregb *dmp;
        register struct buf     *bp,*abp;

        sps = PS->integ;
        spl5();
        dmp         = DMCADDR + (dev.d_minor*sizeof(*dmp));
        dmp->bsel1 = MCLR;

        dmp = &dmc11[dev.d_minor];
        dmp->dm_flags =& ~(DMCBUSY|DMCMANT|DMCRDYI|DMCINIT);
        dmp->dm_rcvn = dmp->dm_xmtn = dmp->dm_cmd = 0;

        /*
         * Mark all active buffers bad.
         */

        abp = dmp->dm_actq;
        dmp->dm_actq = 0;
        while(bp=abp) {
                bp->b_flags =| B_ERROR;
                abp = bp->dm_actq;
                iodone(bp);
        }

        /*
         * Mark all buffers waiting to be
         * given to the DMC11 bad.
         */

        abp = dmp->dm_watq;
        dmp->dm_watq = 0;
        while(bp=abp) {
                bp->b_flags =| B_ERROR;
                abp = bp->dm_watq;
                iodone(bp);
        }
        PS->integ = sps;
        wakeup(dmp);
}
/*.s'dmcstrategy'Queue buffer'*/
dmcstrategy(bp)
register struct buf *bp;
{
        int sps;
        register struct dmccntrl *dmp;
        register struct buf      *abp;

        if(bp->b_bcount > DMCXFER) {
                bp->b_flags =| B_ERROR;
                iodone(bp);
                return;
        }

        dmp = abp = &dmc11[bp->b_dev.d_minor];
        sps = PS->integ;
        spl5();
```

```
        /*
         * Put this request on the end of the wait Q.
         */

        while(abp->dm_watq)
                abp = abp->dm_watq;
        abp->dm_watq = bp;
        bp->dm_watq  = 0;
        dmcstart(dmp);
        PS->integ = sps;
}
/*.s'dmcstart'Give DMC11 buffers for I/O'*/
dmcstart(dmp)
register struct dmccntrl *dmp;
{
        register struct buf *bp,*obp;
        struct   buf *abp;

        if(dmp->dm_flags&DMCBUSY)
                return;

        /*
         * Take things off the wait Q until the wait Q is
         * empty or until a DMC11 limit is exceeded.
         */

        bp = (obp=dmp)->dm_watq;
        while(bp) {
                if(bp->b_flags&B_READ)
                        if(dmp->dm_rcvn < DMCRDMX) {
                                dmp->dm_rcvn++;
                        } else {
                                bp = (obp=bp)->dm_watq;
                                continue;
                        }
                else
                        if(dmp->dm_xmtn < DMCXMMX) {
                                dmp->dm_xmtn++;
                        } else {
                                bp = (obp=bp)->dm_watq;
                                continue;
                        }

                /*
                 * Remove the request from the wait Q and add it to
                 * the end of the active Q.  Then attempt to give it
                 * to the DMC11.
                 */

                obp->dm_watq = bp->dm_watq;
                abp = obp;  obp = dmp;
                while(obp->dm_actq)
                        obp = obp->dm_actq;
                obp->dm_actq = bp;
                bp ->dm_actq = 0;
```

```
                obp = abp;
                if(dmccmd(BACCI,bp,dmp) == FALSE)
                        return;
                bp = obp->dm_watq;
        }
}
/*.s'dmccmd'Issue command to DMC11'*/
dmccmd(type,bp,dmp)
struct dmccntrl *dmp;
register struct buf *bp;
{
        register int count,cmd;
        int base;

        /*
         * Called when DMC11 not busy - at priority 5.
         * NOTE:  This routine should never be called with a
         *        BASEI request.  It won't even recognize it.
         */


        switch(type) {

        case BACCI:
                cmd = BACCI|((bp->b_flags&B_READ)<<2);
                if((bp->b_flags&B_MAP) == 0) {
                        mapalloc(bp);
                }
                base = bp->b_paddr.loword;
                count = (bp->b_bcount)&CCOUNT;
                count =| (bp->b_paddr.hiword&03)<<14;
                break;

        case CNTLI:
                cmd = CNTLI;
                base = 0;
                count = bp->b_paddr;
                break;

        }
        if(dmcload(dmp,cmd,base,count) == TRUE) {
                if(bp->b_bcount == 0)
                        iodone(bp);
                return(TRUE);
        }
        dmp->dm_cmd = bp;
        return(FALSE);
}
/*.s'dmcinit'Initialize the DMC11'*/
/*.s'dmcload'Load parameters into DMC11'*/
dmcload(dmp,type,w0,w1)
register struct dmccntrl *dmp;
{
        register struct dmcregb *dmcadr;
        register int n;
```

```
        dmcadr = DMCADDR + (dmp-dmc11)*sizeof(*dmcadr);

        /*
         * If DMCRDYI is already on, then it means that this
         * command has already been loaded; the DMC11 didn't
         * set RDYI fast enough so the driver decided to wait
         * for the interrupt (which has now been received).
         */

        if((dmp->dm_flags&DMCRDYI) == 0) {
                dmcadr->bsel0 = (RQI|type);
                n = 5;
                while((dmcadr->bsel0&RDYI) == 0) {
                        if(--n == 0) {
                                dmcadr->bsel0 =| IEI;
                                dmp->dm_flags =| DMCBUSY;
                                VTMISCENT(21+dmcpos,"wait",type);
                                VTMISCENT(22+dmcpos,"flags",dmp->dm_flags);
                                VTMISCENT(23+dmcpos,"sel0",dmcadr->sel0);
                                dmcpos = (dmcpos+3)%39;
                                VTMISCENT(21+dmcpos,"******",dmccnt);
                                return(FALSE);
                        }
                }
        }

        /*
         * DMC11 ready to take data.  Load data and
         * then clear RQI to tell it data is loaded.
         */

        dmp->dm_flags =& ~DMCRDYI;
        dmcadr->sel4  = w0;
        dmcadr->sel6  = w1;
        dmcadr->bsel0 =& ~RQI;
        while(dmcadr->bsel0&RDYI);

        VTMISCENT(21+dmcpos,"LOAD",type);
        VTMISCENT(22+dmcpos,"w0=",w0);
        VTMISCENT(23+dmcpos,"w1=",w1);
        dmcpos = (dmcpos+3)%39;
        VTMISCENT(21+dmcpos,"******",dmccnt);

        return(TRUE);
}
/*,s'dmciint'Deque delayed commands'*/
dmciint(dev)
{
        register struct dmcregb *dmp;
        register int    type;
        register struct buf     *bp;

        /*
         * The only time input interrupts are enabled is when
         * a command is issued and the DMC11 does not set RDYI
         * fast enough to suit the driver.  Thus there always
```

```
         * should be some kind of command queued up when this
         * routine is called.
         */

        dmp         = DMCADDR + (dev.d_minor * sizeof(*dmp));
        dmp->bsel0 =& ~IEI;

        VTMISCENT(21+dmcpos,"DMCI",dmp->bsel0);
        VTMISCENT(22+dmcpos,"w0=",dmp->sel4);
        VTMISCENT(23+dmcpos,"w1=",dmp->sel6);
        dmcpos = (dmcpos+3)%39;
        VTMISCENT(21+dmcpos,"*******",dmccnt);

        dmp = &dmc11[dev.d_minor];
        dmp->dm_flags =& ~DMCBUSY;
        dmp->dm_flags =| DMCRDYI;
        wakeup(dmp);

        if(dmp->dm_flags&DMCINIT) {
                dmcinit(dev);
        } else if(bp=dmp->dm_cmd) {
                if(bp->b_bcount == 0)
                        type = bp->b_error;
                else
                        type = BACCI;

                if(dmccmd(type,bp,dmp) == FALSE)
                        return;
        }
        dmp->dm_cmd = 0;
        dmcstart(dmp);
}
/*,s'dmcoint'Handle DMC11 output interrupts'*/
dmcoint(dev)
{
        register struct dmccntrl *dmp;
        register struct buf *bp,*abp;
        struct dmcregb *dmcadr;
        int count,reinit;

        /*
         * Output interrupt occurred.
         */

        dmp = &dmc11[dev.d_minor];
        dmcadr = DMCADDR + (dev.d_minor*sizeof(*dmcadr));

        VTMISCENT(21+dmcpos,"DMCO",dmcadr->bsel2);
        VTMISCENT(22+dmcpos,"w0=",dmcadr->sel4);
        VTMISCENT(23+dmcpos,"w1=",dmcadr->sel6);
        dmcpos = (dmcpos+3)%39;
        VTMISCENT(21+dmcpos,"*******",dmccnt);

        switch(dmcadr->bsel2&TYPEO) {

        case BACCO:
```

```
                count = dmcadr->sel6&CCOUNT;
                abp   = dmp;

                /*
                 * Look through the active Q for the buffer
                 * the DMC11 says is finished.  Assume the DMC11
                 * returns the buffers in FIFO order.
                 */

                while(bp=abp->dm_actq) {
                        if((dmcadr->bsel2&OUTIO)>>2 == (bp->b_flags&B_READ)) {
                                dmcadr->bsel2 =& ~RDYO;
                                dmcadr = &dmcer[bp->b_dev.d_minor];
                                if(bp->b_flags&B_READ) {
                                        dmp->dm_rcvn--;
                                        dmcadr->dmc_rmsg++;
                                } else {
                                        dmp->dm_xmtn--;
                                        dmcadr->dmc_tmsg++;
                                }
                                bp->b_resid = bp->b_bcount - count;
                                abp->dm_actq = bp->dm_actq;
                                iodone(bp);
                                dmcstart(dmp);
                                return;
                        }
                        abp = bp;
                }
                dmcadr->bsel2 =& ~RDYO;
                printf("DMC11(%d): block not found\n",dev.d_minor);
                return;

        case CNTLO:
                bp    = &dmcer[dev.d_minor];
                count = dmcadr->sel6;
                dmcadr->bsel2 =& ~RDYO;
                reinit  = FALSE;
                if(count&DATACK)
                        bp->dmc_rtry++;
                if(count&TIMEOUT)
                        bp->dmc_timo++;
                if(count&ORUN)
                        bp->dmc_orun++;
                if(count&MAINTR)
                        bp->dmc_mant++;
                if(count&DISC)
                        bp->dmc_disc++;
                if(count&LSTDATA) {
                        reinit = TRUE;
                        bp->dmc_lost++;
                }
                if(count&STRTRC) {
                        reinit = TRUE;
                        bp->dmc_strt++;
                }
                if(count&NXMEM) {
```

```
                        reinit = TRUE;
                        bp->dmc_xmem++;
                }
                if(count&PRCERR) {
                        reinit = TRUE;
                        bp->dmc_prce++;
                }
                if(reinit)
                        dmcinit(dev,dmp);
        }
}
/*.s'dmcioctl'DMC11 Maintenance functions'*/
dmcioctl(dev, cmd, addr, flag)
caddr_t addr;
{
        register int *uaddr,*saddr,n;
        struct buf *bp;
        int dmcinit();

        VTMISCENT(0,"ioctl",dmccnt++);
        VTMISCENT(1,"addr=",addr);
        VTMISCENT(2,"cmd",cmd);

        if (cmd == DMCGTTY || cmd == DMCCGTTY) {
                uaddr = addr;
                n     = sizeof(dmcer[0])>>1;
                saddr = &dmcer[dev.d_minor];
                while(n--) {
                        suword(uaddr++,*saddr);
                        if (cmd == DMCCGTTY)
                                *saddr++ = 0;
                        else
                                saddr++;
                }
                return;
        }

        uaddr = &dmc11[dev.d_minor];
        spl5();
        while(uaddr->dm_flags&DMCBUSY)
                sleep(uaddr,DMCSPRI);

        switch(cmd) {
        case DMCRESTART:
                dmcinit(dev);
                break;

        case DMCBOOT:
                dmcinit(dev);
                while(uaddr->dm_flags&DMCBUSY)
                        sleep(uaddr,DMCSPRI);
                uaddr->dm_flags =| DMCMANT;
                if(dmcboot(dev) == FALSE) {
                        u.u_error = EIO;
                        dmcinit(dev);
                } else {
```

```
                        timeout(dmcinit,dev,DELAY);
                        while(uaddr->dm_flags&DMCMANT)
                                sleep(uaddr, DMCSPRI);
                }
                break;

        case DMCEOF:
                bp = dmcbufq(dev,dmcmopd[dev.d_minor],1,B_WRITE);
                while((bp->b_flags&B_DONE) == 0)
                        sleep(bp,DMCSPRI);
                hrelse(bp);
                break;
        }
        spl0();
}
/*.s'dmcboot'Force downstream boot'*/
dmcboot(dev)
{
        register struct buf *bp,*abp;
        register int    j;

        VTMISCENT(10,"*BOOT*",dmccnt++);

        /*
         * First force DMC11 into maintenance mode.
         */

        bp          = getbfh();
        bp->b_bcount = 0;
        bp->b_paddr   = MAINT;
        bp->b_dev    = dev;
        bp->b_error  = CNTLI;
        dmccmd(CNTLI,bp,&dmc11[dev.d_minor]);
        while((bp->b_flags&B_DONE) == 0)
                sleep(bp, DMCSPRI);
        if(bp->b_flags&B_ERROR) {
                hrelse(bp);
                return(FALSE);
        }
        hrelse(bp);

        /*
         * Now clear buffer where response to maintenance message
         * to be sent will be received.
         */

        for(j=0;j<sizeof(dmcmopd[0]);j++)
                dmcmopd[dev.d_minor][j] = 0;

        /*
         * Set up receive buffer for response and the send maintenance
         * message to satellite to force it into maintenance mode.
         */

        bp  = dmcbufq(dev,dmcmopd[dev.d_minor],sizeof(dmcmopd[0]),B_READ);
        abp = dmcbufq(dev,dmcmop,sizeof(dmcmop),B_WRITE);
```

```
        VTMISCENT(11,"x-boot",dmccnt++);
        while((abp->b_flags&B_DONE) == 0)
                sleep(abp, DMCSPRI);
        VTMISCENT(12,"r-boot",dmccnt++);
        while((bp->b_flags&B_DONE) == 0)
                sleep(bp, DMCSPRI);
        hrelse(bp);
        hrelse(abp);

        /*
         * See if correct response was received.
         */

        VTMISCENT(13,"check",dmccnt++);
        for(j=0;j<sizeof(dmcmopr);j++) {
                if(dmcmopd[dev.d_minor][j] != dmcmopr[j])
                        return(FALSE);
        }

        /*
         * Satellite is now waiting for boot routine.
         */

        u.u_offset = 0;
        u.u_base   = u.u_arg[0];
        u.u_count  = u.u_arg[2];
        VTMISCENT(14,"physio",dmccnt++);
        physio(&dmcstrategy,dev,B_WRITE,0);
        u.u_ar0[R0] = bp->b_resid;
        return(TRUE);
}


/*.t'dmcbufq'Get a buffer for reading/writing and Q'*/
dmcbufq(dev,bufa,nbytes,flag)
{
        register struct dmccntrl *dmp;
        register struct buf *bp;

        bp           = getbfh();
        bp->b_bcount = nbytes;
        bp->b_paddr = (paddr_t)(unsigned)bufa;
        bp->b_dev    = dev;
        bp->b_error  = 0;
        bp->b_flags  =| flag;
        dmp          = &dmc11[dev.d_minor];
        bp ->dm_watq = dmp->dm_watq;
        dmp->dm_watq = bp;
        dmcstart(dmp);
        return(bp);
}


/*.t'dmcread'Raw DMC11 read'*/
dmcread(dev)
{
```

```
        physio(dmcstrategy,dev,B_READ,0);
}


/*.t'dmcwrite'Raw DMC11 write'*/
dmcwrite(dev)
{
        physio(dmcstrategy,dev,B_WRITE,0);
}
```

```
/*      @(#)dn.c        2.4.1.3 */

/*****************************************************************/
/* DN DRIVER                                        s.michael    */
/* this DN DRIVER provides the following functions:              */
/*      1. multiple writes can be done with out doing a close    */
/*      2. a write can be done anytime after a open              */
/*      3. d causes the 801 to drop the current call             */
/*      4. e causes the driver to send a EON to the 801          */
/*      5. - causes the driver to time for 4 seconds             */
/*      6. f causes the driver to FLASH                          */
/*****************************************************************/

#include "sys/param.h"
#include "sys/conf.h"
#include "sys/confx.h"
#include "sys/user.h"
#include "sys/userx.h"

struct {
        struct dnreg {
                int     dn_reg;
        } dnline[][4];
};

/*
 * This structure overlays the dnreg structure
 *      dn_stat is unreferenced
 *      dn_datum used to output the next digit to be dialed
 */

struct {
        char    dn_stat;
        char    dn_datum;
};

#ifndef NDN11
#define NDN11   4
#endif

int     ndn11   =       NDN11;

#ifndef DNADDR
#define DNADDR  0175200
#endif

#define PWI     0100000
#define ACR     040000
#define DLO     010000
#define DONE    0200
#define IENABLE 0100
#define DSS     040
#define PND     020
#define MENABLE 04
#define DPR     02
#define CRQ     01
```

```
#define EON     014
#define WDIAL   015             /* Wait for dial tone */
#define SEC1    60              /* flash time in 1/60 of sec            */
#define SEC2    120             /* delay for return to user after d */
#define DELAY   240             /* delay for second dial time in 1/60 sec */

#define DNPRI   5
#define MAXDNS 16               /*maximum number of dnlls allowed*/

dnopen(dev, flag)
{
        register struct dnreg *dnadr;
        register int physdn;
        register int seldn;

#ifdef PWR_FAIL
        if (dev == NODEV)
                return;
#endif
        seldn = minor(dev);
/*determine which physical dn to address*/
        physdn = (seldn < MAXDNS ? seldn : (seldn >> 4) - 1);
        dnadr = DNADDR->dnline[0] + physdn;
        if(physdn >= ndnll || dnadr->dn_reg & PWI)
        {
                u.u_error = ENXIO;
                return;
        }
        if(seldn < MAXDNS && dnadr->dn_reg & DLO)
        {
                u.u_error = EBUSY;
                return;
        }
        ((caddr_t)dnadr & ~07)->dn_reg |= MENABLE;
        if(seldn < MAXDNS)
                return;
/*the driver now assumes that shared ACU hardware is intalled  */
/*and writes the data set select digit to the ACU so the user  */
/*doesn't have to worry about it.                               */
        spl5();
        while(dnadr->dn_reg & DLO)
                sleep(&(dnadr->dn_datum),DNPRI);
        dnadr->dn_reg |= (CRQ | IENABLE);
        if((dnadr->dn_reg & (PWI | ACR | DSS)) == 0)
        {
                if((dnadr->dn_reg & PND) == 0)
                        sleep(dnadr,DNPRI);
                dnadr->dn_datum = (seldn & 017) + 1;
                dnadr->dn_reg |= DPR;
                sleep(dnadr,DNPRI);
        }
        if(dnadr->dn_reg & (PWI | ACR))
                u.u_error = ENXIO;
        else if(dnadr->dn_reg & DSS)
                u.u_error = EBUSY;
        if(u.u_error)
```

```
                dnadr->dn_reg &= ~(CRQ);
        spl0();
}

dnclose(dev)
{
        register int seldn;
        register int physdn;
        seldn = minor(dev);
/*determine which physical dn to address*/
        physdn = (seldn < MAXDNS ? seldn : (seldn >> 4) - 1);
        DNADDR->dnline[0][physdn].dn_reg &= MENABLE;
}

dnwrite(dev)
{
        extern wakeup();
        register struct dnreg *dnadr;
        register c;
        register int seldn;
        register int physdn;
        seldn = minor(dev);
/*determine which physical dn to address*/
        physdn = (seldn < MAXDNS ? seldn : (seldn >> 4) - 1);
        dnadr = DNADDR->dnline[0] + physdn;

        /* Set priority level to 5 so that, in the case of
         * use of a shared ACU facility , a race condition
         * involving the PND signal is resolved.
        */
        spl5();
        dnadr->dn_reg |= (CRQ | IENABLE);
        while((dnadr->dn_reg & (PWI | ACR | DSS)) == 0)
        {
                if((dnadr->dn_reg&PND) == 0 || u.u_count == 0 || (c=cpass()) <
*  0)
                        sleep(dnadr;DNPRI);
                else
                {
                        switch(c)
                        {
                                case 'd':
                                        dnadr->dn_reg &= ~(CRQ | IENABLE);
                                        timeout(wakeup,dnadr,SEC2);
                                        sleep(dnadr,DNPRI);
                                        /* Set back to level 0 */
                                        spl0();
                                        return;
                                case '-':
                                        timeout(wakeup,dnadr,DELAY);
                                        sleep(dnadr,DNPRI);
                                        break;
                                case 'f':
                                        dnadr->dn_reg &= ~CRQ;
                                        timeout(wakeup,dnadr,SEC1);
                                        sleep(dnadr,DNPRI);
```

```
                                        dnadr->dn_reg |= CRQ;
                                        break;
                                case 'e':
                                        c = EON;
                                        goto def;
                                case 'w':
                                        c = WDIAL;
                                        goto def;
                                case '*':
                                        c = 012;
                                        goto def;
                                case '#':
                                        c = 013;
                                        goto def;
                                default:
                                        if(c<'0' || c>'9')
                                                continue;
                                def:
                                        dnadr->dn_datum = (c & 017);
                                        dnadr->dn_reg |= DPR;
                                        sleep(dnadr,DNPRI);
                        }
                }
        }
        if(dnadr->dn_reg&(PWI | ACR))
                u.u_error = EIO;
        dnadr->dn_reg &= ~CRQ;
        wakeup(&(dnadr->dn_datum));
        /* Return to level 0 */
        spl0();
}

dnint(dev)
{
        register struct dnreg *dnadr;
        register count;

        dnadr = DNADDR->dnline[dev.d_minor];

        dnadr->dn_reg &= ~(MENABLE);
        dnadr->dn_reg |= MENABLE;
        count = 4;
        do {
                if(dnadr->dn_reg & DONE) {
                        dnadr->dn_reg &= ~(DONE);
                        wakeup(dnadr);
                }
        } while(dnadr++, --count);
}
```

```
/*      @(#)ds40.c      2.7     */

/*,n'ds40'Terminal Controls Package'*/
#include "sys/param.h"
#include "sys/crtctl.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/tty.h"
#include "sys/ttyx.h"

#define SDESC   0200            /* output escape flag */
#define NONE    0177            /* no direct conversion flag */

char ds40map[] {
        LCA,    NONE,           /* Load Cursor Adress (simulated) */
        CUP,    SDESC|'7',      /* Move Cursor Up */
        CDN,    SDESC|'B',      /* Move Cursor Down */
        CRI,    SDESC|'C',      /* Move Cursor Right */
        CLE,    010,            /* Move Cursor Left */
        NL,     NONE,           /* Special DS40 newline */
        CRTN,   SDESC|'G',      /* Return Cursor to start of line */
        HOME,   SDESC|'H',      /* Home Cursor */
        CS,     SDESC|'R',      /* Clear Screen */
        EEOP,   SDESC|'J',      /* Erase to end of page */
        EEOL,   NONE,           /* Erase to end of line */
        IL,     SDESC|'L',      /* Insert Line */
        DL,     SDESC|'M',      /* Delete Line */
        IC,     SDESC|'^',      /* Insert Char */
        DC,     SDESC|'P',      /* Delete Char */
        STB,    SDESC|'3',      /* Start Blink */
        SPB,    SDESC|'4',      /* Stop  Blink */
        BPRT,   SDESC|'W',      /* Begin Protect */
        EPRT,   SDESC|'X',      /* End   Protect */
        STAB,   SDESC|'0',      /* Set Single    Tab */
        ATAB,   SDESC|'1',      /* Set Column of Tabs */
        CTAB,   SDESC|'2',      /* Clear Tabs */
        USCRL,  SDESC|'S',      /* Scroll Up */
        DSCRL,  SDESC|'T',      /* Scroll Down */
        ASEG,   SDESC|'U',      /* Advance Segment */
/*      KBL,    NONE,           /* Keyboard Lock */
/*      KBU,    NONE,           /* Keyboard Unlock */
        UVSCN,  NONE,
        DVSCN,  NONE,
        0,
};

extern int rowpres,colpres;
/*.s'ds40output'Translate control into ds40 equivalent'*/
ds40output(ac, atp)
struct tty *atp;
{
        register struct clist *qp;
        register char *cp;
        register int  c;
```

```
        c = ac;  qp = &atp->t_outq;
        for(cp=ds40map;*cp;cp=+2) {
                if(c == *cp) {

                        /*
                         * If mapping given, output hardware control
                         * sequence.
                         */

                        if(*++cp != NONE) {
                                if(*cp&SDESC)
                                        putc(ESC,qp);
                                qputc(*cp&0177,qp);
                        }

                        /*
                         * Add in control timing delays, or simulate
                         * functions not in hardware.
                         */

                        switch(c) {

                        case DL:  case DSCRL:  case DC:
                        case IL:  case USCRL:  case IC:
                                putc(QESC,qp);
                                putc(0213,qp);
                                break;

                        case NL:
                                putc(ESC,qp);
                                putc('G',qp);
                                putc(ESC,qp);
                                putc('B',qp);
                                break;

                        case CS:
                                putc(QESC,qp);
                                putc(0213,qp);

                        case HOME:
                                atp->t_tmflgs =| TERM_BIT;
                                break;

                        case EEOL:
                                for(c=0;c<80;c++)
                                        ds40output(DC,atp);
                                break;

                        case EEOP:
                                putc(QESC,qp);
                                putc(0216,qp);
                                break;

                        case CTAB:
                                putc(QESC,qp);
                                putc(0212,qp);
```

```
                        break;

                case SPB:
                case STB:
                        atp->t_col--;
                        break;

                case LCA:
                        if((atp->t_tmflgs&TERM_BIT) == 0) {
                                ds40output(HOME,atp);
                                rowpres = colpres = 0;
                        }
                        ds40lca(atp->t_row,atp->t_col,atp);
                        break;
                case UVSCN:
                        ttuvscn(atp);
                        break;
                case DVSCN:
                        ttdvscn(atp);
                        break;

                }
        }
    }
}
/*,s'ds40lca'Simulate LCA'*/
ds40lca(row, col, atp)
struct tty *atp;
{
        register int rowmoves,colmoves,aseg;
        int rowdir,coldir,x;

        /*
         * Compute the number of rows and columns to move relative
         * to the current position of the cursor.
         */

        rowmoves = row - rowpres%(atp->t_lrow+1);
        if(rowmoves < 0) {
                rowdir   = CUP;
                rowmoves = -rowmoves;
        } else
                rowdir =  CDN;

        colmoves = col - ((colpres > 79) ? 79: colpres);
        if(colmoves < 0) {
                coldir   = CLE;
                colmoves = -colmoves;
        } else
                coldir   = CRI;
        if(colmoves > (col-1)) {
                colmoves = col;
                coldir   = CRTN;
        }

        /*
```

```
         * Compute the number of moves if the cursor is homed.
         */

        x       = 1 + row/24 + row%24 + col;      /* "home" move sequence */

        if(x < rowmoves+colmoves+1) {
                rowdir   = HOME;
                rowmoves = row%24;
                coldir   = CRI;
                colmoves = col;
                aseg     = row/24;
        } else
                aseg     = 0;

        if(rowdir == HOME) {
                ds40output(HOME,atp);
                rowdir = CDN;
        }
        while(rowmoves--) ds40output(rowdir,atp);
        while(aseg--)     ds40output(ASEG,atp);
        if(coldir == CRTN) {
                ds40output(CRTN,atp);
                coldir = CRI;
        }
        while(colmoves--) ds40output(coldir,atp);
}
/*.s'ds40input'Input character mapping'*/
#define GS      035
#define EOT     004
#define CR      015

ds40input(ac, atp)
struct tty *atp;
{
        register int c;

        c = ac&0177;
        if(c == GS)
                c = EOT;

        if(c == CR)
                ttyoutput('\n',atp);

        return(c);

}


/*.t'ds40ioctl'Dummy routine for ioctl'*/
ds40ioctl(tp, flag, nvrow)
register struct tty *tp;
register unsigned nvrow;
{

        if (nvrow > 71) {
                u.u_error = EINVAL;
```

```
                return;
        }
        if (flag == LSET) {
                tp->t_lrow = 71;
                tp->t_flags = NLDELAY|NCDELAY|NTDELAY|EVENP|ODDP|CRMOD|
                                ECHO|XTABS|TANDEMI;
                tp->t_tmflgs = ANL;
        }
}
```

```
/*      @(#)dz.c        2.12    */

#

/*
 *                      DZ11 DRIVER                     s.michael
 *
 *  the dz11 driver contains the following subroutines.
 *   1.dzopen-opens a dz line and sets line type 0 if line was never opened.
 *   2.dzcntl-enables the dz11 hardware and set line type 0 if line was
 *           never open.
 *   3.dzparam-set dz11 line control parameter register.
 *   4.dzioctl-modify or read dz11 line control parameters.
 *   5.dzmcntrl-sets,clears, or tests dtr register.
 *   6.dzscan-scans silo and carrier bits.
 *   7.dzclose-turns off dtr lead and clears ISOPEN flag.
 *   8.dzread-reads a dz11 line.
 *   9.dzwrite-writes to a dz11 line.
 *  10.dzrint-receiver interrupt routine.
 *  11.dzxint-transmit interrupt routine.
 *  12.dzstart-starts a dz11 line for transmitting.
 *  13.dzrstrt-restarts transmission on a line after a break or timeout.
*/

/* dz11 includes and defines */

#include "sys/param.h"
#include "sys/conf.h"
#include "sys/confx.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/ioctl.h"
#include "sys/tty.h"
#include "sys/ttyx.h"
#include "sys/proc.h"
#include "sys/procx.h"
#include "sys/dm11.h"

#ifndef NDZ11
#define NDZ11 16                /* number of dz11's */
#endif
#ifndef DZADDR
#define DZADDR 0160140          /* start address of first dz11 */
#endif
#define DZENABL 050140          /* enable dz11 hardware */
#ifndef DZSRATE
#define DZSRATE 2               /* dz11 scan rate */
#endif
#define SSPEED 7                /* standard speed 300 baud rate */
#define RECV_ON 010000          /* enable receiver bit */
#define PENABLE 000100          /* parity enable mask */
#define OPAR 0200               /* odd parity mask */
#define B6BITS 010              /* 6 bit code mask */
#define B7BITS 020              /* 7 bit code mask */
#define B8BITS 030              /* 8 bit code mask */
```

```
#define STOP2 040               /* 2 stop bits */

/*  structures for dz11 hardward registers.
 *  there are 2 structures the first one is for write only registers
 *  and the second one is for read/write registers. this is necessary
 *  because the dz11 shares register addresses.
*/

struct dzregs
{
        int  dzcsr;             /* control and status registers */
        int  dzrbuf;            /* receiver buffer register */
        char dztcr;             /* transmitter control register */
        char dzdtr;             /* data terminal ready register */
        char dzring;            /* ring register */
        char dzcar;             /* carrier register */
} ;
struct
{
        int  dzcsr;             /* control and status register */
        int  dzlpr;             /* line parameter register */
        char dztcr;             /* transmitter control register */
        char dzdtr;             /* data terminal ready register */
        char dztbuf;            /* transmitter buffer */
        char dzbrk;             /* break register */
} ;

/* this array maps the standard unix transmit and receive speeds
 * into the speeds used in the dz11 hardware. If a speed is selected
 * that is not available on the dz11 it will be ignored.
*/

char dzspeed[]
{
        00000,          /*  not available */
        00000,          /*    50 baud */
        00001,          /*    75 baud */
        00002,          /*   110 baud */
        00003,          /* 134.5 baud */
        00004,          /*   150 baud */
        00000,          /*  not available */
        00005,          /*   300 baud */
        00006,          /*   600 baud */
        00007,          /*  1200 baud */
        00010,          /*  1800 baud */
        00012,          /*  2400 baud */
        00014,          /*  4800 baud */
        00016,          /*  9600 baud */
        00000,          /*  not available */
        00000,          /*  not available */
};
int ndz11 NDZ11;        /* number of dz11 lines */
struct tty dz11[NDZ11]; /* tty structers */
int     dzopenf;        /* open flag */
char sdzbrk[(NDZ11+7)/8];       /* software copy of brk register */
```

```
/* open a dz11 line */

dzopen(dev,flag)
{
        register struct tty *tp;
        register int line;
        register struct dzregs *dzaddr;

#ifdef PWR_FAIL
        extern unsigned pwr_fail;

        if (dev == NODEV) {
                if (pwr_fail)
                        return;
                pwr_init(dz11, NDZ11, 0);
                dzaddr = DZADDR;
                for (line=0; line < (NDZ11+7)/8; line++, dzaddr++)
                        if (dzopenf & (1<<line)) {
                                dzaddr->dztcr = 0377;
                                dzxint(line);
                        }
                return;
        }
#endif

        if((line = dev.d_minor) >= NDZ11)
        {
                u.u_error = ENXIO;
                return;
        }
        tp = &dz11[line];
        dzcntrl(dev,1);
        if((tp->t_state & ISOPEN) == 0)
        {
                if((tp->t_state & EVEROPEN) == 0)
                {
                        dzopenf |= (1 << (dev.d_minor>>3));
                        tp->t_speeds = SSPEED | (SSPEED << 8);
                        tp->t_flags = ANYP|CRMOD|XTABS|ECHO|NCDELAY|NLDELAY;
                }
                dzparam(dev);
        }
        (*linesw[tp->t_ltype].l_open)(tp);
}

/* control a dz11 line */

dzcntrl(dev,action)
{
        register struct tty *tp;
        register struct dzregs *dzadr;
        extern dzstart(),dzscan();
        static scan;

        tp = &dz11[dev.d_minor];
        tp->t_dev = dev;
```

```
        tp->t_state =| SSTART;
        tp->t_addr = dzstart;
        dzadr=DZADDR+((dev.d_minor>>3) * sizeof(*dzadr));
        dzadr->dzcsr =| DZENABL;
        if(action)
                dzmcntrl(dev,'s',CDLEAD);
        else
                dzmcntrl(dev,'c',CDLEAD);
        if(dzmcntrl(dev, 't', CARRIER))
                tp->t_state =| CARR_ON;
        else
                tp->t_state =& ~CARR_ON;
        if (scan == 0)
        {
                timeout(&dzscan,0,DZSRATE);
                scan++;
        }
}

/* set dz11 line control register */

dzparam(dev)
{
        struct {
                int     low12:12;
                int     top4:4;
        } ;
        struct {
                int     low12:12;
                int     sdbits:3;
        } ;
        register struct tty *tp;
        register struct dzregs *dzadr;
        register int lpr;

        tp = &dz11[dev.d_minor];
        dzadr = DZADDR + ((dev.d_minor>>3) * sizeof(*dzadr));
        spl5();
        lpr = ((dzspeed[tp->t_speeds.lobyte]<<8) | ((dev.d_minor&07) |
                        RECV_ON));
        if(tp->t_speeds.lobyte != 2)
                if((tp->t_flags&(EVENP|ODDP))==EVENP) {
                        lpr =| B7BITS|PENABLE;
                } else {
                        if((tp->t_flags&(EVENP|ODDP))==ODDP) {
                                lpr =| B7BITS|OPAR|PENABLE;
                        } else {
                                lpr =| B8BITS;
                        }
                }

        if(tp->t_speeds.lobyte <= 3)            /* 110 baud */
                lpr =| STOP2;

        if(tp->t_speeds < 0) {
                lpr =& ~(B8BITS | STOP2);
```

```
                lpr =| tp->t_speeds.sdbits<<3;
        }
        tp->t_speeds.top4 = ((lpr & (B8BITS | STOP2))>>3);

        dzadr->dzlpr = lpr;
        spl0();
}

/* set or read dz11 line parameters */

dzioctl(dev, cmd, addr, flag)
caddr_t addr;
{
        register struct tty *tp;
        register flg;

        tp = &dz11[dev.d_minor];
        flg = 0;
        if (cmd == OLDSGTTY) {
                flg = lsgtty(addr, tp);
        } else if(ttioccomm(cmd, tp, addr, dev)) {
                if (cmd==TIOCSETP || cmd==TIOCSETN || cmd==TIOCSETO)
                        flg++;
        } else
                u.u_error = ENOTTY;
        if (flg)
                dzparam(dev);
}

/* control dm11-the dz11 does not have separate dm hardware
 *              so the subroutine uses the dz11 registers
*/

dzmcntrl(dev,flag,bits)
{
        register struct dzregs *dzadr;
        register int dtr;
        register int mask;
        int carrier;

        dzadr = DZADDR + ((dev.d_minor>>3) * sizeof(*dzadr));
        mask = 1 << (dev.d_minor&07);
        carrier = 0;
        if((bits&CARRIER) && (dzadr->dzcar&mask))
                carrier = CARRIER;
        dtr = 0;
        if(bits&CDLEAD) {
                switch(flag)
                {
                        case 's':
                                dtr = (dzadr->dzdtr =| mask);
                                break;
                        case 'c':
                                dtr = (dzadr->dzdtr =& ~mask);
                                break;
                        case 't':
```

```
                                dtr = dzadr->dzdtr;
                                break;
                        }
                        if(dtr&mask)
                                dtr = CDLEAD;
                        else
                                dtr = 0;
                }
                return(carrier|dtr);
}

/* scan the carrier register and receiver silo */

dzscan ()
{
        register struct tty *tp;
        register struct dzregs *dzadr;
        register int n;
        int i;
        static int halfsec = 0; /* scan carrier only every half second */

        tp = &dz11;
        dzadr = DZADDR;

        /* scan carrier only once per half second */
        if (halfsec == 0) {

#if     (HZ / (DZSRATE * 2))
        halfsec = (HZ / (DZSRATE * 2)) - 1;
#endif

        for(i=0; i <((NDZ11+7)/8); i++, dzadr++)
            for(n=1; n<=0200; n =<< 1, tp++)
                if ((dzopenf&(1<<i)) && (((dzadr->dzcar&n)==n)
                                        ^ ((tp->t_state&CARR_ON)==CARR_ON)))
                        (*linesw[tp->t_ltype].l_dst)
                                (tp,CTRANS,(dzadr->dzcar & n) ? CARRIER : 0);

        } else halfsec -= 1;

        dzrint();
        timeout(dzscan,0,DZSRATE);
}

/* close a dz11 line */

dzclose(dev)
{
        register struct tty *tp;

        tp = &dz11[dev.d_minor];
        (*linesw[tp->t_ltype].l_close)(dev, tp);
        if(!(tp->t_flags&NOHUP))
                dzmcntrl(dev,'c',CDLEAD);
}
```

```
/* read a dz11 line */

dzread(dev)
{
        register struct tty *tp;

        tp = &dz11[dev.d_minor];
        (*linesw[tp->t_ltype].l_read) (tp);
}

/* write a dz11 line */

dzwrite(dev)
{
        register struct tty *tp;

        tp = &dz11[dev.d_minor];
        (*linesw[tp->t_ltype].l_write) (tp);
}

/* dz11 receiver interrupt handler */

dzrint()
{
        register struct tty *tp;
        register struct dzregs *dzadr;
        register int c;
        register int dev;

        dzadr = DZADDR;
        for(dev=0;dev < ((NDZ11+7)/8);dev++)
        {
                if(dzopenf&(1<<dev))
                        while((c = dzadr->dzrbuf) < 0)
                        {
                                tp = &dz11[((c >> 8) & 007) + (dev << 3)];
                                if(tp >= &dz11[NDZ11])
                                        continue;
                                if(((c&0177)==XON)||((c&0177)==XOFF)) {
                                        if (tp->t_flags&TANDEMI) {
                                                dzxoff(tp, c&0177);
                                                continue;
                                        }
                                }
                                (*linesw[tp->t_ltype].l_rcvd) (c,tp);
                        }
                dzadr++;
        }
}

/* dz11 suspend/restore output on XOFF/XON */

dzxoff(tp, c)
register struct tty *tp;
{
        register struct dzregs *dzadr;
```

```
        register linebit;

        dzadr = DZADDR + (tp->t_dev.d_minor>>3) * sizeof(*dzadr);
        linebit = 1 << (tp->t_dev & 07);
        if (c==XON) {
                tp->t_state &= ~XMTXOFF;
                dzadr->dztcr |= linebit;
        } else {
                dzadr->dztcr &= ~linebit;
                tp->t_state |= (BUSY|XMTXOFF);
        }
}

/* dz11 transmit interrupt routine */

dzxint(dev)
{
        register struct tty *tp;
        register struct dzregs *dzadr;
        register int c;
        struct tty *btp;
        int lineno;
        extern dzrstrt();

        btp = &dz11[dev.d_minor << 3];
        dzadr = DZADDR + dev.d_minor * sizeof(*dzadr);
        while(dzadr->dzcsr < 0)
        {
                lineno = ((dzadr->dzcsr >> 8) & 007);
                tp = btp + lineno;
                c = (*linesw[tp->t_ltype].l_xmtd) (tp, 0);
                if(c < 0)
                {
                        dzadr->dztbuf = c;
                        continue;
                }
                if(c & CBREAK)
                {
                        sdzbrk[dev.d_minor] =| (1<<lineno);
                        dzadr->dzbrk = sdzbrk[dev.d_minor];
                }
                if(c & (CBREAK | CTOUT))
                {
                        timeout(dzrstrt,tp,(c & 0177)+chrdelay[tp->t_speeds
                                .hibyte & 017]);
                        tp->t_state =| TIMEOUT;
                }
                if(c == 0)
                        (*linesw[tp->t_ltype].l_xmtd) (tp, 1);

                dzadr->dztcr =& ~(1 << lineno);
                tp->t_state &= ~BUSY;
        }
}

/* start a transmission on a dz11 line */
```

```
dzstart(atp)
struct tty *atp;
{
        register struct tty *tp;
        register struct dzregs *dzadr;
        register int lineno;

        tp = atp;
        if((tp->t_state &(TIMEOUT|BUSY)) == 0)
        {
                lineno = tp - dz11;
                dzadr = DZADDR + (lineno>>3) * sizeof(*dzadr);
                lineno = 1 << (lineno&07);
                dzadr->dztcr =| lineno;
                tp->t_state |= BUSY;
        }
}

/* restart a transmission after a break or timeout */

dzrstrt(atp)
struct tty *atp;
{
        register struct tty *tp;
        register struct dzregs *dzadr;
        register int lineno;

        tp = atp;
        lineno = tp - dz11;
        dzadr = DZADDR + (lineno>>3) * sizeof(*dzadr);
        sdzbrk[lineno>>3] =& ~(1<<(lineno&07));
        dzadr->dzbrk = sdzbrk[lineno>>3];
        ttrstrt(tp);
}
```