

```

/*      @(#)lfh.c      2.10      */
#
/* .n'lfh' Includes, Defines, and Data Declarations' */

/*
 *      LOGICAL FILE SYSTEM HANDLER.
 *
 *      initial version
 *      written by:      J. C. Kaufeld      3/23/77
 *
 *      enhanced version
 *      written by:      J. R. McSkimin    1/17/79
 */

#include "sys/param.h"
#include "sys/buf.h"
#include "sys/bufx.h"
#include "sys/conf.h"
#include "sys/confx.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/proc.h"
#include "sys/systm.h"
#include "sys/reg.h"
#include "sys/vtmm.h"
#include "sys/lfsh.h"

#define FREE      1
#define INUSE     0
#define LFPRI     -30

struct lflayout lfll;          /* file system global data */

char lfflag;                  /* lf handler state */
int lfdev = ((5<<8)|4);      /* LFH device */
struct lfdsc phead;           /* translation table header */

static int blockf;            /* # of sectors per allocation */
static int lfsize;            /* size of logical file header */
int fdblk = (SECSIZE/sizeof(phead)); /* # lfdsc per sector */
#ifdef LFSINS
static struct lfstat lfstat;
#endif
/* .s'lfopen' Open logical file system' */
lfopen(dev, flag)
{
    register struct lfhead *lhp;
    register struct lfdsc *pf, *qf;

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

    lflock();

```

```

#ifdef LFSINS
    lfstat.lf_lfscal++;
#endif
    if((lfflag & LF_OPEN) == 0)
    {
        lfiio(SECOFST, &lfll, sizeof(*lhp), B_READ);
        if(u.u_error || lfll.lh_magic != LFMAGIC || lfll.lh_nlfs > MAXLFS)
        {
            if(u.u_error == 0)
                u.u_error = EACCES;
        }
        else
        {
            blockf = lfll.lh_blkf;
            lfflag |= LF_OPEN;
            /* initialize translation table memory ptrs */
            phead.ld_lfn = 0; /* end of chain flag */
            phead.ld_forw = phead.ld_back = &phead;
            qf = &phead;
            pf = lfll.lfds;
            while(pf < &lfll.lfds(MAXLFD1))
            {
                pf->ld_lfn = pf->ld_forw = pf->ld_back = 0;
                pf->ld_flag = 0;
                qf->ld_avforw = pf;
                pf->ld_avback = qf;
                qf = pf;
                pf++;
            }
            phead.ld_avback = qf;
            qf->ld_avforw = &phead;
        }
    }
    lfrlse();
}

/* .s'lfclose' Close logical file system */
lfclose(dev, flag)
{
    lfflag = 0;
#ifdef LFSINS
    lfstat.lf_lfscal++;
#endif
}
/* .s'lfioctl' User request handler */
lfioctl(dev, cmd, addr, flag)
register int cmd;
caddr_t addr;
{
    register struct lfdsc *lp1, *lp2;
    struct lfcb lfcb;
    caddr_t i;

#ifdef LFSINS
    lfstat.lf_lfscal++;

```

```

#endif
if (copyin(addr, (caddr_t)&lfcb, sizeof(lfcb))) {
    u.u_error = EFAULT;
    return;
}
switch (cmd) {
    case L_READ:
    case L_WRITE:
        lfget((cmd==L_READ) ? B_READ: B_WRITE, &lfcb);
        lfret(&lfcb, addr);
        return;

    case L_CREAT:
    case L_DELETE:
        lfmake(cmd, &lfcb);
        lfret(&lfcb, addr);
        return;

    case L_SWITCH:
        lp1 = getlfd(lfcb.lf_lfn);
        lp2 = getlfd(lfcb.lf_arg1);
        if(u.u_error || !(lfopn(lp1) && lfopn(lp2))) {
            if(lp1) remlfd(lp1);
            if(lp2) remlfd(lp2);
            return;
        }
        lflock();
        cmd = lp1->ld_start;
        lp1->ld_start = lp2->ld_start;
        lp2->ld_start = cmd;

        cmd = lp1->ld_size;
        lp1->ld_size = lp2->ld_size;
        lp2->ld_size = cmd;

        cmd = lp1->ld_secsz;
        lp1->ld_secsz = lp2->ld_secsz;
        lp2->ld_secsz = cmd;

        putlfd(lfcb.lf_lfn, lp1);
        putlfd(lfcb.lf_arg1, lp2);
        lfrlse();
        return;

    case L_COPY:
        /* NOT YET IMPLEMENTED */
        u.u_error = EINVAL;
        return;

    case L_SIZE:
        lp1 = getlfd(lfcb.lf_lfn);
        if(u.u_error || !lfopn(lp1))
            {
                if(lp1) remlfd(lp1);
                return;
            }

```

```

    }

    lfcb.lf_lfn = lp1->ld_secsz;
    lfret(&lfcb, addr);
    return;

#ifdef LFSINS
    case L_ISTAT:
        for(i = &lfstat;
            i < (caddr_t)&lfstat + sizeof(lfstat);
            *i++ = 0);
        return;

    case L_STAT:
        if(copyout((caddr_t)&lfstat, addr, sizeof(lfstat)))
            u.u_error = EFAULT;

        return;

#endif
    default:
        u.u_error = EINVAL;
        return;
}

/*s'lfio'Do I/O into lf core area'*/
static int lfio(blkno, coreaddr, bytes, rdflg)
char *coreaddr, *blkno;
{
    register struct buf *bp;

    bp = getbfh();
    bp->b_flags |= B_BUSY | rdflg;
    bp->b_dev = lfdev;
    bp->b_blkno = blkno;
    bp->b_bcount = bytes;
    bp->b_paddr = coreaddr;
    bp->b_error = 0;
    (*bdevsw[lfdev]>>81.d_strategy)(bp);
    spl6();
    while((bp->b_flags & B_DONE) == 0)
        sleep(bp, PRIBIO);
    spl0();
    if(bp->b_resid)
        if(u.u_error == 0)
            u.u_error = EIO;
    geterror(bp);
    hrelease(bp);
#ifdef LFSINS
    lfstat.lf_brc++;
#endif
    return;
}

/*s'lfget'Get/put a block in a file'*/
static int lfget(rwflag, lp)

register struct lfcb *lp;

```

```

(
    register struct lfdsc *ldp;

    if((ldp=getlfd(lp->lf_lfn)) == 0)
        return;
    if(lfopn(ldp) == 0)
        return;

    /* convert block number to byte offset */
    u.u_offset = ((long)(ldp->ld_start*blockf) + lp->lf_arg1) << SECSIZ2;
    u.u_base = lp->lf_arg2;
    if(ldp->ld_secsiz < lp->lf_arg1 + lp->lf_arg3)
    {
        if(ldp->ld_secsiz <= lp->lf_arg1)
            if(rwflag & B_READ) {
                lp->lf_lfn = 0;
                return;
            } else {
                u.u_error = EIO;
                return;
            }
        lp->lf_arg3 = ldp->ld_secsiz - lp->lf_arg1;
    }
    u.u_count = lp->lf_arg3 * SECSIZE;
    physio(*bdevsw[lfddev]>>8).d_strategy, lfddev, rwflag, 0);
    if(u.u_error != 0 || u.u_count) {
        if(u.u_error == 0)
            u.u_error = ENXIO;
        return;
    }
    lp->lf_lfn = lp->lf_arg3;

#ifdef LFSINS
    if(rwflag == B_READ)
    {
        lfstat.lf_rawrc++;
        lfstat.lf_rawrb += lp->lf_arg3;
    }
    else
    {
        lfstat.lf_rawwc++;
        lfstat.lf_rawwb += lp->lf_arg3;
    }
#endif
}

/* .s'getlfd' Return pointer to logical file descriptor */
/* retrieve lf descriptor from translation table or from disk,
   as appropriate and return pointer to caller.
*/
static int getlfd(lfn)
register int lfn;
{
    register struct lfdsc *ldp;
    struct lfdsc *bfp;

```

```

    register struct buf *bp;
    if(lfn <= 0 || lfn > lf11.lh_nlfs) {
        u.u_error = EINVAL;
        return(0);
    }
    if(lfdp = inttm(lfn)) return(lfdp);    /* in core */
    bp = abread(lfdev, lfn/fdblk+FDOFST);
#ifdef LFSINS
    lfstat.lf_brc++;
#endif
    bfp = bp->b_paddr + lfn*fdblk * sizeof(*lfdp);
    lfdp = gettm(lfn);    /* allocate descriptor */

    /* copy other stuff from bfp to lfdp */
    lfdp->ld_lfn = lfn;
    lfdp->ld_flag = bfp->ld_flag;
    lfdp->ld_start = bfp->ld_start;
    lfdp->ld_size = bfp->ld_size;
    lfdp->ld_secsiz = bfp->ld_secsiz;
    brelse(bp);
    return(lfdp);
}

```

```

/*s'putlfd'Put logical file descriptor to disk' */
/* Write lf descriptor to disk, but leave in translation table.
*/

```

```

static int putlfd(lfn, lfdp)

```

```

register int lfn;
register struct lfdsc *lfdp;
{
    struct lfdsc *bfp;
    register struct buf *bp;

    bp = abread(lfdev, lfn/fdblk+FDOFST);
    bfp = bp->b_paddr + lfn*fdblk * sizeof(*lfdp);
    copy(lfdp, bfp, sizeof(*lfdp));
    bawrite(bp);
#ifdef LFSINS
    lfstat.lf_brc++;
    lfstat.lf_bwc++;
#endif
}

```

```

/*s'lfmake'Create/delete logical files'*/
static int lfmake(type, lp)

```

```

struct lfcb *lp;
{
    register struct lfdsc *lpl;
    register int lfn, newsize;
    char *size, *start;

    lfn = lp->lf_lfn;
    if((type == L_DELETE && lfn == 0) || ((newsize=lp->lf_arg1) < 0)) {

```

```

        u.u_error = EINVAL;
        return;
    }
    iflock();
    if(lfn) {
        if((lpl=getlfd(lfn)) == 0) {
            ifrlse();
            return;
        }
    } else {
        /* search descriptors for temp file */
        for(lfn=lf11.lh_nlf1; lfn>0; lfn--)
        {
            lpl = getlfd(lfn);
            if(lpl->ld_flag == 0)
                break;
            remlfd(lpl);
        }

        if(lfn == 0)
        {
            u.u_error = ENFILE;
            ifrlse();
            return;
        }
    }

    if((lpl->ld_secsz>=newsize)&&(lpl->ld_secsz!=0||type==L_DELETE)) {
        /*
         * Delete file or file gets smaller.
         */

        size = lpl->ld_size;
        lpl->ld_size = (newsize+(blockf-1))/blockf;
        start = lpl->ld_start + lpl->ld_size;
        size -= lpl->ld_size;
        if(newsize == 0 && type == L_DELETE)
            remlfd(lpl);
        lffree(start, size);
        lpl->ld_secsz = newsize;
    } else {
        /*
         * New file or file gets larger.
         */

        size = (newsize+(blockf-1))/blockf;
        if(size > lpl->ld_size) {
            /*
             * New file or file will not fit in current
             * number of blocking factors.
             */

```

```

        if((start=lfalloc(size)) == 0) {
            remlfd(lpl);
            lfrlse();
            return;
        }
        if((size > lpl->ld_size) && lpl->ld_secsiz) {
            lfcopy(lpl->ld_start*blockf,
                start*blockf, lpl->ld_secsiz);
            lffree(lpl->ld_start, lpl->ld_size);
        }
        lpl->ld_start = start;
    }
    lpl->ld_flag    = | LF_OPEN;
    lpl->ld_size    = size;
    lpl->ld_secsiz  = newsiz;
}
putlfd(lfn, lpl);
lfrlse();
lp->lf_lfn = lfn;
}
/*s' lffree' Return space to free list'*/
static int lffree(start, size)
int start;
register int size;
{
    register int *bfp;
    register struct buf *bp;

    if(size < 0) {
        u.u_error = ENOSPC;
        return;
    }
    bp = ahead(lfdev, lfll.lh_fres);
    bfp = bp->b_paddr;
    mfree(bfp, size, start);
    bfp[254] = bfp[255] = 0;      /* prevent overwrites */
    bawrite(bp);
#ifdef LFSINS
    lfstat.lf_brc++;
    lfstat.lf_bwc++;
#endif
    lfspace(start, size, FREE);
}

/*s' lfalloc' Get space from free list'*/
static int lfalloc(size)
register int size;
{
    register int *bfp;
    register struct buf *bp;
    int start;

    if(size < 0) {
        u.u_error = EIO;
    }

```



```

        return(0);
    }
    bp = abread(lfdev, lf11.lh_fres);
#ifdef LFSINS
    lfstat.lf_brc++;
#endif
    bfp = bp->b_paddr;
    if((start=malloc(bfp,size)) == 0) {
        u.u_error = ENOSPC;
        brelse(bp);
        return(0);
    }
    bawrite(bp);
#ifdef LFSINS
    lfstat.lf_bwc++;
#endif
    lfspace(start,size,INUSE);
    return(start);
}

/*s'lfspace'Reset bit in free space bit map*/
static int lfspace(start,size,on)
register int start;
{
    register int word,*bfp;
    int block,nblock;
    struct buf *bp;

    block = 0;
    while(size-->0) {
        nblock = start/(SECSIZE<<3) + lf11.lh_fres + 1;
        word = (start*(SECSIZE<<3))>>4;
        if(nblock != block) {
            if(block)
            {
                bawrite(bp);
#ifdef LFSINS
                lfstat.lf_bwc++;
#endif
            }

            bp = abread(lfdev,block=nblock);
#ifdef LFSINS
            lfstat.lf_brc++;
#endif
        }
        bfp = bp->b_paddr;
        *(bfp+word) = & ~(1 << (start&017));
        *(bfp+word) |= (on << (start&017));
        start++;
    }
    if(block)
    {
        bawrite(bp);
#ifdef LFSINS

```

```

        lfstat.lf_bwc++;
#endif
    )
}
/*s'lfcopy'Copy sectors'*/
static int lfcopy(fblk,tblk,nblks)
register int nblks,fblk;
{
    register struct buf *bp;
    while(nblks--) {
        bp = bread(lfdev,fblk++);
        bp->b_blkno = tbk++;
        bawrite(bp);
        /* non-addressable ok */
    }
#ifdef LFSINS
    lfstat.lf_brc++;
    lfstat.lf_bwc++;
#endif
}

/*s'lflock'Protect against multiple accessors'*/
static int lflock()
{
    register int sps;
    sps = PS->integ;
    spl6();
    while(lfflag&LF_BUSY) {
        lfflag |= LF_WANT;
        sleep(&lfflag,LFPRI);
    }
    lfflag |= LF_BUSY;
    PS->integ = sps;
}

/*s'lfrlse'Release protection on lf system'*/
static int lfrlse()
{
    register int sps;
    sps = PS->integ;
    spl6();
    if(lfflag&LF_WANT) {
        lfflag = & ~LF_WANT;
        wakeup(&lfflag);
    }
    lfflag = & ~LF_BUSY;
    PS->integ = sps;
}

/*s'lfret'Put a real return argument in 1st field of users structure
because ioctl only returns a 0 or a 1 */

```

```
static int lfret(lfcb1,addr)
struct lfcb *lfcb1;
caddr_t addr;
{
    if(copyout((caddr_t)lfcb1,addr,sizeof(struct lfcb)))
        u.u_error = EFAULT;
    return;
}
```

```
/*s'copy'Copies one buffer to another' */
static int copy(from, to, numb)
register char *from, *to;
register int numb;
```

```
{
    while(numb--)
        *to++ = *from++;
}
```

```
/*s'lfopn'Check for open Logical File'*/
```

```
static int lfopn(lp)
register struct lfdsc *lp;
{
    if((lp->ld_flag&LF_OPEN) == 0) {
        if(u.u_error == 0)
            u.u_error = EBADF;
        return(0);
    }
    return(1);
}
```

```
/*s'gettm'Allocate translation table entry' */
```

```
static int gettm(lfn)
register int lfn;
{
    register struct lfdsc *qf;

    qf = phead.ld_avforw;

    /* unlink from front of avail list and put on back
       to affect lru strategy */
    avlink(qf,&phead);

    /* unlink from lfn list */
    if(qf->ld_forw)
    {
        (qf->ld_forw)->ld_back = qf->ld_back;
        (qf->ld_back)->ld_forw = qf->ld_forw;
    }

    /* insert on lfn list in proper slot */
    ittm(qf,lfn);
}
```

```

    return(qf);
}

/*s'ittm'Insert lfd on lfn list' */
static int ittm(qf,lfn)
register struct lfdsc *qf;
register int lfn;
{
    register struct lfdsc *pf;

    /* start scan at most efficient place */
    pf = (qf->ld_lfn > lfn ? qf->ld_back : phead.ld_back);

    /* search backwards so it works when list empty (ld_lfn=0) */
    while(pf->ld_lfn > lfn)
        pf = pf->ld_back;

    /* now pf points to entry just prior to proper insertion pt */
    qf->ld_forw = pf->ld_forw;
    qf->ld_back = pf;
    (pf->ld_forw)->ld_back = qf;
    pf->ld_forw = qf;
}

/*s'inttm'Determine if lf descriptor is in ttm' */
static int inttm(lfn)
register int lfn;
{
    register int tlfn;
    register struct lfdsc *qf;

    /* search translation table memory to see if lfn in core */
    qf = phead.ld_forw;
    while(tlfn = qf->ld_lfn)        /* check for 0 lfn */
    {
        if(tlfn == lfn)            /* found it */
        {
            /* unlink from avail list and put on end
             to affect lru strategy */
            avlink(qf,&phead);

#ifdef LFSINS
            lfstat.lf_ttmhit++;
#endif

            return(qf);
        }
        if(tlfn > lfn) break;
        qf = qf->ld_forw;
    }

#ifdef LFSINS
    lfstat.lf_ttmmiss++;
#endif
    return(0);
}

/*s'avlink'Remove lfd from avail list and link' */
static int avlink(qf,af)
register struct lfdsc *qf, *af;

```

```

{
    /* unlink qf from avail list */
    (qf->ld_avback)->ld_avforw = qf->ld_avforw;
    (qf->ld_avforw)->ld_avback = qf->ld_avback;

    /* link before af entry on avail list */
    qf->ld_avforw = af;
    qf->ld_avback = af->ld_avback;
    (af->ld_avback)->ld_avforw = qf;
    af->ld_avback = qf;
}
/*.*remldf remove lfd from ttm'*/
static int remldf(qf)
register struct lfdsc *qf;
{
    /* unlink from lfn list */
    if(qf->ld_forw)
    {
        (qf->ld_forw)->ld_back = qf->ld_back;
        (qf->ld_back)->ld_forw = qf->ld_forw;
    }
    qf->ld_forw = qf->ld_back = qf->ld_lfn = 0;
    qf->ld_flag = 0;

    /* link onto front of avail list */
    avlink(qf, phead.ld_avforw);
}

```

```
/*      @(#)lp.c      2.3      */
#
/*
*/

/*
 * LP-11 Line printer driver
 */

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

#define LPADDR 0177514

#define IENABLE 0100
#define DONE 0200
#define ERROR 0100000

#define LPDELAY 60
#define LPPRI 10
#define LPLWAT 50
#define LPHWAT 200
#define EJLINE 60
#define MAXCOL 80

struct {
    int lpsr;
    int lpbufl;
};

struct {
    int cc;
    int cf;
    int cl;
    int flag;
    int mcc;
    int ccc;
    int mlc;
} lp11;

#define CAP 01
#define EJECT 02
#define OPEN 04
#define IND 000
#define LPBUSY 020

#define FORM 014

lpopen(dev, flag)
register dev, flag;
{
    lp11.flag |= CAP;

/* Upper Case only */
/* Set to 0 for no indent, else to 010 */
/* timeout entry flag */
}
```

```

lpopen1(dev, flag);
}

lpopen1(dev, flag)
{
#ifdef PWR_FAIL
    extern unsigned pwr_fail;

    if (dev == NODEV) {
        if (pwr_fail == NULL && (lp11.flag&OPEN)) {
            lp11.flag = & ~LPBUSY;
            LPADDR->lpsr = IENABLE;
            lpint();
        }
        return;
    }
#endif

    if (lp11.flag & OPEN) {
        u.u_error = EIO;
        return;
    }
    lp11.flag = I (IND|OPEN);
    LPADDR->lpsr = IENABLE;
    lp11.canon(FORM);
}

lp11.close(dev, flag)
{
    lp11.canon(FORM);
    lp11.flag = & ~(OPEN|CAP|IND|EJECT);
}

lp11.write()
{
    register int c;

    while ((c=cpass())>=0)
        lp11.canon(c);
}

lp11.canon(c)
{
    register cl, c2;

    cl = c;
    if (lp11.flag&CAP) {
        if (cl>='a' && cl<='z')
            cl = 'A'-'a'+cl;
        switch (cl) {
            case '[':
                c2 = '(';
                goto esc;
            case ']':

```

```

        c2 = ')';
        goto esc;

    case '^':
        c2 = '\^';
        goto esc;

    case '|':
        c2 = '|';
        goto esc;

    case '~':
        c2 = '~';

    esc:
        lpCanon(c2);
        lp11.ccc--;
        c1 = '-';
    }
}

switch(c1) {
    case '\t':
        lp11.ccc = (lp11.ccc+8) & ~7;
        return;

    case FORM:
    case '\n':
        if((lp11.flag&EJECT) == 0 ||
            lp11.mcc!=0 || lp11.mlc!=0) {
            lp11.mcc = 0;
            lp11.mlc++;
            if(lp11.mlc >= EJLINE && lp11.flag&EJECT) {
                c1 = FORM;
                lpoutput(c1);
                if(c1 == FORM)
                    lp11.mlc = 0;
            }
        }

    case '\r':
        lp11.ccc = 0;
        if(lp11.flag&IND)
            lp11.ccc = 8;
        return;

    case 010:
        if(lp11.ccc > 0)
            lp11.ccc--;
        return;

    case ' ':
        lp11.ccc++;
        return;

    default:

```



```

        if(lp11.ccc < lp11.mcc) {
            lpoutput('\r');
            lp11.mcc = 0;
        }
        if(lp11.ccc < MAXCOL) {
            while(lp11.ccc > lp11.mcc) {
                lpoutput(' ');
                lp11.mcc++;
            }
            lpoutput(c1);
            lp11.mcc++;
        }
        lp11.ccc++;
    }
}

lprstrt()
{
    lp11.flag = & ~LPBUSY;
    lpstart();
}

lpstart()
{
    register int c;
    int lprstrt();

    if(lp11.flag & LPBUSY)
        return;

    lp11.flag = ! LPBUSY;
    if(LPADDR->lpsr&ERROR) {
        timeout(lprstrt, 0, LPDELAY);
        return;
    }
    while (LPADDR->lpsr&DONE && (c = getc(&lp11)) >= 0)
        LPADDR->lpbuf = c;

    lp11.flag = & ~LPBUSY;
}

lpint()
{
    lpstart();
    if (lp11.cc == LPLWAT || lp11.cc == 0)
        wakeup(&lp11);
}

lpoutput(c)
{
    if (lp11.cc >= LPHWAT)
        sleep(&lp11, LPPRI);
    putc(c, &lp11);
    spl4();
    lpstart();
}

```

sp10();

2

```

/*      @(#)lpm.c      2.3      */

/*
 * LP-11 Multiple Line printer drive
 */

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

#define NLPR      1                /* Number of LP11 Printers -- Must have
                                   corresponding entries in struct lpr */

#define IENABLE 0100
#define DONE    0200
#define ERROR    0100000

#define LPDELAY 60
#define LPPRI   10
#define LPLWAT  50
#define LPHWAT  200
#define EJLINE  60
#define MAXCOL  132

struct {
    int lpsr;
    int lpbuf;
};

struct {
    int    cc;
    int    cf;
    int    cl;
    int    flag;
    int    mcc;
    int    ccc;
    int    mlc;
} lp11[NLPR];

#define CAP      01
#define EJECT    02
#define OPEN     04
#define IND      000    /* Set to 0 for no indent, else to 010 */
#define LPBUSY   020    /* timeout entry flag */

#define FORM     014

/*
 * LP-11 hardware address and upper/lower case option table
 * One entry for each LPR.
 */
struct lpr {
    int    *addr;

```

```

    int    lpcase;
} lprtab[NLPR] {
    0177514, 0,
},
lpclose(dev, flag)
{
    register struct lpr *lp;
    register lpr;

#ifdef PWR_FAIL
    extern unsigned pwr_fail;

    if (dev == NODEV) {
        if (pwr_fail == NULL)
            for (lpr=0; lpr<NLPR; lpr++) {
                lp = &lprtab[lpr];
                if (lp->flag & OPEN) {
                    lprtab[lpr].addr->lpsr = IENABLE;
                    lp->flag = ~LPBUSY;
                    lpint(lpr);
                }
            }
        return;
    }
#endif

    if ((lpr = dev.d_minor) >= NLPR) {
        u.u_error = ENXIO;
        return;
    }

    lp = &lprtab[lpr];

    if (lp->flag & OPEN) {
        u.u_error = EIO;
        return;
    }

    lp->flag = lprtab[lpr].lpcase | OPEN;
    lprtab[lpr].addr->lpsr = IENABLE;
    lpcanon(lpr, FORM);
}

lpclose(dev, flag)
register dev, flag;
{
    register lpr;

    lpr = dev.d_minor;
    lpcanon(lpr, FORM);
    lprtab[lpr].flag = 0;
}

lpwrite(dev)
register dev;

```

```

{
    register int c;

    while ((c=cpass())>=0)
        lpcanon(dev.d_minor, c);
}

lpcanon(lpr, c)
{
    register struct lpr *lp;
    register c1, c2;

    lp = &lp1[lpr];
    c1 = c;
    if(lp->flag&CAP) {
        if(c1>='a' && c1<='z')
            c1 =+ 'A'-'a'; else
            switch(c1) {

                case '[':
                    c2 = '(';
                    goto esc;

                case ']':
                    c2 = ')';
                    goto esc;

                case '^':
                    c2 = '\^';
                    goto esc;

                case '|':
                    c2 = '|';
                    goto esc;

                case '~':
                    c2 = '~';

                esc:
                    lpcanon(lpr, c2);
                    lp->ccc--;
                    c1 = '-';
            }

        switch(c1) {

            case '\t':
                lp->ccc = (lp->ccc+8) & ~7;
                return;

            case FORM:
            case '\n':
                if((lp->flag&EJECT) == 0 ||
                    lp->mcc!=0 || lp->mlc!=0) {
                    lp->mcc = 0;
                }
            }
        }
    }
}

```

```

        lp->mlc++;
        if(lp->mlc >= EQLINE && lp->flag&EJECT)
            cl = FORM;
        lpoutput(lpr, cl);
        if(cl == FORM)
            lp->mlc = 0;
    }

case '\r':
    lp->ccc = 0;
    if(lp->flag&IND)
        lp->ccc = 8;
    return;

case 010:
    if(lp->ccc > 0)
        lp->ccc--;
    return;

case ' ':
    lp->ccc++;
    return;

default:
    if(lp->ccc < lp->mcc) {
        lpoutput(lpr, '\r');
        lp->mcc = 0;
    }
    if(lp->ccc < MAXCOL) {
        while(lp->ccc > lp->mcc) {
            lpoutput(lpr, ' ');
            lp->mcc++;
        }
        lpoutput(lpr, cl);
        lp->mcc++;
    }
    lp->ccc++;
}

}

lpstrrt(lpr)
register lpr;
{
    lp11[lpr].flag = &~LPBUSY;
    lpstart(lpr);
}

lpstart(lpr)
register lpr;
{
    register struct lpr *lp;
    register int c;
    int lpstrrt();

    lp = &lp11[lpr];
    if(lp->flag&LPBUSY)

```

```

        return;
    lp->flag |= LPBUSY;
    if (lp->tab[lp].addr->lpsr & ERROR) {
        timeout(lp->str, lp, LPDELAY);
        return;
    }
    while (lp->tab[lp].addr->lpsr & DONE && (c =getc(lp)) >= 0)
        lp->tab[lp].addr->lpbuf = c;

    lp->flag &= ~LPBUSY;
}

lpint(lpr)
register int lpr;
{
    register struct lpr *lp;

    lp = &lp1[lpr];
    lpstart(lp);
    if (lp->cc == LPLWAT || lp->cc == 0)
        wakeup(lp);
}

lpoutput(lpr, c)
register lpr;
{
    register struct lpr *lp;

    lp = &lp1[lpr];
    if (lp->cc >= LPHWAT)
        sleep(lp, LPPRI);
    putc(c, lp);
    spl4();
    lpstart(lp);
    spl0();
}

```

```
/*      @(#)malloc.c      2.3      */
```

```
#include "sys/param.h"
#include "sys/system.h"
```

```
/*
 * Allocate 'size' units from the given
 * map. Return the base of the allocated
 * space.
 * In a map, the addresses are increasing and the
 * list is terminated by a 0 size.
 * The core map unit is 64 bytes; the swap map unit
 * is 512 bytes.
 * Algorithm is first-fit.
 */
```

```
malloc(mp, size)
```

```
struct map *mp;
```

```
{
    register int a;
    register struct map *bp;

    for (bp=mp; bp->m_size; bp++) {
        if (bp->m_size >= size) {
            a = bp->m_addr;
            bp->m_addr += size;
            if ((bp->m_size -= size) == 0)
                do {
                    bp++;
                    (bp-1)->m_addr = bp->m_addr;
                } while ((bp-1)->m_size != bp->m_size);
            return(a);
        }
    }
    return(0);
}
```

```
/*
 * Free the previously allocated space aa
 * of size units into the specified map.
 * Sort aa into map and combine on
 * one or both ends if possible.
 */
```

```
mfree(mp, size, aa)
```

```
struct map *mp;
```

```
char *aa;
```

```
{
    register struct map *bp;
    register int t;
    register char *a;

    a = aa;
    if ((bp = mp) == coremap && runin) {
        runin = 0;
        wakeup(&runin);
    }
    /* Wake scheduler when freeing core */
}
```



```

for (; bp->m_addr<=a && bp->m_size!=0; bp++);
if (bp>mp && (bp-1)->m_addr+(bp-1)->m_size == a) {
    (bp-1)->m_size += size;
    if (a+size == bp->m_addr) {
        (bp-1)->m_size += bp->m_size;
        while (bp->m_size) {
            bp++;
            (bp-1)->m_addr = bp->m_addr;
            (bp-1)->m_size = bp->m_size;
        }
    }
} else {
    if (a+size == bp->m_addr && bp->m_size) {
        bp->m_addr -= size;
        bp->m_size += size;
    } else if (size) do {
        t = bp->m_addr;
        bp->m_addr = a;
        a = t;
        t = bp->m_size;
        bp->m_size = size;
        bp++;
    } while (size = t);
}
}

```

```

/*      @(#)mem.c      2.7.1.1 */

/*
 *      Memory special file
 *      minor device 0 is physical memory
 *      minor device 2 is EOF/NULL
 *      minor devices >= 8 reserved for MAUS
 *      RESTRICTION: a single read or write to this driver from the user
 *                   may not request more than 8128 bytes. This is a result of
 *                   a limitation in copyio.
 *      BUG: Reading minor device 0 will not return an EOF at end-of-file,
 *           but will return ENXIO instead.
 *      Both the above restriction and bug could be overcome at the
 *      cose of additional code.
 */

```

```

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

```

```

#ifdef MAKEMAUS

```

```

/*
 *      Definition of MAUS regions
 */

```

```

struct mausmap mausmap[] {
    0,      2,      /* System Bulletin Board */
    2,      128,    /* DAS / DTP Parameters */
    130,    128,    /* *** / DTP Tables */
    258,    128,    /* *** / DA11 QUEUE Area */
    -1

```

```

};
#endif

```

```

mmread(dev)

```

```

{
    register unsigned n;
    register long offset;
    unsigned cnt;
    long mmoff();

    dev = dev.d_minor;
    if (dev == 2)
        return;
    while((offset=mmoff(dev,&cnt)) >= 0 && cnt != 0) {
        n = (cnt > 8128 ? 8128 : cnt);
        if (copyio(offset, u.u_base, n, U_RUD)) {
            u.u_error = ENXIO;
            break;
        }
        u.u_offset += n;
        u.u_base += n;
        u.u_count -= n;
    }
}

```

```

    }
}

mmwrite(dev)
{
    register unsigned n;
    register long offset;
    long mmoft();
    unsigned cnt = ~0;

    dev = dev.d_minor;
    while((offset=mmoft(dev,&cnt)) >= 0 && cnt != 0) {
        n = (cnt > 8128 ? 8128 : cnt);
        if (dev != 2)
            if (copyio(offset, u.u_base, n, U_WUD)) {
                u.u_error = ENXIO;
                break;
            }
        u.u_offset += n;
        u.u_base += n;
        u.u_count -= n;
    }
    if(cnt == ~0 && dev > 2) u.u_error = ENXIO;
}

/*
 * Calculate offset and count.
 * Check for MAUS minor device.
 */
long
mmoft(dev, cnt):
register dev;
register unsigned *cnt;
{
    register int bn;
    extern nmausent;

    if (dev <= 2) {
        *cnt = u.u_count;
        return(u.u_offset);
    }
#ifdef MAKEMAUS
    dev -= 8;
    if (dev >= 0 && dev < nmausent) {
        bn = (mausmap[dev].bsize<<6) - u.u_offset;
        if (bn <= 0)
            return(-1L);
        *cnt = min(u.u_count, bn);
        return((((long)mauscore+mausmap[dev].boffset)<<6)+u.u_offset);
    }
#endif
    u.u_error = ENXIO;
    return(-1L);
}

```

```

/*      @(#)mx1.c      2.9      */

#include "sys/param.h"
#include "sys/systm.h"
#include "sys/dir.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/reg.h"
#include "sys/proc.h"
#include "sys/tty.h"
#include "sys/inode.h"
#include "sys/mx.h"
#include "sys/file.h"
#include "sys/conf.h"
#include "sys/confx.h"

/*
 * Multiplexor:  clist version
 *
 * installation:
 *      requires a line in cdevsw -
 *          mxopen, mxclose, mxread, mxwrite, mxioctl, nulldev, 0,
 *
 *      also requires a line in linesw -
 *          mcread, nulldev, mcwrite, nulldev, nulldev, nulldev,
 *          nulldev, nulldev,
 *
 *      The linesw entry for mpx should be the last one in the table.
 *      'nldisc' (number of line disciplines) should not include the
 *      mpx line.  This is to prevent mpx from being enabled by an ioctl.
 */
struct chan  chans[NCHANS];
struct schan schans[NPORTS];
struct group *groups[NGROUPS];
int  mpxline;
struct chan *xcp();
dev_t  mpxdev = -1;

char  mcdebugs[NDEBUGS];

/*
 * Allocate a channel, set c_index to index.
 */
struct chan *
challoc(index, isport)
{
    register s,i;
    register struct chan *cp;

    s = spl6();
    for(i=0; i<(((isport)?NPORTS:NCHANS); i++) {
        cp = (isport)? schans+i: chans+i;
    }
}

```

```

        if(cp->c_group == NULL) {
            cp->c_index = index;
            cp->c_pgrp = 0;
            cp->c_flags = 0;
            splx(s);
            return(cp);
        }
    }
    splx(s);
    return(NULL);
}

/*
 * Allocate a group table cell.
 */
gpalloc()
{
    register i;

    for (i=NGROUPS-1; i>=0; i--)
        if (groups[i]==NULL) {
            groups[i]++;
            return(i);
        }
    u.u_error = ENXIO;
    return(i);
}

/*
 * Add a channel to the group in
 * inode ip.
 */
struct chan *
addch(ip, isport)
struct inode *ip;
{
    register struct chan *cp;
    register struct group *gp;
    register i;

    plock(ip);
    gp = &ip->i_un.i_group;
    for(i=0; i<NINDEX; i++) {
        cp = (struct chan *)gp->g_chans[i];
        if (cp == NULL) {
            if ((cp=challoc(i, isport)) != NULL) {
                gp->g_chans[i] = cp;
                cp->c_group = gp;
            }
            break;
        }
    }
    cp = NULL;
}

```

```

    prele(ip);
    return(cp);
}

/*
 * Mpxchan system call.
 */

mpxchan()
{
    extern mxopen(), mcread(), uchar();
    struct inode *ip, *gip;
    struct tty *tp;
    struct file *fp, *chfp, *gfp;
    struct chan *cp;
    struct group *gp, *ngp;
    struct mx_args vec;
    struct a {
        int cmd;
        int *argvec;
    } astr;
    struct a *uap = &astr;
    dev_t dev;
    int *npgrp, nppgrp;
    register int i;

    /*
     * Common setup code.
     */

    uap->cmd = u.u_arg[0];
    uap->argvec = u.u_arg[1];
    copyin(uap->argvec, &vec, sizeof vec);
    gp = gfp = cp = NULL;

    switch(uap->cmd) {
    case NPGRP:
        if (vec.m_arg[1] < 0)
            break;
    case CHAN:
    case JOIN:
    case EXTR:
    case ATTACH:
    case DETACH:
    case CSIG:
        gfp = getf(vec.m_arg[1]);
        if (gfp==NULL)
            return;
        gip = gfp->f_inode;
        gp = &gip->i_un.i_group;
        if (gp->g_inode != gip) {
            u.u_error = ENXIO;
            return;
        }
    }
}

```

```

switch(uap->cmd) {
/*
 * Create an MPX file.
 */
case MPX:
case MPXN:
    if (mpxdev < 0) {
        for (i=0; linesw[il.l_open; i++) {
            if (linesw[il.l_read==mcread) {
                mpxline = i;
                for (i=0; cdevsw[il.d_open; i++) {
                    if (cdevsw[il.d_open==mxopen) {
                        mpxdev = (dev_t)(i<<8);
                    }
                }
            }
        }
        if (mpxdev < 0) {
            u.u_error = ENXIO;
            return;
        }
    }
    if (uap->cmd==MPXN) {
        if ((ip=ialloc(rootdev))==NULL)
            return;
        ip->i_mode = ((vec.m_arg[0]&0777)+IFMPC) & ~u.u_mask;
        ip->i_flag = IACC|IUPD|ICHG;
    } else {
        u.u_dirp = vec.m_name;
        ip = namei(u.u_dirp, 1);
        if (ip != NULL) {
            u.u_error = EEXIST;
            iput(ip);
            return;
        }
        if (u.u_error)
            return;
        ip = maknode((vec.m_arg[0]&0777)+IFMPC);
        if (ip == NULL)
            return;
    }
    if ((i=gpallocc()) < 0) {
        iput(ip);
        return;
    }
    if ((fp=fallocc()) == NULL) {
        iput(ip);
        groups[il] = NULL;
        return;
    }
    ip->i_un.i_rdev = (daddr_t)(mpxdev+i);
    ip->i_count++;
    prele(ip);

```

```

gp = &ip->i_un.i_group;
groups[i] = gp;
gp->g_inode = ip;
gp->g_state = INUSE|ISGRP;
gp->g_group = NULL;
gp->g_file = fp;
gp->g_index = 0;
gp->g_rotmask = 1;
gp->g_rot = 0;
gp->g_datq = 0;
for(i=0; i<NINDEX; )
    gp->g_chans[i++] = NULL;

fp->f_flag = FREAD|FWRITE|FMP;
fp->f_inode = ip;
fp->f_un.f_chan = NULL;
return;

```

```

/*
 * join file descriptor (arg 0) to group (arg 1)
 * return channel number
 */

```

```

case JOIN:
    if ((fp=getf(vec.m_arg[0]))==NULL)
        return;
    ip = fp->f_inode;
    switch (ip->i_mode & IFMT) {
    case IFMPC:
        if ((fp->f_flag&FMP) != FMP) {
            u.u_error = ENXIO;
            return;
        }
        ngp = &ip->i_un.i_group;
        if (mtree(ngp, gp) == NULL)
            return;
        fp->f_count++;
        u.u_ar0[R0] = cpx(ngp);
        return;
    case IFCHR:
        dev = (dev_t)ip->i_un.i_rdev;
        tp = cdevsw[major(dev)].d_ttys;
        if (tp==NULL) {
            u.u_error = ENXIO;
            return;
        }
        tp = &tp[minor(dev)];
        if (tp->t_chan) {
            u.u_error = ENXIO;
            return;
        }
        if ((cp=addch(gip, 1))==NULL) {
            u.u_error = ENXIO;

```



```

        return;
    }
    tp->t_chan = cp;
    cp->c_fy = fp;
    fp->f_count++;
    cp->c_ttyp = tp;
    cp->c_line = tp->t_line;
    cp->c_flags = XGRP+PORT;
    u.u_ar0[R0] = cpx(cp);
    return;

default:
    u.u_error = ENXIO;
    return;
}

/*
 * Attach channel (arg 0) to group (arg 1).
 */

case ATTACH:
    cp = xcp(gp, vec.m_arg[0]);
    if (cp==NULL || cp->c_flags&ISGRP) {
        u.u_error = ENXIO;
        return;
    }
    u.u_ar0[R0] = cpx(cp);
    wakeup((caddr_t)cp);
    return;

case DETACH:
    cp = xcp(gp, vec.m_arg[0]);
    if (cp==NULL) {
        u.u_error = ENXIO;
        return;
    }
    detach(cp);
    return;

/*
 * Extract channel (arg 0) from group (arg 1).
 */

case EXTR:
    cp = xcp(gp, vec.m_arg[0]);
    if (cp==NULL) {
        u.u_error = ENXIO;
        return;
    }
    if (cp->c_flags & ISGRP) {
        mxfallloc(((struct group *)cp)->g_file);
        return;
    }
    if ((fp=cp->c_fy) != NULL) {
        mxfallloc(fp);
    }

```

```

        return;
    }
    if ((fp=falloc()) == NULL)
        return;
    fp->f_inode = gip;
    gip->i_count++;
    fp->f_un.f_chan = cp;
    fp->f_flag = (vec.m_arg[2]) ?
        (FREAD|FWRITE|FMPY) : (FREAD|FWRITE|FMPX);
    cp->c_fy = fp;
    return;

/*
 * Make new chan on group (arg 1).
 */
case CHAN:
    if ((gfp->f_flag&FMP)==FMP) cp = addch(gip, 0);
    if (cp == NULL) {
        u.u_error = ENXIO;
        return;
    }
    cp->c_flags = XGRP;
    cp->c_fy = NULL;
    cp->c_ttyp = cp->c_ottyp = (struct tty *)cp;
    cp->c_line = cp->c_oline = mppline;
    u.u_ar0[R0] = cpx(cp);
    return;

/*
 * Connect fd (arg 0) to channel fd (arg 1).
 * (arg 2 < 0) => fd to chan only
 * (arg 2 > 0) => chan to fd only
 * (arg 2 == 0) => both directions
 */
case CONNECT:
    if ((fp=getf(vec.m_arg[0]))==NULL)
        return;
    if ((chfp=getf(vec.m_arg[1]))==NULL)
        return;
    ip = fp->f_inode;
    i = ip->i_mode&IFMT;
    if (i!=IFCHR) {
        u.u_error = ENXIO;
        return;
    }
    dev = (dev_t)ip->i_un.i_rdev;
    tp = cdevsw[major(dev)].d_ttys;
    if (tp==NULL) {
        u.u_error = ENXIO;
        return;
    }
    tp = &tp[minor(dev)];
    if ((chfp->f_flag&FMPY)) {
        u.u_error = ENXIO;
    }

```

```

        return;
    }
    cp = chfp->f_un.f_chan;
    if (cp==NULL || cp->c_flags&PORT) {
        u.u_error = ENXIO;
        return;
    }
    i = vec.m_arg[2];
    if (i>=0) {
        cp->c_ottyp = tp;
        cp->c_oline = tp->t_line;
    }
    if (i<=0) {
        tp->t_chan = cp;
        cp->c_ttyp = tp;
        cp->c_line = tp->t_line;
    }
    u.u_ar0[R0] = 0;
    return;

case NPGRP:
    if (gp != NULL) {
        cp = xcp(gp, vec.m_arg[0]);
        if (cp==NULL) {
            u.u_error = ENXIO;
            return;
        }
        npgrp = &cp->c_pgrp;
    }
    else
        npgrp = &u.u_procp->p_pgrp;

    if ((nppgrp = vec.m_arg[2]) < 0) {
        u.u_error = ENXIO;
        return;
    }

    u.u_ar0[R0] = *npgrp =
        nppgrp ? nppgrp : u.u_procp->p_pid;
    return;

case CSIG:
    cp = xcp(gp, vec.m_arg[0]);
    if (cp==NULL || (i = vec.m_arg[2]) <= 0 || i > NSIG) {
        u.u_error = ENXIO;
        return;
    }
    signal(cp->c_pgrp, i);
    return;

case DEBUG:
    i = vec.m_arg[0];
    if (i<0 || i>NDEBUGS)
        return;
    mcdebugs[i] = vec.m_arg[1];

```

```

        if (i==ALL)
            for(i=0;i<NDEBUGS;i++)
                mcdebugs[i] = vec.m_arg[i];
        return;

    default:
        u.u_error = ENXIO;
        return;
}

detach(cp)
register struct chan *cp;
{
    register struct group *master,*sub;
    register index;

    if (cp->c_flags&ISGRP) {
        sub = (struct group *)cp;
        master = sub->g_group; index = sub->g_index;
        closef(sub->g_file);
        master->g_chans[index] = NULL;
        return;
    } else if (cp->c_flags&PORT && cp->c_ttyp != NULL) {
        closef(cp->c_fy);
        chdrain(cp);
        chfree(cp);
        return;
    }
    if (cp->c_fy && (cp->c_flags&WCLOSE)==0) {
        cp->c_flags |= WCLOSE;
        chwake(cp);
    } else {
        chdrain(cp);
        chfree(cp);
    }
}

mxfallloc(fp)
register struct file *fp;
{
    register i;

    if (fp==NULL) {
        u.u_error = ENXIO;
        return(-1);
    }
    i = ufalloc(0);
    if (i < 0)
        return(i);
    u.u_ofile[i] = fp;
    fp->f_count++;
    u.u_ar0[R0] = i;
    return(i);
}

```

```
/*
 * Grow a branch on a tree.
 */
```

```
mtree(sub, master)
register struct group *sub, *master;
{
    register i;
    int mtresiz, stresiz;

    if ((mtresiz=mup(master, sub)) == NULL) {
        u.u_error = ENXIO;
        return(NULL);
    }
    if ((stresiz=mdown(sub, master)) <= 0) {
        u.u_error = ENXIO;
        return(NULL);
    }
    if (sub->g_group != NULL) {
        u.u_error = ENXIO;
        return(NULL);
    }
    if (stresiz+mtresiz > NLEVELS) {
        u.u_error = ENXIO;
        return(NULL);
    }
    for (i=0; i<NINDEX; i++) {
        if (master->g_chans[i] != NULL)
            continue;
        master->g_chans[i] = (struct chan *)sub;
        sub->g_group = master;
        sub->g_index = i;
        return(1);
    }
    u.u_error = ENXIO;
    return(NULL);
}
```

```
mup(master, sub)
struct group *master, *sub;
{
    register struct group *top;
    register int depth;

    depth = 1; top = master;
    while (top->g_group) {
        depth++;
        top = top->g_group;
    }
    if (top == sub)
        return(NULL);
    return(depth);
}
```

```
mdown(sub, master)
struct group *sub, *master;
{
    register int maxdepth, i, depth;

    if(sub == (struct group *)NULL || (sub->g_state&ISGRP) == 0)
        return(0);
    if(sub == master)
        return(-1);
    maxdepth = 0;
    for(i=0; i<NINDEX; i++) {
        if((depth=mdown(sub->g_chans[i], master)) == -1)
            return(-1);
        maxdepth = (depth>maxdepth) ? depth: maxdepth;
    }
    return(maxdepth+1);
}
```

```

/*      @(#)mx2.c      2.10      */

#include "sys/param.h"
#include "sys/systm.h"
#include "sys/dir.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/reg.h"
#include "sys/proc.h"
#include "sys/tty.h"
#include "sys/inode.h"
#include "sys/inodex.h"
#include "sys/mx.h"
#include "sys/file.h"
#include "sys/filex.h"
#include "sys/conf.h"
#include "sys/confx.h"
#include "sys/buf.h"
#include "sys/seg.h"

/*
 * multiplexor driver
 */
struct chan      chans[NCHANS];
struct group      *groups[NGROUPS];
int      mpxline;

short      cmask[16]      = {
        01,      02,      04,
        010,      020,      040,
        0100,      0200,      0400,
        01000,      02000,      04000,
        010000,      020000,      040000,      0100000
};

struct chan *xcp(), *addch(), *nextcp();

#define HIQ      32
#define LOQ      20
#define MIN(a,b)      ((a<b)?a:b)
#define FP      ((struct file *)cp)

char mcdebugs[NDEBUGS];

struct group *
getmpx(dev)
dev_t dev;
{
    register d;

    d = minor(dev);
    if (d >= NGROUPS || groups[d] == NULL) {
        u.u_error = ENXIO;
        return(NULL);
    }
}

```

```
    return(groups[id]);
}
```

```
mxopen(dev, flag)
```

```
{
    register struct group *gp;
    register struct file *fp;
    register struct chan *cp;
    int      msg;

    if (dev == NODEV)      /* power fail restart */
        return;
    if ((gp=getmpx(dev)) == NULL) {
        return;
    }
    if (!(gp->g_state&INUSE)) {
        u.u_error = ENXIO;
        return;
    }
    fp = u.u_ofile[u.u_ar0[R0]];
    if (fp->f_inode != gp->g_inode) {
        u.u_error = ENXIO;
        return;
    }
    if ((cp=addch(gp->g_inode,0)) == NULL) {
        u.u_error = ENXIO;
        return;
    }

    cp->c_flags = XGRP;
    cp->c_ottyp = cp->c_ttyp = (struct tty *)cp;
    cp->c_line = cp->c_oline = mppline;

    fp->f_flag |= FMPY;
    fp->f_flag |= FREAD+FWRITE;
    fp->f_un.f_chan = cp;

    if (gp->g_inode == mpxip) {
        plock(mpxip);
        mpiname(cp);
        msg = M_OPEN;
    } else
        msg = M_WATCH;

    scontrol(cp, msg+(cp->c_index<<8), u.u_uid);
    sleep((caddr_t)cp, PZERO);
    if (cp->c_flags&NMBUF)
        prele(mpxip);
    if (cp->c_flags & WCLOSE) {
        chdrain(cp);
        chfree(cp);
        u.u_error = ENXIO;
        return;
    }
    cp->c_fy = fp;
}
```



```
cp->c_pgrp = u.u_procp->p_pgrp;
```

```
char    mxnmdbuf[NMSIZE];
int     nmsize;
struct  chan *mxnmcp;
```

```
mpxname(cp)
register struct chan *cp;
{
    register char *np;
    register c;

    np = mxnmdbuf;
    u.u_dirp = (caddr_t)u.u_arg[0];

    while (np < &mxnmdbuf[NMSIZE]) {
        c = uchar();
        if (c <= 0)
            break;
        *np++ = c;
    }
    *np++ = '\0';
    nmsize = np - mxnmdbuf;
    cp->c_flags |= NMBUF;
}
```

```
mxclose(dev, flag, cp)
dev_t    dev;
register struct chan *cp;
{
    register struct group *gp;
    register struct inode *ip;
    register struct file *fp;
    int    i, fmp;

    fmp = flag&FMP;

    /*
     * close a channel
     */
    if (cp!=NULL && fmp && fmp!=FMP) {
        for(fp=file; fp<&file[NFILE]; fp++)
            if (fp->f_count && fp->f_flag&FMP && fp->f_un.f_chan==cp){
                return;
            }
        chdrain(cp);
        if ((cp->c_flags&WCLOSE)==0) {
            scontrol(cp, M_CLOSE, 0);
            cp->c_flags |= WCLOSE;
        } else {
            chfree(cp);
        }
        return;
    }
}
```

```

    }

    if ((gp=getmpx(dev)) == NULL)
        return;

    ip = gp->g_inode;
    if (ip==NULL || (ip->i_mode&IFMT)!=IFMPC) {
        return;
    }

    for(fp=file; fp < &file[NFILE]; fp++) {
        if (fp->f_count && (fp->f_flag&FMP)==FMP && fp->f_inode==ip) {
            return;
        }
    }

    if (ip == mpxip) {
        mpxip = NULL;
        prele(ip);
    }

    for(i=0; i<NINDEX; i++)
        if ((cp=gp->g_chans[i])!=NULL)
            detach(cp);

    groups[minor(dev)] = NULL;
    plock(ip);

    /*
     * The following prevents someone doing an open from getting
     * attached to the wrong process because he (the opener) didn't
     * know that this guy went away without unlinking his mpx file.
     */

    i = ip->i_mode;
    i &= ~IFMT;
    i |= IFCHR;
    ip->i_mode = i;

    zero(gp, sizeof (struct group));
    ip->i_flag |= IUPD|ICHG;
    iput(ip);
}

zero(s, cc)
register char *s;
register cc;
{
    while (cc--)
        *s++ = 0;
}

char m_eot[] = { M_EOT, 0, 0, 0 };

/*
 * Mxread + mxwrite are entered from cdevsw

```

```

* for all read/write calls. Operations on
* an mpx file are handled here.
* Calls are made through linesw to handle actual
* data movement.
*/

```

```

mxread(dev)

```

```

{
    register struct group *gp;
    register struct chan *cp;
    register esc;
    struct rh h;
    caddr_t base;
    unsigned count;
    int s, xfr, more, fmp;

    if ((FP=getf(u.u_ar0[R01])) == NULL) {
        return;
    }
    fmp = FP->f_flag & FMP;
    if (fmp != FMP) {
        msread(fmp, FP->f_un.f_chan);
        return;
    }
    if ((gp=getmpx(dev)) == NULL) {
        return;
    }
    if (((int)u.u_base & 1) {
        u.u_error = ENXIO;
        return;
    }

    s = spl6();
    while (gp->g_datq == 0) {
        sleep((caddr_t)&gp->g_datq, TTIPRI);
    }
    splx(s);

    while (gp->g_datq && u.u_count >= CNTLSIZ + 2) {
        esc = 0;
        cp = nextcp(gp);
        if (cp == NULL) {
            continue;
        }
        h.index = cpx(cp);
        if (count = cp->c_ctlx.c_cc) {
            count += CNTLSIZ;
            if (cp->c_flags & NMBUF)
                count += mmsize;
            if (count > u.u_count) {
                sdata(cp);
                return;
            }
            esc++;
        }
        base = u.u_base;
        count = u.u_count;
    }
}

```

```

    u.u_base += sizeof h;
    u.u_count -= sizeof h;
    xfr = u.u_count;
    if (esc) {
        more = mcread(cp);
    } else {
        more = (*linesw[cp->c_line].l_read)(cp->c_ttyp);
    }
    if (more > 0)
        sdata(cp);
    if (more < 0)
        scontrol(cp, M_CLOSE, 0);
    if (xfr == u.u_count) {
        esc++;
        lmove(m_eot, sizeof m_eot, B_READ);
    }
    xfr -= u.u_count;
    if (esc) {
        h.count = 0;
        h.ccount = xfr;
    } else {
        h.count = xfr;
        h.ccount = 0;
        mxrstrt(cp, &cp->cx.datq, BLOCK|ALT);
    }
    if (u.u_count && (xfr&1)) {
        u.u_base++;
        u.u_count--;
    }
    copyout(&h, base, sizeof h);
}

```

```

mxwrite(dev)
{
    register struct chan *cp;
    struct wh h; struct tty *tp;
    struct group *gp;
    int ucount, esc, fmp, burpcount, line;
    caddr_t ubase, hbase, waddr;

    if ((FP=getf(u.u_ar0[R0])) == NULL) {
        return;
    }
    fmp = FP->f_flag & FMP;
    if (fmp != FMP) {
        mswrite(fmp, FP->f_un.f_chan);
        return;
    }
    if ((gp=getmpx(dev)) == NULL) {
        return;
    }
    burpcount = 0;
    while (u.u_count >= sizeof h) {
        hbase = u.u_base;

```

```

    iomove(&h, sizeof h, B_WRITE);
    if (u.u_error)
        return;
    esc = 0;
    if (h.count==0) {
        esc++;
        h.count = h.ccount;
    }
    cp = xcp(gp, h.index);
    if (cp==NULL || cp->c_flags&ISGRP) {
        u.u_error = ENXIO;
        return;
    }
    ucount = u.u_count;
    ubase = u.u_base;
    u.u_count = h.count;
    u.u_base = h.data;

    if (esc==0) {
        if (cp->c_flags&PORT) {
            line = cp->c_line;
            tp = cp->c_ttyp;
        } else {
            line = cp->c_oline;
            tp = cp->c_ottyp;
        }
loop:
        waddr = (caddr_t)(*linesw[line].l_write)(tp);
        if (u.u_count) {
            if (gp->g_state&ENAMSG) {
                burpcount++;
                cp->c_flags |= BLKMSG;
                h.ccount = -1;
                h.count = u.u_count;
                h.data = u.u_base;
                copyout(&h, hbase, sizeof h);
            } else {
                if (waddr == 0) {
                    u.u_error = ENXIO;
                    return;
                }
                sleep(waddr, TTOPRI);
                goto loop;
            }
        }
    } else
        mxwcontrol(cp);

    u.u_count = ucount;
    u.u_base = ubase;
}
u.u_count = burpcount;

```

```

/*
 * Mcread and mcwrite move data on an mpx file.
 * Transfer addr and length is controlled by mxread/mxwrite.
 * Kernel-to-Kernel and other special transfers are not
 * yet in.
 */
mcread(cp)
register struct chan *cp;
{
register struct clist *q;
register char *np;

int cc;

    q = (cp->c_ctlx.c_cc) ? &cp->c_ctlx : &cp->cx.datq;
    cc = mxmove(q, B_READ);

    if (cp->c_flags&NMBUF && q == &cp->c_ctlx) {
        np = mxnmbuf;
        while (nmsize-->0)
            passc(*np++);
        cp->c_flags &= ~NMBUF;
        prele(mpxip);
    }
    if (cp->c_flags&PORT)
        return(cp->c_ctlx.c_cc + cp->c_ttyp->t_rawq.c_cc); else
        return(cp->c_ctlx.c_cc + cp->cx.datq.c_cc);
}

mcwrite(cp)
register struct chan *cp;
{
register struct clist *q;
register cc;
int s;

    q = &cp->cy.datq;
    while (u.u_count) {
        s = spl6();
        if (q->c_cc > HIQ || (cp->c_flags&EOTMARK)) {
            cp->c_flags |= SIGBLK;
            splx(s);
            break;
        }
        splx(s);
        cc = mxmove(q, B_WRITE);
    }
out:
    wakeup((caddr_t)q);
    return((caddr_t)q);
}

/*

```

```

* Msread and mswrite move bytes
* between user and non-multiplexed channel.
*/

```

```

msread(fmp, cp)
register struct chan *cp;
{
    register struct clist *q;
    register cc;
    int s;

    q = (fmp&FMPX) ? &cp->cx.datq : &cp->cy.datq;
    s = spl6();
    while (q->c_cc == 0) {
        if (cp->c_flags & EOTMARK) {
            cp->c_flags &= ~EOTMARK;
            if (msgenab(cp))
                scontrol(cp, M_UBLK, 0);
            else {
                wakeup((caddr_t)cp);
                wakeup((caddr_t)q);
            }
            goto out;
        }
        if (cp->c_flags & WCLOSE) {
            u.u_error = ENXIO;
            goto out;
        }
        sleep((caddr_t)q, TTIPRI);
    }
    splx(s);
    while (mxmove(q, B_READ) > 0)
        ;
    mxrstrrt(cp, q, SIGBLK);
    return;
out;
    splx(s);
}

```

```

mswrite(fmp, cp)
register struct chan *cp;
{
    register struct clist *q;
    register unsigned int cc;

    q = (fmp&FMPX) ? &cp->cy.datq : &cp->cx.datq;
    while (u.u_count) {
        spl6();
        if (cp->c_flags & WCLOSE) {
            u.u_error = EPIPE;
            psignal(u.u_procp, SIGPIPE);
            spl0();
            return;
        }
        if (q->c_cc >= HIO) {
            sdata(cp);

```

```

        cp->c_flags |= BLOCK;
        sleep((caddr_t)q+1, TTOPRI);
        spl0();
        continue;
    }
    spl0();
    cc = mxmove(q, B_WRITE);
    if (cc < 0)
        break;
}
if (fmp&FMPX) {
    if (cp->c_flags&YGRP) sdata(cp);
    else wakeup((caddr_t)q);
} else {
    if (cp->c_flags&XGRP) sdata(cp);
    else wakeup((caddr_t)q);
}
}

```

```

/*
 * move chars between clist and user space.
 */

```

```

mxmove(q, dir)
register struct clist *q;
register dir;
{
    register cc;
    char buf[HIO];

    cc = MIN(u.u_count, HIO);
    if (dir == B_READ)
        cc = q_to_b(q, buf, cc);
    if (cc <= 0)
        return(cc);
    iomove(buf, cc, dir);
    if (dir == B_WRITE)
        cc = b_to_q(buf, cc, q);
    return(cc);
}

```

```

mxrstrt(cp, q, b)
register struct chan *cp;
register struct clist *q;
register b;
{
    int s;

    s = spl6();
    if (cp->c_flags&b && q->c_cc<LOQ) {
        cp->c_flags &= ~b;
        if (b&ALT)
            wakeup((caddr_t)q+1); else

```



```

        mcstart(cp, q);
    }
    if (cp->c_flags & WFLUSH)
        wakeup((caddr_t)q+2);
    splx(s);
}

/*
 * called from driver start or xint routines
 * to wakeup output sleeper.
 */
mcstart(cp, q)
register struct chan *cp;
register caddr_t q;
{
    if (cp->c_flags & (BLKMSG)) {
        cp->c_flags &= ~BLKMSG;
        scontrol(cp, M_UBLK, 0);
    } else
        wakeup(q);
}

mxwcontrol(cp)
register struct chan *cp;
{
    short  cmd[2];
    int    s;

    bmove(cmd, sizeof cmd, B_WRITE);
    switch(cmd[0]) {
/*
        Not ready to queue this up yet.
*/
        case M_EOT:
            s = spl6();
            while (cp->c_flags & EOTMARK)
                if (msgenab(cp)) {
                    scontrol(cp, M_BLK, 0);
                    goto out;
                } else
                    sleep(cp, TTOPRI);
            cp->c_flags |= EOTMARK;

        out:
            wakeup(&cp->cy.datq);
            splx(s);
            break;

        case M_IOCTL:
            printf("M_IOCTL\n");
            break;

        default:
            u.u_error = ENXIO;
    }
}

```

}

```

mxioctl(dev, cmd, addr, flag)
caddr_t addr;
{
    struct group *gp;
    int fmp;
    struct file *fp;

    if ((gp=getmpx(dev))!=NULL || (fp=getf(u.u_arg[0]))!=NULL) {
        return;
    }

    fmp = fp->f_flag & FMP;
    if (fmp == FMP) {
        switch(cmd) {

            case MXNBLK:
                gp->g_state |= ENAMSG;
                break;

            case MXLSTN:
                if (mpxip == NULL) {
                    mpxip = gp->g_inode;
                    break;
                }

            default:
                u.u_error = ENXIO;
        }
    }
}

```

```

chdrain(cp)
register struct chan *cp;
{
    register struct tty *tp;
    int wflag;

    chwake(cp);

    tp = cp->c_ttyp;
    if (tp == NULL) /* prob not required */
        return;
    if (cp->c_flags&PORT && tp->t_chan == cp) {
        cp->c_ttyp = NULL;
        tp->t_chan = NULL;
        return;
    }
    wflag = (cp->c_flags&WCLOSE)==0;
    if (wflag)
        wflush(cp, &cp->cx.datq);
    else

```

```

        flush(&cp->cx.datq);
    if (!((cp->c_flags&YGRP)) {
        flush(&cp->cy.datq);
    }
}

chwake(cp)
register struct chan *cp;
{
    register char *p;

    wakeup(cp);
    flush(&cp->c_ctlx);
    p = (char *)&cp->cx.datq;
    wakeup(p); wakeup(++p); wakeup(++p);
    p = (char *)&cp->cy.datq;
    wakeup(p); wakeup(++p); wakeup(++p);
}

```

```

chfree(cp)
register struct chan *cp;
{
    register struct group *gp;
    register i;

    gp = cp->c_group;
    if (gp==NULL)
        return;
    i = cp->c_index;
    if (cp == gp->g_chans[i])
        gp->g_chans[i] = NULL;
    cp->c_group = NULL;
}

```

```

flush(q)
register struct clist *q;
{
    while(q->c_cc)
        getc(q);
}

```

```

wflush(cp,q)
register struct chan *cp;
register struct clist *q;
{
    register s;

    s = spl6();
    while(q->c_cc) {
        if (cp->c_flags & WCLOSE) {
            flush(q);
            break;
        }
    }
}

```

```

    }
    cp->c_flags |= WFLUSH;
    sdata(cp);
    sleep((caddr_t)q+2, TTOPRI);
}
cp->c_flags &= ~WFLUSH;
splx(s);
}

```

```

scontrol(cp, event, value)
register struct chan *cp;
short event, value;
{
    register struct clist *q;
    int s;

```

```

    q = &cp->c_ctlx;
    s = spl6();
    if (sdata(cp) == NULL)
        return;
    putw(event, q);
    putw(value, q);
    splx(s);
}

```

```

sdata(gp)
register struct group *gp;
{
    register struct group *ngp;
    register int s;

    ngp = gp->g_group;
    if (ngp == NULL || (ngp->g_state & ISGRP) == 0)
        return(NULL);

    s = spl6();
    do {
        ngp->g_datq |= cmask(gp->g_index);
        wakeup((caddr_t)&ngp->g_datq);
        gp = ngp;
    } while (ngp != gp->g_group);
    splx(s);
    return((int)gp);
}

```

```

struct chan *
xcp(gp, x)
register struct group *gp;
register short x;
{
    register int i;

    if ((x & 017) >= NINDEX)

```

```

        return((struct chan *)NULL);
while (gp->g_group) gp=gp->g_group;
for (i=0; i<NLEVELS; i++) {
    if ((x&017) >= NINDEX)
        break;
    if (gp==NULL || (gp->g_state&ISGRP)==0)
        return((struct chan *)NULL);
    gp = (struct group *)gp->g_chans[x&017];
    x >>= 4;
}
return((struct chan *)gp);
}

cpx(cp)
register struct chan *cp;
{
    register x;
    register struct group *gp;

    x = (-1<<4) + cp->c_index;
    gp = cp->c_group;
    while (gp->g_group) {
        x <<= 4;
        x |= gp->g_index;
        gp = gp->g_group;
    }
    return(x);
}

struct chan *
nextcp(gp)
register struct group *gp;
{
    register struct group *lgp, *ngp;

    do {
        while ((gp->g_datq & cmask[gp->g_rot]) == 0) {
            gp->g_rot = (gp->g_rot+1)%NINDEX;
        }
        lgp = gp;
        gp = (struct group *)gp->g_chans[gp->g_rot];
    } while (gp!=NULL && gp->g_state&ISGRP);

    lgp->g_datq &= ~cmask[lgp->g_rot];
    lgp->g_rot = (lgp->g_rot+1)%NINDEX;

    while (ngp=lgp->g_group) {
        ngp->g_datq &= ~cmask[lgp->g_index];
        if (ngp->g_datq)
            break;
        lgp = ngp;
    }
    return((struct chan *)gp);
}

```

```
msgenab(cp)
register struct chan *cp;
{
    register struct group *gp;

    for(gp=cp->c_group; gp->g_state & ISGRP; gp=gp->g_group)
        if(gp->g_state & ENAMSG) return(1);
    return(0);
}
```

```
/*
 * b_to_q: return number of bytes not transferred
 */
```

```
b_to_q(buf, count, q)
register char *buf;
register count;
register struct clist *q;
{
    if (count <= 0)
        return(0);
    do {
        if (putc(*buf++, q) == -1)
            return(count);
    } while (--count);
    return(0);
}
```

```
/*
 * q_to_b: return number of bytes transferred
 */
```

```
q_to_b(q, buf, count)
register char *buf;
register count;
register struct clist *q;
{
    int c, ocnt;

    if (count <= 0)
        return(0);
    ocnt = count;
    do {
        if ((c = getc(q)) == -1)
            return(ocnt-count);
        else
            *buf++ = c;
    } while (--count);
    return(ocnt-count);
}
```

```
iomove(adr, cnt, flg)
{
    register paddr_t padr;
```

```
padr = (unsigned)KDSA->rf(adr)>>13)&07f;  
padr <= 6;  
padr += adr&017777;  
pimove(padr, cnt, flg);  
}
```

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

```
#
```

```
/*  
 * Named pipe interface  
 */
```

```
#include "sys/param.h"  
#include "sys/file.h"  
#include "sys/filex.h"  
#include "sys/conf.h"  
#include "sys/reg.h"  
#include "sys/user.h"  
#include "sys/userx.h"  
#include "sys/ioctl.h"
```

```
#ifndef NNAMPIPE  
#define NNAMPIPE 10  
#endif
```

```
struct nmpipe {  
    struct file *p_rfp;  
    struct file *p_wfp;  
} nmpipe[NNAMPIPE];
```

```
npopen(dev, flag)
```

```
{  
    register struct nmpipe *npp;  
    extern struct file *getf();  
    register i;
```

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

```
#endif
```

```
    if (dev.d_minor >= NNAMPIPE) {  
        u.u_error = ENXIO;  
        return;
```

```
    }
```

```
    npp = &nmpipe[dev.d_minor];
```

```
    while (npp->p_rfp == 1)  
        sleep((caddr_t)npp, -1);
```

```
    if (npp->p_rfp == 0) {  
        npp->p_rfp = 1;  
        i = u.u_ar0[R0];  
        pipe();  
        wakeup((caddr_t)npp);  
        if (u.u_error) {  
            npp->p_rfp = 0;  
            return;
```

```
        }  
        npp->p_rfp = getf(u.u_ar0[R0]);  
        (npp->p_wfp = getf(u.u_ar0[R1]))->f_flag |= FNPIPE;
```



```

        u.u_ofile[u.u_ar0[R0]] = 0;
        u.u_ofile[u.u_ar0[R1]] = 0;
        u.u_ar0[R0] = i;
    }
    getf(u.u_ar0[R0])->f_flag = FPIPE;
}

```

npclose(dev, flag)

```

{
    register struct nmpipe *npp;
    register struct file *rfp, *wfp;

    npp = &nmpipe[dev.d_minor];

    rfp = npp->p_rfp;
    npp->p_rfp = 0;
    wfp = npp->p_wfp;
    npp->p_wfp = 0;
    closef(rfp);
    closef(wfp);
}

```

npread(dev)

```

{
    register struct nmpipe *npp;

    npp = &nmpipe[dev.d_minor];
    readp(npp->p_rfp);
}

```

npwrite(dev)

```

{
    register struct nmpipe *npp;

    npp = &nmpipe[dev.d_minor];
    writep(npp->p_wfp);
}

```

```

/*
 * This routine is actually used to pass back information to the
 * fstat system call as well as being used by the user to condition
 * the reading and writing end of a named pipe to sleep or
 * not to sleep when the pipe is empty or full. For the user ioctl
 * this routine is called from ioctl. For the stat info it is called
 * from fstat.
 */

```

npioctl(dev, cmd, addr, flag)

```

caddr_t addr;
{
    register struct nmpipe *npp;
    register *ip;
    struct pipcb pipcb;

    npp = &nmpipe[dev.d_minor];
    switch (cmd) {
    case OLDSGTTY:

```

```

        if (addr == 0) { /* stty */
            if(u.u_arg[1] != 0376) {
                u.u_error = EINVAL;
                return;
            }
            pipcb.pip_rflg = u.u_arg[0];
            pipcb.pip_wflg = u.u_arg[2];
            goto set;
        } else { /* gtty */
            if(addr[1] != 0376) {
                u.u_error = EINVAL;
                return;
            }
            addr[0] = (npp->p_rfp->f_flag & FNPIPE) ? 0 : 1;
            addr[2] = (npp->p_wfp->f_flag & FNPIPE) ? 0 : 1;
        }
        return;

case FIOSPIPE:
    if (copyin(addr, (caddr_t)&pipcb, sizeof(pipcb))) {
        u.u_error = EFAULT;
        return;
    }

set:
    if (pipcb.pip_rflg)
        npp->p_rfp->f_flag &= ~FNPIPE;
    else
        npp->p_rfp->f_flag |= FNPIPE;
    if (pipcb.pip_wflg)
        npp->p_wfp->f_flag &= ~FNPIPE;
    else
        npp->p_wfp->f_flag |= FNPIPE;
    ip = npp->p_rfp->f_inode;
    wakeup((caddr_t)ip+1);
    wakeup((caddr_t)ip+2);
    return;

case FIOGPIPE:
    pipcb.pip_rflg = (npp->p_rfp->f_flag & FNPIPE) ? 0 : 1;
    pipcb.pip_wflg = (npp->p_wfp->f_flag & FNPIPE) ? 0 : 1;
    if (copyout((caddr_t)&pipcb, addr, sizeof(pipcb))) {
        u.u_error = EFAULT;
        return;
    }

case GETRFP:
    return(npp->p_rfp);

case GETWFP:
    return(npp->p_wfp);

default:
    u.u_error = EINVAL;
    break;
}

```

```
/*      @(#)partab.c      2.3      */
```

```
/*  
 *      Copyright 1973 Bell Telephone Laboratories Inc  
 */
```

```
char partab[] {  
    0001,0201,0201,0001,0201,0001,0001,0201,  
    0202,0004,0003,0205,0005,0206,0201,0001,  
    0201,0001,0001,0201,0001,0201,0201,0001,  
    0001,0201,0201,0001,0201,0001,0001,0201,  
    0200,0000,0000,0200,0000,0200,0200,0000,  
    0000,0200,0200,0000,0200,0000,0000,0200,  
    0000,0200,0200,0000,0200,0000,0000,0200,  
    0200,0000,0000,0200,0000,0200,0200,0000,  
    0200,0000,0000,0200,0000,0200,0200,0000,  
    0000,0200,0200,0000,0200,0000,0000,0200,  
    0000,0200,0200,0000,0200,0000,0000,0200,  
    0000,0200,0200,0000,0200,0000,0000,0200,  
    0200,0000,0000,0200,0000,0200,0200,0000,  
    0200,0000,0000,0200,0000,0200,0200,0000,  
    0000,0200,0200,0000,0200,0000,0000,0201  
};
```

```
/*  
 * Character delay table--number of clock ticks required for a character  
 * time at a given speed. Indexed by tp->t_speed&017.  
 */
```

```
char chrdelay[16] {  
    0, 7, 6, 6, 5, 5, 4, 3, 2, 1, 1, 1, 1, 1, 1, 6  
};
```

```
/*      @(#)pipe.c      2.4      */  
#  
/*  
*      Copyright 1973 Bell Telephone Laboratories Inc  
*/
```

```
#include "sys/param.h"  
#include "sys/systm.h"  
#include "sys/user.h"  
#include "sys/userx.h"  
#include "sys/inode.h"  
#include "sys/inodex.h"  
#include "sys/file.h"  
#include "sys/filex.h"  
#include "sys/reg.h"
```

```
/*  
* Max allowable buffering per pipe  
* is specified by PIPSI2 in param.h.  
* This is also the max size of the  
* file created to implement the pipe.  
* If this size is bigger than 4096,  
* pipes will be implemented in LARG  
* files, which is probably not good.  
*/
```

```
/*  
* The sys-pipe entry.  
* Allocate an inode on the root device.  
* Allocate 2 file structures.  
* Put it all together with flags.  
*/
```

pipe()

```
{  
    register struct inode *ip;  
    register struct file *rf, *wf;  
    int r;  
  
    ip = ialloc(rootdev);  
    if(ip == NULL)  
        return;  
    rf = falloc();  
    if(rf == NULL) {  
        iput(ip);  
        return;  
    }  
    r = u.u_ar0[R0];  
    wf = falloc();  
    if(wf == NULL) {  
        rf->f_count = 0;  
        u.u_ofile[r] = NULL;  
        iput(ip);  
        return;  
    }  
}
```

```

    u.u_ar0[R1] = u.u_ar0[R0];
    u.u_ar0[R0] = r;
    wf->f_flag = FWRITE|FPIPE;
    wf->f_inode = ip;
    rf->f_flag = FREAD|FPIPE;
    rf->f_inode = ip;
    ip->i_count = 2;
    ip->i_flag = IACC|IUPD|ICHG;
    ip->i_mode = IFREG;
}

/*
 * Read call directed to a pipe.
 */
readp(fp)
register struct file *fp;
{
    register struct inode *ip;

    ip = fp->f_inode;

loop:
    /*
     * Very conservative locking.
     */

    plock(ip);
    /*
     * If nothing in pipe, wait.
     */
    if(ip->i_size1 == 0) {
        /*
         * If there are not both reader and
         * writer active, return without
         * satisfying read.
         * Also if the Named pipe bit is set return immediately.
         * Note that this bit may or may not be set on
         * a named pipe. See the named pipe s/gtty routine.
         */

        prele(ip);
        if(ip->i_count < 2 || (fp->f_flag & FNPIPE))
            return;
        ip->i_mode |= IREAD;
        sleep((caddr_t)ip+2, PPIPE);
        goto loop;
    }

    /*
     * Read and return
     */

    u.u_offset = fp->f_un.f_offset;
    readi(ip);
    fp->f_un.f_offset = u.u_offset;
    /*

```

```

    * If the head (read) has caught up with
    * the tail (write), reset both to 0.
    */

    if(fp->f_un.f_offset == ip->i_size) {
        fp->f_un.f_offset = 0;
        itrunc(ip);
        if(ip->i_mode&IWRITE) {
            ip->i_mode = & ~IWRITE;
            wakeup((caddr_t)ip+1);
        }
    }
    prele(ip);
}

/*
 * Write call directed to a pipe.
 */
writep(fp)
register struct file *fp;
{
    register c;
    register struct inode *ip;

    ip = fp->f_inode;
    c = u.u_count;

loop:

    /*
     * If all done, return.
     */

    plock(ip);
    if(c == 0 ||
        (fp->f_flag&FNPIPE && c>PIPSIZ-ip->i_size)) {
        prele(ip);
        u.u_count = c;
        return;
    }

    /*
     * If there are not both read and
     * write sides of the pipe active,
     * return error and signal too.
     */

    if(ip->i_count < 2) {
        prele(ip);
        u.u_error = EPIPE;
        psignal(u.u_procp, SIGPIPE);
        return;
    }

    /*
     * If the pipe is full,

```

```

    * wait for reads to deplete
    * and truncate it.
    */

    if(ip->i_size1 >= PIPSI2) {
        ip->i_mode |= IWRITE;
        prele(ip);
        sleep((caddr_t)ip+1, PPIPE);
        goto loop;
    }

    /*
     * Write what is possible and
     * loop back.
     * If writing less than PIPSI2, it always goes.
     * One can therefore get a file > PIPSI2 if write
     * sizes do not divide PIPSI2.
     */

    u.u_offset = ip->i_size1;
    u.u_count = min(c, PIPSI2);
    c -= u.u_count;
    writei(ip);
    prele(ip);
    if(ip->i_mode & IREAD) {
        ip->i_mode &= ~IREAD;
        wakeup((caddr_t)ip+2);
    }
    if (u.u_error == 0)
        goto loop;
    return;
}

/*
 * Lock a pipe.
 * If its already locked,
 * set the WANT bit and sleep.
 */
plock(ip)
register struct inode *ip;
{
    while(ip->i_flag & ILOCK) {
        ip->i_flag |= IWANT;
        sleep((caddr_t)ip, -3);
    }
    ip->i_flag |= ILOCK;
}

/*
 * Unlock a pipe.
 * If WANT bit is on,
 * wakeup.
 * This routine is also used
 * to unlock inodes in general.
 */

```

```
prele(ip)
register struct inode *ip;
{
    ip->i_flag &= ~ILOCK;
    if(ip->i_flag&IWANT) {
        ip->i_flag &= ~IWANT;
        wakeup((caddr_t)ip);
    }
}
```



```

/*      @(#)rf.c      2.5.1.1 */
#
/*
 * RF disk driver
 */

#include "sys/param.h"
#include "sys/system.h"
#include "sys/buf.h"
#include "sys/bufx.h"
#include "sys/conf.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/proc.h"
#include "sys/elog.h"
#include "sys/iobuf.h"

struct device {
    int     rfcs;
    int     rfwc;
    int     rfba;
    int     rfda;
    int     rfdae;
};

#define NRF      1
#define NRFBLK   1024
#define RFADDR   0177460

struct iostat rfstat[NRF];
struct iobuf  rftab  tabinit(RF0,&rfstat);

#define GO      01
#define RCOM    02
#define WCOM    04
#define CTLCLR  0400
#define IENABLE 0100
#define WLO     02000

/*
 * Monitoring device number
 */
#define DK_N     0

rfopen(dev, flag)
{
#ifdef PWR_FAIL
    extern unsigned pwr_fail;

    if (dev == NODEV) {
        if (flag) {
            rftab.b_active = 0;
            if (pwr_fail == NULL)

```

```

        rfstart();
    }
    return;
}

#endif

if(dev.d_minor >= NRF)
    u.u_error = ENXIO;
rftab.io_addr = RFADDR;
rftab.io_nreg = NDEVREG;
}

rfstrategy(bp)
register struct buf *bp;
{
    register struct buf *p1, *p2;

    if((bp->b_flags&B_MAP) == 0)
        mapalloc(bp);
    if (bp->b_blkno >= (daddr_t)NREBLK*(bp->b_dev.d_minor+1)) {
        if (bp->b_flags&B_READ)
            bp->b_resid = bp->b_bcount;
        else {
            bp->b_flags |= B_ERROR;
            bp->b_error = ENXIO;
        }
        iodone(bp);
        return;
    }
    bp->b_pri = u.u_procp->p_nice;
    spl5();
    if ((p1 = rftab.b_actf) == 0) {
        rftab.b_actf = bp;
        bp->av_forw = 0;
    } else {
        for (; p2 = p1->av_forw; p1 = p2)
            if (p2->b_pri > bp->b_pri)
                break;
        bp->av_forw = p2;
        p1->av_forw = bp;
        while (p2) {
            if (p2->b_pri > bp->b_pri)
                p2->b_pri--;
            p2 = p2->av_forw;
        }
    }
    if (rftab.b_active==0)
        rfstart();
    spl0();
}

rfstart()
{
    register struct buf *bp;

    if ((bp = rftab.b_actf) == 0)

```

```

        return;
    rftab.b_active++;
    rfstat[(bp->b_blkno.hibyte>>2)&071.io_ops++;
    blkacty = 1 (1<<RF0);
    RFADDR->rfae = bp->b_blkno.hibyte;
    devstart(bp, &RFADDR->rfa, bp->b_blkno<<8, 0);
    dk_busy = 1 (1<<DK_N;
    dk_numb[DK_N] += 1;
    dk_wds[DK_N] += (bp->b_bcount>>6) & 03777;
}

rfintr()
{
    register struct buf *bp;
    register status;
    struct device rfregs[0];

    if (rftab.b_active == 0) {
        logstray(RFADDR);
        return;
    }
    blkacty = & ~(1<<RF0);
    dk_busy = & ~(1<<DK_N);
    bp = rftab.b_actf;
    rftab.b_active = 0;
    if (RFADDR->rfa < 0) { /* error bit */
        status = RFADDR->rfa;
        rftab.io_stp = &rfstat[(bp->b_blkno.hibyte>>2)&071];
        fmtberr(&rftab, 0);
        RFADDR->rfa = CTLCLR;
        if (++rftab.b_errcnt < 10 && (status&WLO) == 0) {
            rfstart();
            return;
        }
        bp->b_flags |= B_ERROR;
    }
    if (rftab.io_erec)
        logberr(&rftab, bp->b_flags&B_ERROR);
    rftab.b_errcnt = 0;
    rftab.b_actf = bp->av_forw;
    bp->b_resid = (-RFADDR->rfa)<<1;
    iodone(bp);
    rfstart();
}

rfread(dev)
{
    physio(rfstategy, dev, B_READ, NRFBLK*(dev.d_minor+1));
}

rfwrite(dev)
{
    physio(rfstategy, dev, B_WRITE, NRFBLK*(dev.d_minor+1));
}

```

```
/*      G( # ) rh.c      2.5.1.1 */
```

```
#
/*
*/
```

```
#include "sys/param.h"
#include "sys/buf.h"
#include "sys/bufx.h"
#include "sys/conf.h"
#include "sys/sysm.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/proc.h"
#include "sys/procx.h"
#include "sys/seq.h"
```

```
/*
 * startup routine for RH controllers.
 */
```

```
#define IENABLE 0100
#define RHWCOM 060
#define RHRCOM 070
#define GO 01
```

```
rhstart(bp, devloc, devblk, abae)
register struct buf *bp;
int *devloc, *abae;
```

```
{
    register int *dp;
    register int com;
```

```
#ifdef PWR_FAIL
    extern pwr_fail;
#endif
```

```
    dp = devloc;
    if (cputype == 70)
        *abae = bp->b_paddr >> 16;
    *dp = devblk; /* block address */
    *--dp = bp->b_paddr; /* buffer address */
    *--dp = -(bp->b_bcount >> 1); /* word count */
    com = IENABLE | GO |
        ((bp->b_paddr >> 8) & 001400);
    if (bp->b_flags & B_READ) /* command + x-mem */
        com = RHRCOM;
    else
        com = RHWCOM;
```

```
#ifdef PWR_FAIL
    *--dp = com ^ pwr_fail;
```

```
#endif
```

```
#ifndef PWR_FAIL
    *--dp = com;
```

```
#endif
}
```

```

/*
 * 11/70 routine to allocate the
 * UNIBUS map and initialize for
 * a unibus device.
 * The code here and in
 * rhstart assumes that an rh on an 11/70
 * is an rh70 and contains 22 bit addressing.
 */
int mapwant;

mapalloc(bp)
register struct buf *bp;
{
    register i, j;
    long dble;
    int regno;

    if(cputype != 70 || bp->b_bcount == 0)
        return;
    j = (bp->b_bcount-1)/8192+1;
    spl6();
    while((regno = malloc(ubmap, j)) == 0) {
        mapwant++;
        sleep(ubmap, PSWP);
    }
    spl0();
    dble = bp->b_paddr;
    j = 2*(regno+j);
    for(i = regno*2; i < j; i += 2) {
        UBMAP->r[i] = dble.loword;
        UBMAP->r[i+1] = dble.hiword;
        dble += 8192;
    }
    bp->b_paddr = ((long)regno)<<13;
    bp->b_flags |= B_MAP;
}

mapfree(bp)
register struct buf *bp;
{
    register regno;

    bp->b_flags &= ~B_MAP;
    regno = bp->b_paddr>>13;
    bp->b_paddr.loword = UBMAP->r[regno*2];
    bp->b_paddr.hiword = UBMAP->r[regno*2+1];
    mfree(ubmap, (bp->b_bcount-1)/8192+1, regno);
    if(mapwant) {
        wakeup(ubmap);
        mapwant = 0;
    }
}

```

```
/*      @(#)rhf.c      2.3      */
```

```
/*  
 * Fake rh code for 11/40's  
 */
```

```
mapalloc()  
{  
}
```

```
mapfree()  
{  
}
```

```
/*      @(#)rk.c      2.5.1.1 */

#
/*
 * RK disk driver
 */

#include "sys/param.h"
#include "sys/systm.h"
#include "sys/buf.h"
#include "sys/bufx.h"
#include "sys/conf.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/proc.h"
#include "sys/elog.h"
#include "sys/iobuf.h"

#define NRK      4
#define NRKBLK  4872
#define RKADDR  0177400

#define RESET   0
#define GO      01
#define DRESET  014
#define IENABLE 0100
#define DRY     0200
#define ARDY    0100
#define WLO     020000
#define CTLRDY  0200

/*
 * Monitoring device bit
 */
#define DK_N     1

struct device {
    int rkds;
    int rker;
    int rkcs;
    int rkwc;
    int rkba;
    int rkda;
};

struct iostat rkstat[NRK];
struct iobuf  rktab  tabinit(RK0,&rkstat);

rkopen(dev, flag)
{
#ifdef PWR_FAIL
    extern unsigned pwr_fail;

    if (dev == NODEV) {
```

```

        if (flag) {
            rktab.b_active = 0;
            if (pwr_fail == NULL)
                rkstart();
        }
        return;
    }
#endif

    if(((dev.d_minor&07) >= NRK)
        u.u_error = ENXIO; /* even in interleaved, max: */
        rktab.io_addr = RKADDR; /* unit num must be < NRK */
        rktab.io_nreg = NDEVREG;
    }

rkstrategy(bp)
register struct buf *bp;
{
    register struct buf *p1, *p2;
    int d;

    if((bp->b_flags&B_MAP) == 0)
        mapalloc(bp);
    d = bp->b_dev.d_minor-7;
    if(d <= 0)
        d = 1;
    if (bp->b_blkno >= NRKBLK*d) {
        if (bp->b_flags&B_READ)
            bp->b_resid = bp->b_bcount;
        else {
            bp->b_flags |= B_ERROR;
            bp->b_error = ENXIO;
        }
        iodone(bp);
        return;
    }
    bp->b_pri = u.u_procp->p_nice;
    spl5();
    if ((p1 = rktab.b_actf) == 0) {
        rktab.b_actf = bp;
        bp->av_forw = 0;
    } else {
        for (; p2 = p1->av_forw; p1 = p2)
            if (p2->b_pri > bp->b_pri)
                break;
        bp->av_forw = p2;
        p1->av_forw = bp;
        while (p2) {
            if (p2->b_pri > bp->b_pri)
                p2->b_pri--;
            p2 = p2->av_forw;
        }
    }
    if (rktab.b_active==0)
        rkstart();
    spl0();
}

```



}

rkaddr(bp)

```

struct buf *bp;
{
    register int b, d, m;

    b = bp->b_blkno;
    m = bp->b_dev.d_minor - 7;
    if (m <= 0)
        d = bp->b_dev.d_minor;
    else {
        d = lrem(b, m);
        b = ldiv(b, m);
    }
    rktab.io_stp = &rkstat[dl];
    return(d<<13 | (b/12)<<4 | b*12);
}

```

rkstart()

```

{
    register struct buf *bp;
    register a;

    if ((bp = rktab.b_actf) == 0)
        return;
    rktab.b_active++;
    a = rkaddr(bp);
    rktab.io_stp->io_ops++;
    blkacty |= (1<<RK0);
    devstart(bp, &RKADDR->rkda, a, 0);
    dk_busy |= (1<<DK_N);
    dk_numb[DK_N] += 1;
    dk_wds[DK_N] += (bp->b_bcount>>6) & 03777;
}

```

rkintr()

```

{
    register struct buf *bp;
    struct device rkregs[0];
    register status;

    if (rktab.b_active == 0)
        return;
    blkacty ^= (1<<RK0);
    dk_busy ^= (1<<DK_N);
    bp = rktab.b_actf;
    rktab.b_active = 0;
    if (RKADDR->rkcs < 0) { /* error bit */
        status = RKADDR->rker;
        fmtberr(&rktab, 0);
        RKADDR->rkcs = RESET|GO;
        rktab.io_stp->io_misc++;
        while((RKADDR->rkcs&CTLRDY) == 0);
        if (++rktab.b_errcnt < 10 && (status&WLO) == 0) {
            rkstart();
        }
    }
}

```

```
        return;
    }
    bp->b_flags |= B_ERROR;
}
if (rktab.io_erec)
    logberr(&rktab, bp->b_flags & B_ERROR);
rktab.b_errcnt = 0;
rktab.b_actf = bp->av_forw;
bp->b_resid = (-RKADDR->rkwc)<<1;
iodone(bp);
rkstart();
}

rkread(dev)
{
    register nblks;

    nblks = dev.d_minor - 7;
    if (nblks <= 0)
        nblks = 1;

    physio(rkstrategy, dev, B_READ, NRKBLK*nblks);
}

rkwrite(dev)
{
    register nblks;

    nblks = dev.d_minor - 7;
    if (nblks <= 0)
        nblks = 1;

    physio(rkstrategy, dev, B_WRITE, NRKBLK*nblks);
}
```