

| Copyright 1982 UniSoft Corporation

| Interrupt vector dispatch table
 | One entry per interrupt vector location

```
.text
.globl _dispatc, _pmvect, _tevect
_dispatc:
bsr lfault | 0    Reset: Initial SSP
bsr lfault | 1    Reset: Initial PC
bsr lbuserr | 2    Bus Error
bsr laddrerr | 3    Address Error
bsr lfault | 4    Illegal Instruction
bsr lfault | 5    Zero Divide
bsr lfault | 6    CHK Instruction
bsr lfault | 7    TRAPV Instruction
bsr lfault | 8    Privilege Violation
bsr lfault | 9    Trace
bsr lfault | 10   Line 1010 Emulator
bsr lfault | 11   Line 1111 Emulator
bsr lfault | 12   (Unassigned, reserved)
bsr lfault | 13   (Unassigned, reserved)
bsr lfault | 14   (Unassigned, reserved)
bsr lfault | 15   (Unassigned, reserved)
bsr lfault | 16   (Unassigned, reserved)
bsr lfault | 17   (Unassigned, reserved)
bsr lfault | 18   (Unassigned, reserved)
bsr lfault | 19   (Unassigned, reserved)
bsr lfault | 20   (Unassigned, reserved)
bsr lfault | 21   (Unassigned, reserved)
bsr lfault | 22   (Unassigned, reserved)
bsr lfault | 23   (Unassigned, reserved)
bsr lfault | 24   Spurious Interrupt
bsr llintr | 25   Level 1 pp 0, sony, vert retrace (clock)
bsr kbintr | 26   Level 2 cops, mouse, rtclock, buttons
bsr pi5 | 27   Level 3 exp slot 3 (default 2-port card)
bsr pi4 | 28   Level 4 exp slot 2 (default 2-port card)
bsr pi3 | 29   Level 5 exp slot 1 (default 2-port card)
bsr scintr | 30   Level 6 scc chip
bsr nmi | 31   Level 7 Interrupt Autovector
bsr lsyscall | 32   System Call
bsr lfault | 33   TRAP Instruction Vector
bsr lfault | 34   TRAP Instruction Vector
bsr lfault | 35   TRAP Instruction Vector
bsr lfault | 36   TRAP Instruction Vector
bsr lfault | 37   TRAP Instruction Vector
bsr lfault | 38   TRAP Instruction Vector
bsr lfault | 39   TRAP Instruction Vector
bsr lfault | 40   TRAP Instruction Vector
bsr lfault | 41   TRAP Instruction Vector
bsr lfault | 42   TRAP Instruction Vector
bsr lfault | 43   TRAP Instruction Vector
bsr lfault | 44   TRAP Instruction Vector
bsr lfault | 45   TRAP Instruction Vector
bsr lfault | 46   TRAP Instruction Vector
bsr lfault | 47   TRAP Instruction Vector
bsr lfault | 48   (Unassigned, reserved)
bsr lfault | 49   (Unassigned, reserved)
bsr lfault | 50   (Unassigned, reserved)
bsr lfault | 51   (Unassigned, reserved)
bsr lfault | 52   (Unassigned, reserved)
bsr lfault | 53   (Unassigned, reserved)
bsr lfault | 54   (Unassigned, reserved)
bsr lfault | 55   (Unassigned, reserved)
bsr lfault | 56   (Unassigned, reserved)
bsr lfault | 57   (Unassigned, reserved)
bsr lfault | 58   (Unassigned, reserved)
bsr lfault | 59   (Unassigned, reserved)
bsr lfault | 60   (Unassigned, reserved)
bsr lfault | 61   (Unassigned, reserved)
bsr lfault | 62   (Unassigned, reserved)
bsr lfault | 63   (Unassigned, reserved)
bsr lfault | 64   (User Interrupt Vector)
bsr lfault | 65   (User Interrupt Vector)
bsr lfault | 66   (User Interrupt Vector)
```

```
bsr lfault | 67    (User Interrupt Vector)
bsr lfault | 68    (User Interrupt Vector)
bsr lfault | 69    (User Interrupt Vector)
bsr lfault | 70    (User Interrupt Vector)
bsr lfault | 71    (User Interrupt Vector)
bsr lfault | 72    (User Interrupt Vector)
bsr lfault | 73    (User Interrupt Vector)
bsr lfault | 74    (User Interrupt Vector)
bsr lfault | 75    (User Interrupt Vector)
bsr lfault | 76    (User Interrupt Vector)
bsr lfault | 77    (User Interrupt Vector)
bsr lfault | 78    (User Interrupt Vector)
bsr lfault | 79    (User Interrupt Vector)
bsr lfault | 80    (User Interrupt Vector)
bsr lfault | 81    (User Interrupt Vector)
bsr lfault | 82    (User Interrupt Vector)
bsr lfault | 83    (User Interrupt Vector)
bsr lfault | 84    (User Interrupt Vector)
bsr lfault | 85    (User Interrupt Vector)
bsr lfault | 86    (User Interrupt Vector)
bsr lfault | 87    (User Interrupt Vector)
bsr lfault | 88    (User Interrupt Vector)
bsr lfault | 89    (User Interrupt Vector)
bsr lfault | 90    (User Interrupt Vector)
bsr lfault | 91    (User Interrupt Vector)
bsr lfault | 92    (User Interrupt Vector)
bsr lfault | 93    (User Interrupt Vector)
bsr lfault | 94    (User Interrupt Vector)
```

```
| Values for expansion card interrupt vectors when Priam card present.
_pmvect:
bsr pmint3 | 27    Level 3 exp slot 3, Priam card
bsr pmint2 | 28    Level 4 exp slot 2, Priam card
bsr pmint1 | 29    Level 5 exp slot 1, Priam card
```

```
| Values for expansion card interrupt vectors when Techmar card present.
_tevect:
bsr teint3 | 27    Level 3 exp slot 3, Tecmar card
bsr teint2 | 28    Level 4 exp slot 2, Tecmar card
bsr teint1 | 29    Level 5 exp slot 1, Tecmar card
```

| Put actual "C" routine name onto the stack and call the system
 | interrupt dispatcher

```
.globl call, fault, buserr, addrerr, syscall, _idleflg
lfault: jmp    fault
lbuserr: jmp buserr
laddrerr: jmp addrerr
lsyscall: jmp syscall
```

```
| tty interrupt priority entry point
.globl _spltty
_spltty: movw sr, d0 | fetch current CPU priority
movw #0x2600, sr | set priority 6
rts
```

```
.globl _llintr
llintr: movw #0x2600, sr | go to spl6
movl #_llintr, sp@ | push call address
movw _idleflg, sp@- | clock needs old value so cheat
jmp call | jump to common interrupt handler
```

```
.globl _kbintr
kbintr: movl #_kbintr, sp@ | push call address
clrw sp@- | device number
jmp call | jump to common interrupt handler
```

```
.globl _ppintr
pi3: movl #_ppintr, sp@ | push call address
movw #1, sp@- | device number
jmp call | jump to common interrupt handler
```

```
pi4: movl #_ppintr, sp@ | push call address
movw #4, sp@- | device number
jmp call | jump to common interrupt handler
```

```
pi5:  movl  #_ppintr,sp@ | push call address
      movw  #7,sp@-    | device number
      jmp   call       | jump to common interrupt handler

      .globl _pmintr
pmintr3: movl  #_pmintr,sp@ | push call address
        movw  #2,sp@-    | device number
        jmp   call       | jump to common interrupt handler

pmintr2: movl  #_pmintr,sp@ | push call address
        movw  #1,sp@-    | device number
        jmp   call       | jump to common interrupt handler

pmintr1: movl  #_pmintr,sp@ | push call address
        clrw  sp@-      | device number
        jmp   call       | jump to common interrupt handler

      .globl _scintr
scintr:  movl  #_scintr,sp@ | push call address
        movw  #1,sp@-    | device number
        jmp   call       | jump to common interrupt handler

      .globl _nmikey
nmi:    movl  #_nmikey,sp@ | push call address
        movw  #0,sp@-    | device number
        jmp   call       | jump to common interrupt handler

      .globl _teintr
teintr3: movl  #_teintr,sp@ | push call address
        movw  #2,sp@-    | device number
        jmp   call       | jump to common interrupt handler

teintr2: movl  #_teintr,sp@ | push call address
        movw  #1,sp@-    | device number
        jmp   call       | jump to common interrupt handler

teintr1: movl  #_teintr,sp@ | push call address
        clrw  sp@-      | device number
        jmp   call       | jump to common interrupt handler
```

```

| USIZE dependencies
USIZE = 0x800          | Size of U area
PGSIZE = 512          | Size of a page
PAGESHIFT= 9          | Page shift

| Configuration dependencies
HIGH = 0x2700         | High priority supervisor mode (spl 7)
LOW = 0x2000          | Low priority, supervisor mode (spl 0)
USTART = 0x0          | Start of user program
PAGEBASE= 0xA00000    | Base page address
UDOT = 0xFA0000       | Logical start of U dot
UBASE = PAGEBASE+UDOT | U dot page map address

SETUP_0 = 0xFCE012    | Turn setup bit off
SETUP_1 = 0xFCE010    | Turn setup bit on
SEG1_0 = 0xFCE008     | Turn SEG1 bit off
SEG1_1 = 0xFCE00A     | Turn SEG1 bit on
SEG2_0 = 0xFCE00C     | Turn SEG2 bit off
SEG2_1 = 0xFCE00E     | Turn SEG2 bit on

        .globl _u, _segoff
        .data
_u = UDOT
cputype: .word 0          | Local copy of _cputype, 0 if 68000

        .text
        .globl start, _end, _edata, _main, _cputype, _dispatc

start:  movw  #HIGH, sr          | spl7
        movl  #_end+PGSIZE+USIZE-1, d7 | End of unix
        andl  #-PGSIZE, d7      | Round to nearest click
        movl  #_edata, a0       | Start clearing here
clrbss: movl  #0, a0@+          | Clear bss
        cmpl  d7, a0
        jcs  clrbss

| Determine cpu type
        movl  #1$, 16          | Illegal instruction vector
        .word 0x4E7A          | movc
        .word 0x0801          | vbr, d0
        movl  #68010, _cputype | No trap. Must be 68010
        movw  #1, cputype      | Local copy

i$:     subl  #USIZE, d7        | Start of U dot
        moveq #PAGESHIFT, d1    | Calculate U dot page entry
        lsrL  d1, d7           | Calculate U dot page entry
        movb #0, SEG1_0        | Set Context 0
        movb #0, SEG2_0        | Set Context 0
        movb #0, SETUP_1       | Enable MMU modification
        movw 0+0x8008, d1      | Read segment origin
        movb #0, SETUP_0       | Disable MMU modification
        andl  #0xFFF, d1       | Take just what is valid
        movw d1, _segoff       | Save segment offset
        addl  d1, d7           | Add segment offset
        movb #0, SETUP_1       | Enable MMU modification
        movw d7, UDOT+0x8008    | Segment origin
        movw #0x7FC, UDOT+0x8000 | Segment access and length (4 clicks)
        movb #0, SETUP_0       | Disable MMU modification
        subl  d1, d7           | Subtract off segment offset
        movl #UDOT+USIZE, sp    | Set stack at top of U area

        jsr  _bminit           | initialize raster display

        movl d7, sp@-          | Click address of udot to main
        jsr  _main             | Long jump to unix, init returns here

        clrL  sp@-             | Indicate short 4 byte stack format
        movl #USTART, sp@-     | Starting program address
        clrw  sp@-             | New sr value
        rte

| save and restore of register sets

        .globl _save, _resume, _gsave
_save:  movl  sp@+, a1          | return address
        movl  sp@, a0          | ptr to label_t

        moveml #0xFCFC, a0@    | save d2-d7, a2-a7
        movl  a1, a0@(48)      | save return address
        moveq #0, d0           |
        jmp   a1@              | return

_gsave: movl  sp@+, a1          | return address
        movl  sp@, a0          | ptr to label_t
        addw  #40, a0          |
        movl  a6, a0@+         | save a6
        movl  a7, a0@+         | save a7
        movl  a1, a0@+         | save return address
        moveq #0, d0           |
        jmp   a1@              | return

_resume: movl  sp@ (4), d0      | click address of new udot
        movl  sp@ (8), a0      | ptr to label_t
        movw  #HIGH, sr        | spl 7
        movb #0, SEG1_0        | Set Context 0
        movb #0, SEG2_0        | Set Context 0
        addw _segoff, d0       | Add segment offset
        movb #0, SETUP_1       | Enable MMU modification
        movw d0, UDOT+0x8008    | Segment origin
        movw #0x7FC, UDOT+0x8000 | Segment access and length (2k)
        movb #0, SETUP_0       | Disable MMU modification
        movb #0, SEG1_1        | Set Context 1
        movb #0, SEG2_1        | Set Context 1
        moveml a0@+, #0xFCFC  | restore the registers
        movw  #LOW, sr         | restore spl 0
        movl  a0@, a1          | fetch the original pc
        moveq #1, d0           | return 1
        jmp   a1@              | return

| spl commands
        .globl _splhi, _spl7, _spl6, _spl5, _spl4, _spl3, _spl2, _spl1, _spl0, _splx

_splhi:
_spl7:  movw  sr, d0            | fetch current CPU priority
        movw  #0x2700, sr      | set priority 7
        rts

_spl6:  movw  sr, d0            | fetch current CPU priority
        movw  #0x2600, sr      | set priority 6
        rts

_spl5:  movw  sr, d0            | fetch current CPU priority
        movw  #0x2500, sr      | set priority 5
        rts

_spl4:  movw  sr, d0            | fetch current CPU priority
        movw  #0x2400, sr      | set priority 4
        rts

_spl3:  movw  sr, d0            | fetch current CPU priority
        movw  #0x2300, sr      | set priority 3
        rts

_spl2:  movw  sr, d0            | fetch current CPU priority
        movw  #0x2200, sr      | set priority 2
        rts

_spl1:  movw  sr, d0            | fetch current CPU priority
        movw  #0x2100, sr      | set priority 1
        rts

_spl0:  movw  sr, d0            | fetch current CPU priority
        movw  #0x2000, sr      | set priority 0
        rts

_splx:  movw  sp@ (6), sr      | set priority
        rts

        .data
        .globl _idleflg
_idleflg: .word 0

        .text
        .globl _idle, _idlei, _waitloc
_idle:   movw  sr, d0            | Fetch current CPU priority
        movw  #1, _idleflg     | Set idle flag
        movw  #0x2000, sr      | Set priority zero
_idlei:  tstw  _idleflg         | Wait for interrupt
_waitloc:
        bne  _idle

```

```

movw d0,sr      | Restore priority
rts

.globl buserr, addrerr, fault, call, _busaddr
.globl _runrun, _trap

fault: clrw sp8-      | this makes ps long aligned
      clrw _idleflg  | clear idle flag
      moveml #0xFFFF,sp8- | save all registers
      movl usp,a0
      movl a0,sp8(60) | save usr stack ptr
      movl sp8(66),d0 | return ptr from the jsr
      subl #_dispatc+4,d0 | subtract dispatch table offset
      asrl #2,d0      | calculate vector number
resched:movl d0,sp8-  | argument to trap
      jsr _trap      | C handler for traps and faults
      addq1 #4,sp
      jsr checknet  | check for net int requests
      btst #5,sp8(70) | did we come from user mode?
      jne 2$        | no, just continue
      tstb _runrun  | should we reschedule?
      jeq 2$        | no, just return normally
      movl #256,d0  | 256 is reschedule trap number
      jra resched  | go back into trap

2$:  movl sp8(60),a0 | restore usr stack ptr
     movl a0,usp    | Set Contex 1
     movb #0,SEG1_1 | Set Contex 1
     movb #0,SEG2_0 | Set Contex 1
     moveml sp8+,#0x7FFF | restore all other registers
     addw #10,sp    | sp, pop fault pc, and alignment word
     rte

.globl syscall, _syscall
syscall:clrw sp8-      | this makes ps long aligned
      moveml #0xFFFF,sp8- | save all registers
      movl usp,a0
      movl a0,sp8(60) | save usr stack ptr
      btst #5,sp8(70) | did we come from user mode?
      jne 3$        | no, error !!!
      jsr _syscall   | Process system call
      jsr checknet  | check for net int requests
      tstb _runrun  | should we reschedule?
      jeq 2$        | no, just return normally
      movl #256,d0  | 256 is reschedule trap number
      jra resched  | go back into trap

3$:  movl sp8(60),a0 | restore user stack pointer
     movl a0,usp    | restore all other registers
     moveml sp8+,#0x7FFF | restore all other registers
     addw #10,sp    | sp, pop fault pc, and alignment word
     rte

3$:  moveml sp8+,#0x7FFF | restore registers
     addq1 #6,sp     | sp, pop fault pc, and alignment word
     movl #_dispatc+132,sp8 | simulate a bsr
     jra fault

| Bus error entry, this has its stack somewhat different. We will
| call a C routine to save the info then fix the stack to look like a trap.
| These entries will be called directly from interrupt vector.

buserr: tstw cputype  | test cpu type
      bne 3$        | branch if 68010
      moveml #0xC0C0,sp8- | save registers that C clobbers
      jsr _busaddr  | save the info for a bus or address error
      moveml sp8+,#0x303 | restore registers
      addq1 #8,sp   | pop bsr address, fcode, aaddr and ireg
35:  movl #_dispatc+12,sp8 | simulate a bsr
     bra fault

addrerr:tstw cputype  | test cpu type
      bne 4$        | branch if 68010
      moveml #0xC0C0,sp8- | save registers that C clobbers
      jsr _busaddr  | save the info for a bus or address error
      moveml sp8+,#0x303 | restore registers

```

```

      addq1 #8,sp   | pop bsr address, fcode, aaddr and ireg
4$:  movl #_dispatc+16,sp8 | simulate a bsr
     bra fault

| common interrupt dispatch

call: moveml #0xFFFF,sp8- | save all registers
      clrw _idleflg  | clear idle flag
      movl usp,a0
      movl a0,sp8(60) | save usr stack ptr
      movl sp8(66),a0 | fetch interrupt routine address
      movl sp,sp8-   | push argument list pointer onto stack
      jsr a0         | jump to actual interrupt handler
      addq1 #4,sp
      jsr checknet  | check for net int requests
      btst #5,sp8(70) | did we come from user mode?
      jne 2$        | no, just continue
      tstb _runrun  | should we reschedule?
      jeq 2$        | no, just return normally
      movl #256,d0  | 256 is reschedule trap number
      jra resched  | go back into trap

2$:  movl sp8(60),a0 | restore usr stack ptr
     movl a0,usp    | Set Contex 1
     movb #0,SEG1_1 | Set Contex 1
     movb #0,SEG2_0 | Set Contex 1
     moveml sp8+,#0x7FFF | restore all other registers
     addw #10,sp    | sp, pop fault pc, and alignment word
     rte

| General purpose code

.globl _tstb, _getusp, _getsr

_tstb: tstb sp8(0)   | stack probe instruction prototype

_getusp:movl usp,a0  | get the user stack pointer
      movl a0,d0
      rts

_getsr: moveq #0,d0  | get the sr
      movw sr,d0
      rts

| Net int request handler

.globl _netisr, _netintr, _svstak
.globl _chkstak
checknet:
      tstl _netisr  | net requesting soft interrupt?
      beq 3$        | no: return
      movw #0x2700,sr | set priority 7
      tstb innet   | already in net code ?
      bne 3$        | yes: return
      movb #1,innet | no: set flag -> in the net code
      movl sp,_svstak | save stack, get net stack
      movl #_netstak+2996,sp
      |movw #0x2600,sr | set priority 6
      movw #0x2000,sr | set priority 0
      jsr _netintr  | do tasks
      movl _svstak,sp
      clrb innet   | clear flag

3$:  rts

.data
innet: .byte 0

.text
.globl csl, _chkstak

| first stage of checkstack routine. save regs, call real routine
|csl:
| moveml #0xFFFF,sp8- | save all registers
| jsr _chkstak        | really check the stack
| moveml sp8+,#0x7FFF | restore all registers

```


mch.s

Fri Sep 5 19:08:47 1986

3

irts

```
#define FONTHORZ 8
#define FONTVERT 8
```

```
char bfont[] = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* space */
0x10, 0x10, 0x10, 0x10, 0x00, 0x00, 0x10, 0x00, /* ! */
0x48, 0x48, 0x48, 0x00, 0x00, 0x00, 0x00, 0x00, /* " */
0x48, 0x48, 0xFC, 0x48, 0xFC, 0x48, 0x48, 0x00, /* # */
0x10, 0x3C, 0x50, 0x38, 0x14, 0x78, 0x10, 0x00, /* $ */
0x00, 0xC4, 0xC8, 0x10, 0x20, 0x4C, 0x8C, 0x00, /* % */
0x60, 0x90, 0x90, 0x60, 0x94, 0x88, 0x74, 0x00, /* & */
0x08, 0x10, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, /* ' */
0x08, 0x10, 0x20, 0x20, 0x20, 0x10, 0x08, 0x00, /* ( */
0x40, 0x20, 0x10, 0x10, 0x10, 0x20, 0x40, 0x00, /* ) */
0x10, 0x54, 0x38, 0x7C, 0x38, 0x54, 0x10, 0x00, /* * */
0x00, 0x10, 0x10, 0x7C, 0x10, 0x10, 0x00, 0x00, /* + */
0x00, 0x00, 0x00, 0x00, 0x00, 0x30, 0x30, 0x60, /* , */
0x00, 0x00, 0x00, 0xFC, 0x00, 0x00, 0x00, 0x00, /* - */
0x00, 0x00, 0x00, 0x00, 0x30, 0x30, 0x00, 0x00, /* . */
0x00, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x00, /* / */
0x78, 0x84, 0x8C, 0xB4, 0xC4, 0x84, 0x78, 0x00, /* 0 */
0x10, 0x30, 0x50, 0x10, 0x10, 0x10, 0x7C, 0x00, /* 1 */
0x78, 0x84, 0x04, 0x18, 0x60, 0x80, 0xFC, 0x00, /* 2 */
0x78, 0x84, 0x04, 0x38, 0x04, 0x84, 0x78, 0x00, /* 3 */
0x08, 0x18, 0x28, 0x48, 0xFC, 0x08, 0x08, 0x00, /* 4 */
0xFC, 0x80, 0xF0, 0x08, 0x04, 0x88, 0x70, 0x00, /* 5 */
0x38, 0x40, 0x80, 0xF8, 0x84, 0x84, 0x78, 0x00, /* 6 */
0xFC, 0x84, 0x08, 0x10, 0x20, 0x20, 0x20, 0x00, /* 7 */
0x78, 0x84, 0x84, 0x78, 0x84, 0x84, 0x78, 0x00, /* 8 */
0x78, 0x84, 0x84, 0x7C, 0x04, 0x08, 0x70, 0x00, /* 9 */
0x00, 0x00, 0x30, 0x30, 0x00, 0x30, 0x30, 0x00, /* : */
0x00, 0x00, 0x30, 0x30, 0x00, 0x30, 0x30, 0x60, /* ; */
0x08, 0x10, 0x20, 0x40, 0x20, 0x10, 0x08, 0x00, /* < */
0x00, 0x00, 0xF8, 0x00, 0xF8, 0x00, 0x00, 0x00, /* = */
0x40, 0x20, 0x10, 0x08, 0x10, 0x20, 0x40, 0x00, /* > */
0x78, 0x84, 0x04, 0x18, 0x20, 0x00, 0x20, 0x00, /* ? */
0x38, 0x44, 0x94, 0xAC, 0x98, 0x40, 0x3C, 0x00, /* @ */
0x30, 0x48, 0x84, 0xFC, 0x84, 0x84, 0x84, 0x00, /* A */
0xF8, 0x44, 0x44, 0x78, 0x44, 0x44, 0xF8, 0x00, /* B */
0x78, 0x84, 0x80, 0x80, 0x80, 0x84, 0x78, 0x00, /* C */
0xF8, 0x44, 0x44, 0x44, 0x44, 0x44, 0xF8, 0x00, /* D */
0xFC, 0x80, 0x80, 0xF0, 0x80, 0x80, 0xFC, 0x00, /* E */
0xFC, 0x80, 0x80, 0xF0, 0x80, 0x80, 0x80, 0x00, /* F */
0x78, 0x84, 0x80, 0x9C, 0x84, 0x84, 0x78, 0x00, /* G */
0x84, 0x84, 0x84, 0xFC, 0x84, 0x84, 0x84, 0x00, /* H */
0x38, 0x10, 0x10, 0x10, 0x10, 0x10, 0x38, 0x00, /* I */
0x1C, 0x08, 0x08, 0x08, 0x08, 0x88, 0x70, 0x00, /* J */
0x84, 0x88, 0x90, 0xE0, 0x90, 0x88, 0x84, 0x00, /* K */
0x80, 0x80, 0x80, 0x80, 0x80, 0x80, 0xFC, 0x00, /* L */
0x84, 0xCC, 0xB4, 0xB4, 0x84, 0x84, 0x84, 0x00, /* M */
0x84, 0xC4, 0xA4, 0x94, 0x8C, 0x84, 0x84, 0x00, /* N */
0x78, 0x84, 0x84, 0x84, 0x84, 0x84, 0x78, 0x00, /* O */
0xF8, 0x84, 0x84, 0xF8, 0x80, 0x80, 0x80, 0x00, /* P */
0x78, 0x84, 0x84, 0x84, 0x94, 0x88, 0x74, 0x00, /* Q */
0xF8, 0x84, 0x84, 0xF8, 0x90, 0x88, 0x84, 0x00, /* R */
0x78, 0x84, 0x80, 0x78, 0x04, 0x84, 0x78, 0x00, /* S */
0x7C, 0x10, 0x10, 0x10, 0x10, 0x10, 0x00, /* T */
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x78, 0x00, /* U */
0x84, 0x84, 0x84, 0x48, 0x48, 0x30, 0x30, 0x00, /* V */
0x84, 0x84, 0x84, 0xB4, 0xB4, 0xCC, 0x84, 0x00, /* W */
0x84, 0x84, 0x48, 0x30, 0x48, 0x84, 0x84, 0x00, /* X */
0x44, 0x44, 0x44, 0x38, 0x10, 0x10, 0x10, 0x00, /* Y */
0xFC, 0x04, 0x08, 0x30, 0x40, 0x80, 0xFC, 0x00, /* Z */
0x78, 0x40, 0x40, 0x40, 0x40, 0x40, 0x78, 0x00, /* [ */
0x00, 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x00, /* \ */
0x78, 0x08, 0x08, 0x08, 0x08, 0x08, 0x78, 0x00, /* ] */
0x10, 0x28, 0x44, 0x00, 0x00, 0x00, 0x00, /* ^ */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* _ */
0x20, 0x10, 0x08, 0x00, 0x00, 0x00, 0x00, /* ` */
0x00, 0x00, 0x70, 0x08, 0x78, 0x88, 0x74, 0x00, /* a */
0x80, 0x80, 0xB8, 0xC4, 0x84, 0xC4, 0xB8, 0x00, /* b */
0x00, 0x00, 0x78, 0x80, 0x80, 0x80, 0x78, 0x00, /* c */
0x04, 0x04, 0x74, 0x8C, 0x84, 0x8C, 0x74, 0x00, /* d */
0x00, 0x00, 0x78, 0x84, 0xFC, 0x80, 0x78, 0x00, /* e */
0x18, 0x24, 0x20, 0xF8, 0x20, 0x20, 0x00, /* f */
0x00, 0x00, 0x74, 0x8C, 0x8C, 0x74, 0x04, 0x78, /* g */
```

```
0x80, 0x80, 0xB8, 0xC4, 0x84, 0x84, 0x84, 0x00, /* h */
0x10, 0x00, 0x30, 0x10, 0x10, 0x10, 0x38, 0x00, /* i */
0x08, 0x00, 0x18, 0x08, 0x08, 0x08, 0x70, /* j */
0x80, 0x80, 0x88, 0x90, 0xA0, 0xD0, 0x88, 0x00, /* k */
0x30, 0x10, 0x10, 0x10, 0x10, 0x10, 0x38, 0x00, /* l */
0x00, 0x00, 0xE8, 0x54, 0x54, 0x54, 0x54, 0x00, /* m */
0x00, 0x00, 0xF8, 0x44, 0x44, 0x44, 0x44, 0x00, /* n */
0x00, 0x00, 0x38, 0x44, 0x44, 0x44, 0x38, 0x00, /* o */
0x00, 0x00, 0xB8, 0xC4, 0xC4, 0xB8, 0x80, 0x80, /* p */
0x00, 0x00, 0x74, 0x8C, 0x8C, 0x74, 0x04, 0x04, /* q */
0x00, 0x00, 0xB8, 0xC4, 0x80, 0x80, 0x80, 0x00, /* r */
0x00, 0x00, 0x7C, 0x80, 0x78, 0x04, 0xF8, 0x00, /* s */
0x20, 0x20, 0xF8, 0x20, 0x20, 0x18, 0x00, /* t */
0x00, 0x00, 0x84, 0x84, 0x84, 0x8C, 0x74, 0x00, /* u */
0x00, 0x00, 0x84, 0x84, 0x84, 0x48, 0x30, 0x00, /* v */
0x00, 0x00, 0x44, 0x44, 0x54, 0x54, 0x6C, 0x00, /* w */
0x00, 0x00, 0x84, 0x48, 0x30, 0x48, 0x84, 0x00, /* x */
0x00, 0x00, 0x84, 0x84, 0x8C, 0x74, 0x04, 0x78, /* y */
0x00, 0x00, 0xFC, 0x08, 0x30, 0x40, 0xFC, 0x00, /* z */
0x38, 0x40, 0x40, 0xC0, 0x40, 0x40, 0x38, 0x00, /* { */
0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, /* | */
0xE0, 0x10, 0x10, 0x18, 0x10, 0x10, 0xE0, 0x00, /* } */
0x00, 0x62, 0xD6, 0x8C, 0x00, 0x00, 0x00, /* ~ */
0xAA, 0x54, 0xAA, 0x54, 0xAA, 0x54, 0xAA, /* . */
};
```

```
/*
 * Copyright 1982 UniSoft Corporation
 * Use of this material is subject to your disclosure agreement with
 * AT&T, Western Electric and UniSoft Corporation.
 *
 * Definitions relating to the COPS and Keyboard SY6522 interface.
 */

/* Breakdown of e_ifr (Interrupt Flag Register) of 6522 */
#define FCA2 0x01 /* ca2 -- handshake with COPS, not intr source */
#define FCA1 0x02 /* ca1 -- intr latch for input from COPS */
#define FSHFT 0x04 /* shft -- completed eight shifts */
#define FCB2 0x08 /* cb2 -- output to speaker (if suitably programmed)*/
#define FCB1 0x10 /* cb1 -- unused */
#define FTIMER2 0x20 /* timeout of timer two -- unused */
#define FTIMER1 0x40 /* timeout of timer one -- 10ms clock */
#define FIRQ 0x80 /* interrupt happened (cleared by clearing intr) */

struct device_e { /* Keyboard SY6522 VIA */
    char e_f0[1]; char e_irb; /* I/O register B */
    char e_f1[1]; char e_ira; /* I/O register A */
    char e_f2[1]; char e_ddrb; /* Data Dir reg B */
    char e_f3[1]; char e_ddra; /* Data Dir reg A */
    char e_f4[1]; char e_t1cl; /* T1 low Latches Counter */
    char e_f5[1]; char e_t1ch; /* T1 hi Latches Counter */
    char e_f6[1]; char e_t1ll; /* T1 low Latches */
    char e_f7[1]; char e_t1lh; /* T1 hi Latches */
    char e_f8[1]; char e_t2cl; /* T2 low Latches Counter */
    char e_f9[1]; char e_t2ch; /* T2 hi Latches Counter */
    char e_fa[1]; char e_sr; /* Shift Register */
    char e_fb[1]; char e_acr; /* Aux Ctrl Reg */
    char e_fc[1]; char e_pcr; /* Perif ctrl Reg */
    char e_fd[1]; char e_ifr; /* Int Flag Reg */
    char e_fe[1]; char e_ier; /* Int Ena Reg */
    char e_ff[1]; char e_aira; /* Alt e_ira (no handshake)*/
};
#define COPSADDR ((struct device_e *) (STDIO+0xDD80))

/* A port definitions */

/* Connects PR (reset pulse line) to the controller reset switch if low,
 * or the parity reset latch if high (when set as an output).
 */
#define CR 0x80
#define CRDY 0x40 /* goes low when cops ready for command */
#define PR 0x20 /* works with CR to get pport parity error */
#define FDIR 0x10 /* floppy dir interrupt request */
#define VC2 0x08
#define VC1 0x04
#define VC0 0x02
#define KBIN 0x01
```

```

/*
 * Corvus definitions
 *
 * (C) 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with Unisoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by Unisoft.
 */

/* block number to virtual drive */
#define cvbtovd(b)      (((b)>>16 & 0xf)+1)
/* virtual drive to block number */
#define cvvdtob(vd)    (((vd)-1) & 0xf) << 16)

/* cv_stat bits */
#define ST_BUSY        0x02
#define ST_HTOC        0x01      /* host to controller direction */

/* normal mode commands */
#define N_READ         0x02
#define N_WRITE        0x02
#define N_PARAM        0x10
#define N_DIAGN        0x11
#define N_R128         0x12
#define N_R256         0x22
#define N_R512         0x32
#define N_W128         0x13
#define N_W256         0x23
#define N_W512         0x33
#define N_BOOT         0x14

/* Sunol only - copy from one virtual drive to another
 * (such as tape to disk or disk to tape)
 */
#define N_COPY         0xE3

/* diagnostic mode commands */
#define D_RESET        0x00
#define D_FORMAT       0x01
#define D_VERIFY       0x07
#define D_RFIRMWARE    0x32
#define D_WFIRMWARE    0x33

/* semaphore commands */
#define S_INIT0        0x1a
#define S_INIT1        0x10
#define S_LOCK0        0x0b
#define S_LOCK1        0x01
#define S_ULOCK0       0x0b
#define S_ULOCK1       0x11
#define S_STAT0        0x1a
#define S_STAT1        0x41
#define S_STAT2        0x03

/* normal mode return status */
#define NS_FATAL        0x80      /* mask */
#define NS_VERIFY       0x40      /* mask */
#define NS_RECOVER      0x20      /* mask */
#define NS_HFAULT       0x00
#define NS_STIME        0x01
#define NS_SFAULT       0x02
#define NS_SERROR       0x03
#define NS_HCRC         0x04
#define NS_ZFAULT       0x05
#define NS_ZTIME        0x06
#define NS_OFFLINE      0x07
#define NS_WFAULT       0x08
#define NS_              0x09
#define NS_RFAULT       0x0a
#define NS_DATACRC      0x0b

```

```

/* Profile Controller Definitions
 * Unless otherwise noted this information comes from an undated
 * document titled 'PROFILE COMMUNICATIONS PROTOCOL' and is apparently
 * either version 3.96 or 3.98
 */
/* command codes */
#define PROREAD 0 /* read command */
#define PROWRITE 1 /* write command */
#define PROWRITEV 2 /* write/verify command */

/* States (responses read from profile) */
/* Kept in pd_state and pd_nextst to determine what to do on intr */
#define SERR 0 /* pseudo state for err and initialization */
#define SCMD 1 /* waiting for a command */
#define SRDBLK 2 /* ready to read a block */
#define SWRTD 3 /* ready to receive for write data */
#define SVERT 4 /* ready to receive for verify data */
#define SFINI 5 /* pseudo state for write status pickup */
#define SPERFORM 6 /* waiting to do actual write or verify */
#define SSTOP 7 /* pseudo state for idle */

/* Responses sent in reply to controller (messages) */
#define PGO 0x55 /* Proceed to next state (exec cmd) */
#define PIDL 0x00 /* Quit and return to idle loop */

/* Polling Delays */
#define RSPTIME 0xC000 /* response timeout (~1 ms) */

/* Operation status word breakdown. Returned for each command
 * and kept in pd_sbuf.
 */
/* Breakdown of status bits for Status Byte 1 */
#define FAIL 0x01 /* Operation unsuccessful */
#define TIMEOUT 0x04 /* Couldn't read header after 9 revs */
#define CRCERR 0x08 /* CRC error while trying to read/write */
#define SEEKERR 0x10 /* Unable in 3 tries to read 3 consec sec*/
#define NOTABLE 0x20 /* Data table not in RAM (spare updated)*/
#define ABORTED 0x40 /* >532 bytes sent or no spare table read*/
#define GOAHEAD 0x80 /* If the profile received 0x55 last time*/

/* Breakdown of status bits for Status Byte 2 */
#define BADSEEK 0x02 /* Seek to wrong track occurred */
#define SPARED 0x04 /* set if sparing occurred */
#define RDSTATS 0x08 /* Ctrl unable to read status sector */
#define BLKTABO 0x10 /* Bad block table overflow: > 100 bads */
#define SECTABO 0x40 /* Spared Sector table overflow >32 secs*/
#define SEEKER2 0x80 /* unable in 1 try to read 3 consec secs*/

/* Breakdown of status bits for Status Byte 3 */
#define PARITYE 0x01 /* Parity error */
#define BADRESP 0x02 /* If Ctrl gave a bad response */
#define WASREST 0x04 /* If Ctrl was reset */
#define BLKIDMM 0x20 /* Block id at end of sector mismatch */
#define BLKNINV 0x40 /* Block number invalid */
#define RESETP 0x80 /* Ctrl has been reset */

/* Status byte 4 is the # of errs rereading after read err*/

/* Mask to remove redundant bits from status word */
#define STATMSK 0x8000

/* Controller Command Format */
struct cmd {
    char p_cmd; /* command register */
    char p_high; /* high block byte */
    char p_mid; /* mid block byte */
    char p_low; /* low block byte */
    char p_retry; /* retry count */
    char p_thold; /* threshold count */
};

#define MAXBLOCK 19455 /* Last sector on disk */
#define SECSIZE 512 /* Number of bytes per sector */
#define LOG_SS 9 /* Log (base 2) (SECSIZE) */
#define NSEC 16 /* Number of sectors/track */
#define NRETRY 10 /* Number of checksum error retries */

```

```

extern struct device_d *pro_da[];

#define logical(x) (minor(x) & 7) /* eight logicals per phys */
#define interleave(x) (minor(x) & 0x8) /* interleave bit for swaping */
#define physical(x) ((minor(x) & 0xF0) >> 4) /* 10 physical devs */

#define ul_forw av_back

/* Since there may be up to ten devices active the driver locals have been
 * collected into this structure to allow easy swithing between them.
 */
struct prodata {
    struct device_d *pd_da; /* physical dev pointer */
    struct cmd pd_cmdb; /* command buffer */
    struct buf *pd_actv; /* ptr to active buf */
    daddr_t pd_blkno; /* present block being transferred */
    daddr_t pd_limit; /* max blk number for this log dev */
    caddr_t pd_addr; /* present core address */
    long pd_bcount; /* present count */
    long pd_sbuf; /* Status */
    long pd_flags; /* mode of operation flags */
    short pd_state; /* state of controller */
    short pd_nextst; /* next state scheduled for disk */
    short pd_unit; /* phys unit number of this dev */
    short pd_offline; /* wait for disk to come online again */
    char *pd_err; /* err mesg for last error */
} prodata[NPDEVES];

/* Breakdown of bits in pd_flags above: */
#define NOCHKSUM 0x01 /* disable checksums failure on reads */
#define NOPARITY 0x02 /* disable fail on parity errors */

extern char pro_secmap[];

#ifdef NODEBUG
#define DEBUG(x) printf x
#else
#define DEBUG(x)
#endif

```

```
short kb_keycount;
short kb_reptrap;
short kb_repwait;
short kb_repdlay;
```

```
char kb_chrbuf;
char kb_ctrl;
char kb_shft;
char kb_lock;
char kb_state;
char kb_idcode;
char kb_lastc;
```

```
/* state of cops processor */
```

```

/* Flags in conversion tables to pick out non-character generating
 * keys.
 */

/* When a character is received from the COPS (depending on the state of
 * the shift keys) a table is used to convert it into the ascii equiv.
 * If the result is one of the following then it wasn't a keycode at all
 * but one of the reset/mouse/clock/plug codes.
 */
#define Imp 0xFF /* invalid keycode */
#define KSC 0xFE /* mouse or reset state flag */
#define D1P 0xFD /* disk inserted into drive one */
#define D1B 0xFC /* disk 1 button pushed */
#define D2P 0xFB /* disk inserted into lower drive */
#define D2B 0xFA /* disk 2 button pushed */
#define PPP 0xF9 /* parrallel port plug */
#define MSB 0xF8 /* mouse button */
#define MSP 0xF7 /* mouse plug */
#define OFF 0xF6 /* soft off */

#define LCK 0xF2 /* Alpha lock on */
#define SFT 0xF1 /* shift key */
#define CMD 0xF0 /* command key */

#define KB_CTRL 0x0
#define KB_SHFT 0x1
#define KB_LOCK 0x2
#define KB_OFF 0x6
#define KB_MSP 0x7
#define KB_MSB 0x8
#define KB_PPORT 0x9
#define KB_D2B 0xA
#define KB_D2P 0xB
#define KB_D1B 0xC
#define KB_D1P 0xD
#define KB_STATE 0xE
#define KB_IMP 0xF

/* Cops reset codes */
#define KB_KBCOPS 0xFF
#define KB_IOCOPS 0xFE
#define KB_UNPLUG 0xFD
#define KB_CLOCKT 0xFC
#define KB_SFTOFF 0xFB
#define KB_RESERV 0xF0
#define KB_RDCLK 0xE0

/* Case change keys
 */
#define ROpt 0x4E /* mapped onto escape */
#define LOpt 0x7C /* SHIFt key */
#define Lck 0x7D /* SHIFt key */
#define Sft 0x7E /* SHIFt key */
#define Cmd 0x7F /* COMMAND key */

/* ASCII codes */
#define Del 0x7F /** backspace key */
#define Bkp 0x08
#define Esc 0x1B
#define Nl 0x0A
#define Cr 0x0D
#define Tab 0x09

/** NOTES on character mapping
 ** Left Option -> Esc
 ** Clear -> Del
 ** Enter -> Nl
 ** arrows -> cursor motion in appropriate direction (sequence \E[<Arrow-value>)
 **/
#define Alt 'D'
#define Art 'C'
#define Aup 'A'
#define Adn 'B'
#define ARROW(i,v) ((i&0x70)==0x20 && (v>='A') && (v<='D'))
/** left arrow -> control h
 ** right arrow -> control l

** up arrow -> control k
** down arrow -> control j
#define Cth ('h'&0x1F)
#define Ctl ('l'&0x1F)
#define Ctk ('k'&0x1F)
#define Ctj ('j'&0x1F)
**/

char ToLA[] = { /* convert keycode to lowercase ascii character
0 1 2 3 4 5 6 7 8 9 A B C D E F*/
KSC, D1P, D1B, D2P, D2B, PPP, MSB, MSP, OFF, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp,
Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp,
Del, '-', Alt, Art, '7', '8', '9', Aup, '4', '5', '6', Adn, '.', '2', '3', Nl,
Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp,
'-', '=', '\\', Imp, 'p', Bkp, Nl, Imp, Cr, '0', Imp, Imp, '/', '1', Esc, Imp,
'9', '0', 'u', 'i', 'j', 'k', '[', ']', 'm', 'l', ';', '\\', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
'e', '6', '7', '8', '5', 'r', 't', 'y', ' ', 'f', 'g', 'h', 'v', 'c', 'b', 'n',
'a', '2', '3', '4', '1', 'q', 's', 'w', Tab, 'z', 'x', 'd', Esc, LCK, SFT, CMD,
};

char ToCC[] = { /* convert keycode to shift-locked ascii character
0 1 2 3 4 5 6 7 8 9 A B C D E F*/
KSC, D1P, D1B, D2P, D2B, PPP, MSB, MSP, OFF, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp,
Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp,
Del, '-', Alt, Art, '7', '8', '9', Aup, '4', '5', '6', Adn, '.', '2', '3', Nl,
Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp,
'-', '=', '\\', Imp, 'p', Bkp, Nl, Imp, Cr, '0', Imp, Imp, '/', '1', Esc, Imp,
'9', '0', 'U', 'I', 'J', 'K', '[', ']', 'M', 'L', ';', '\\', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
'E', '6', '7', '8', '5', 'R', 'T', 'Y', ' ', 'F', 'G', 'H', 'V', 'C', 'B', 'N',
'A', '2', '3', '4', '1', 'Q', 'S', 'W', Tab, 'Z', 'X', 'D', Esc, LCK, SFT, CMD,
};

char ToUA[] = { /* Convert keycode into uppercase ascii character
0 1 2 3 4 5 6 7 8 9 A B C D E F*/
KSC, D1P, D1B, D2P, D2B, PPP, MSB, MSP, OFF, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp,
Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp,
Del, '-', '+', '*', '7', '8', '9', '/', '4', '5', '6', ' ', ' ', ' ', ' ', '2', '3', Nl,
Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp, Imp,
'-', '+', '|', Imp, 'P', Bkp, Nl, Imp, Cr, '0', Imp, Imp, '2', '1', Esc, Imp,
'(', ')', 'U', 'I', 'J', 'K', '{', '}', 'M', 'L', ';', '\\', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
'E', '^', '&', '*', '%', 'R', 'T', 'Y', '-', 'F', 'G', 'H', 'V', 'C', 'B', 'N',
'A', '@', '#', '$', '!', 'Q', 'S', 'W', Tab, 'Z', 'X', 'D', Esc, LCK, SFT, CMD,
};

#define Nil 0
char altkpad[] = { /* convert keycode into alternate keypad mode values
0 1 2 3 4 5 6 7 8 9 A B C D E F*/
Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil,
Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil,
Nil, 'm', 'p', 'q', 'w', 'x', 'y', 'r', 't', 'u', 'v', 'l', 'n', 'r', 's', 'm',
Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil,
Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, 'p', Nil, Nil, Nil, 'q', Nil, Nil,
Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil,
Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil,
Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil,
};

char *ccvtab[] = {
ToLA, /* LCK up, SFT up */
ToCC, /* LCK down, SFT up */
ToUA, /* LCK up, SFT down */
ToUA, /* LCK down, SFT down */
};

```

```
int l2_dtime;          /* seconds idle before dimming */
int l2_dtrap;         /* time at which screen will goes dim (mch.s) */
int l2_dimcont;       /* contrast setting when dimmed */
int l2_crate;         /* rate at which contrast change occurs */
char l2_rcflag;       /* timeout pending flag for ramp contrast */
char l2_dimmed;       /* contrast is dimmed */
char l2_bvol;         /* bell volumn */
int l2_bpitch;        /* bell pitch */
int l2_btime;         /* bell time */
char l2_contrast;     /* current contrast setting */
char l2_desired;      /* desired contrast */
char l2_defcont;      /* default contrast */
```



```

/* Various machine specific constants */

/*
 * States needed for multi-byte COPS input sequences.
 */
#define NORMALWAIT 0
#define MOUSERD 1
#define YMOUSE 2
#define RESETCODE 3
#define SHUTDOWN 4
#define CLKREAD 5 /* must be last */

#define MAXXLOC 720
#define MAXYLOC 360

/* Raster Display information ... */
/* Lowest possible (most dim) contrast setting */
#define TOTALDIM 0x3F
#define SCRNSIZE 0x8000 /* 32K */

/* Contrast value set by boot rom */
#define ONCONT 0x20

#define MAXROW 40
#define MAXCOL 90

#define BPL 90
#define V_RESO 9
#define H_RESO 8

/* The Real Time Clock and Timer (wrist watch type)
 * This device hangs off the I/O board cops and is communicated with
 * through the console VIA (ie. l2copscmd). The command byte is as follows:
 * 0x02 read clock data; the setting of the clock is returned as
 * a seven byte reset code.
 * 0x2C start clock and alarm set cycle. Upto 16 nibbles of alarm
 * and clock data (aaaaa y ddd hh mm ss t) may be sent as the
 * low nibble of the following clock command.
 * 0x1n write nibble 'n' to clock (only valid after 0x2C cmd)
 * 0x25 terminates setup cycle and enables clock, timer is still
 * disabled.
 * The Clock is connected to the power circuitry and can turn the system off
 * or be programed to turn it when the timer underflows.
 * 0x20 disbles the clock, timer, and shuts the system off.
 * 0x23 shuts off system leaving clock enabled and timer set to
 * power up on underflow.
 */
#define SHUTOFF 0x21
#define READCLOCK 0x02
#define SETCLOCK 0x2C
#define CLKNIBBLE 0x10
#define STRTCLOCK 0x25

/* Other extrainous cops commands */
#define MOUSEON 0x7F /* mouse on and int's every 28 ms */
#define MOUSEOFF 0x70 /* mouse off */
#define ENT_LOW 0x6F /* low nibble of lights */
#define ENT_HI 0x52 /* high nibble of lights */
#define NMI_LOW 0x60 /* low nibble of NMI key */
#define NMI_HI 0x50 /* high nibble of NMI key */
#define IND_HI 0x40 /* high nibble of indicator lights */
#define IND_LOW 0x30 /* low nibble of indicator lights */
#define KBENABLE 0x00 /* enable keyboard */

struct rtime { /* See page 35 LHRM */
    time_t rt_tod; /* Seconds since the Epoch */
    long rt_alm; /* Seconds remaining to trigger alarm */
    short rt_year; /* year (0 - 15) */
    short rt_day; /* julian day (1 - 366) */
    short rt_hour; /* hour (0 - 23) */
    short rt_min; /* minute (0 - 59) */
    short rt_sec; /* second (0 - 59) */
    short rt_tenth; /* tenths of a second (0 - 9) */
};

#define MILLIRATE (1000/HZ) /* rate of timer in milliseconds */

```

```

#define BOOTDEV (*(char *) (0x1B3)) /* boot device ID (used in System III) */
#define NSLOTS 3 /* number of expansion slots */
#define SLOTID1 ((short *) (0x298)) /* expansion slot 1 card ID */
#define SLOTID2 ((short *) (0x29A)) /* expansion slot 2 card ID */
#define SLOTID3 ((short *) (0x29C)) /* expansion slot 3 card ID */
#define SLOTMASK 0xFFF /* card type (ID #) of type code */
#define ID_APLNET 1 /* card ID for appletnet */
#define ID_2PORT 2 /* card ID for 2-port card */
#define ID_PRO 3 /* card ID for ProFile card */
#define ID_PRIAM 5 /* card ID for Priam card */
#define EXPVECT 27 /* intr vector for 1st of 3 exp cards */
#define devtoslot(d) ((d)==0 ? 2 : ((d)==2 ? 0 : 1))
/* dev 0 is ivec 29, dev 1 is ivec 28, dev 2 is ivec 27 */

```

```

/*
 * Copyright 1982 UniSoft Corporation
 *
 * Use of this code is subject to your disclosure agreement with AT&T,
 * Western Electric, and UniSoft Corporation
 */

#define NUMCONTX      4                /* number of user contexts */
#define MEMEND        (char **) (0x2A8) /* Addr of logical end of mem */
#define MEMBASE       (long *) (0x2A4)  /* Addr of phys beg of mem */

/*
 * Constants and definitions
 */
#define SPECIO        0xFE0000         /* Special I/O address */
#define STDIO         0xFC0000         /* Standard I/O address */

/* Special I/O Space Locations */
#define STATUSA (char *) (STDIO+0xF800) /* Status Register address */

#define STATUS *(short *) (STATUSA)    /* Status Register */
#define MEMERR *(char *) (STDIO+0xF000) /* Memory Error Address Latch */
#define SETUP_0 *(char *) (STDIO+0xE012) /* Turn setup bit off */
#define SETUP_1 *(char *) (STDIO+0xE010) /* Turn setup bit on */
#define SEG1_0 *(char *) (STDIO+0xE008) /* Turn SEG1 bit off */
#define SEG1_1 *(char *) (STDIO+0xE00A) /* Turn SEG1 bit on */
#define SEG2_0 *(char *) (STDIO+0xE00C) /* Turn SEG2 bit off */
#define SEG2_1 *(char *) (STDIO+0xE00E) /* Turn SEG2 bit on */
#define VRON *(char *) (STDIO+0xE01A) /* Turn Video retrace int on */
#define VROFF *(char *) (STDIO+0xE018) /* Turn Video retrace int off */
#define VIDADDR *(char *) (STDIO+0xE800) /* Video address latch */
#define ADDRMASK 0xFFFFFFFF           /* relevant address bits */

/*
 * Access Control Bits
 */
#define ASROS 0x400 /* Address space is read only stack */
#define ASRO 0x500 /* Address space is read only */
#define ASRWS 0x600 /* Address space is read write stack */
#define ASRW 0x700 /* Address space is read write */
#define ASIO 0x800 /* Address space is I/O */
#define ASINVAL 0xC00 /* Address space is invalid */
#define ASSPIO 0xF00 /* Address space is special I/O */
#define PROTMASK 0xF00 /* Address space protection mask */

#define UDOTBASE 0xFA0000 /* Logical base of udot area */

/* Special I/O Bits */
#define BSEGORIG 0x8 /* Segment Origin Register */

/*
 * Definitions for memory management.
 */
#define NPAGEPERSEG 256 /* number mappable pages per segment */
#define SEGMASK 0xFE0000 /* mask for segment number as address */
#define OFFMASK 0x1FFFF /* Offset mask for virtual address */
#define PAGEMASK 0x1FE00 /* Page mask for virtual address */
#define PAGESIZE 512 /* bytes per page */
#define PAGESHIFT 9 /* page shift for virtual addr */
#define DISPMASK 0x001FF /* Disp mask for virtual address */
#define SEGBASE 0xFFF /* mask for segment base */
#define SEGNMASK 0x3F /* mask for segment number as number */
#define SEGSHIFT 17 /* shift for segment number */
#define VIRTSHIFT 9 /* seg shift for virtual address */
#define ACCLIM 0x008000 /* Access Limit Register */
#define ACCSEG 0x008008 /* Access Segment Register */

/* Status register bits */
#define S_MEMERR 0x1 /* Soft Memory Error */
#define S_HMEMERR 0x2 /* Hard Memory Error */
#define S_VR 0x4 /* Vertical Retrace */

#define vtoseq(x) ((int) (x) & SEGMASK)

/* size of the current process */
#define procsz(p) ((p)->p_size)

```

```
/*
 * (C) 1983, 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with Unisoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by Unisoft.
 *
 * Driver definitions for the Lisa mouse.
 */

/* Types of mouse interrupts */
#define M_FLUSH 0 /* timeout, mouse records out of date */
#define M_PLUG 1 /* mouse unplugged or plugged in */
#define M_BUT 2 /* button changed */
#define M_CTL 3 /* apple key changed */
#define M_SFT 4 /* shift key changed */
#define M_VRT 5 /* verticle retrace interrupt (unused) */
#define M_MOVE 6 /* mouse moved */
```

```
/*
 * Copyright 1982 UniSoft Corporation
 * Use of this material is subject to your disclosure agreement with
 * AT&T, Western Electric and UniSoft Corporation.
 *
 * Parallel Port interface definitions
 */

struct device_d {
    char d_f0[1];   char d_irb;      /* I/O register B */
    char d_f1[7];   char d_ira;      /* I/O register A */
    char d_f2[7];   char d_ddrb;     /* Data Dir reg B */
    char d_f3[7];   char d_ddra;     /* Data Dir reg A */
    char d_f4[7];   char d_tlcl;     /* T1 low Latches Counter */
    char d_f5[7];   char d_tlch;     /* T1 hi Latches Counter */
    char d_f6[7];   char d_tlll;     /* T1 low Latches */
    char d_f7[7];   char d_tllh;     /* T1 hi Latches */
    char d_f8[7];   char d_t2cl;     /* T2 low Latches Counter */
    char d_f9[7];   char d_t2ch;     /* T2 hi Latches Counter */
    char d_fa[7];   char d_sr;       /* Shift Register */
    char d_fb[7];   char d_acr;      /* Aux Ctrl Reg */
    char d_fc[7];   char d_pcr;      /* Perif Ctrl Reg */
    char d_fd[7];   char d_ifr;      /* Int Flag Reg */
    char d_fe[7];   char d_ier;      /* Int Ena Reg */
    char d_ff[7];   char d_aira;     /* Alt d_ira (no handshake)*/
};

#define PPADDR (struct device_d *) (STDIO + 0xD900)

/* Breakdown of d_irb byte (above) [pg. 45 LHRM APR-82]*/
#define OCD 0x01 /* Open cable detect (paper empty)*/
#define BSY 0x02 /* Busy bit (handshake line) */
#define DEN 0x04 /* Disk Enable (buffers which drive interface lines)*/
#define DRW 0x08 /* Direction (Disk Read/Write) off = write to disk */
#define CMD 0x10 /* CMD bit (printer ON/OFF line)*/
#define PCHK 0x20 /* enable parity check(input for printer/out for disk)*/
#define DSKDIAG 0x40
#define WCNT 0x80

#define NPPDEVS 10 /* number of parallel port devices */
#define PPOK(x) (((x)>0) && ((x)<9) && ((x)!=3) && ((x)!=6)) /* 1,2,4,5,7,8 */
#define PPSLOT(x) ((x)/3) /* expansion slot number from physical unit */
```

```

/*
 * Priam datatower definitions
 *
 * (C) 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with Unisoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by Unisoft.
 */

struct pm_base {
    /* read registers */
    uchar_t status; char xx0; /* status register */
    union {
        ushort WData; /* r/w disk data register */
        uchar_t PData[2]; /* write only high byte for packet */
    } Data;
#define data Data.WData /* normal word access */
#define pdata Data.PData[0] /* byte access for packet (format) */
    uchar_t r0; char xx1; /* result 0 */
    uchar_t r1; char xx2; /* result 1 */
    uchar_t r2; char xx3; /* result 2 */
    uchar_t r3; char xx4; /* result 3 */
    uchar_t r4; char xx5; /* result 4 */
    uchar_t r5; char xx6; /* result 5 */
};

/* write registers */
#define cmdreg status /* command register */
#define p0 r0 /* parameter 0 */
#define p1 r1 /* parameter 1 */
#define p2 r2 /* parameter 2 */
#define p3 r3 /* parameter 3 */
#define p4 r4 /* parameter 4 */
#define p5 r5 /* parameter 5 */

/* channel definitions */
#define DISK 0
#define NOTUSED 1
#define TAPE 2
#define HOST 3
#define SPECIAL 4

/* flag definitions */
#define IDLING 0
#define INITING 1
#define DREADING 2
#define DWREADING 3
#define TREADING 4
#define TWREADING 5
#define TCONT 6

/* read parity register (pmpaddr, lo select space) */
#define parity status
#define PMPERROR 0x80 /* extract parity FF on reads */

/* status register bits */
#define CMD_REJECT 0x80 /* command reject */
#define CMD_DONE 0x40 /* command completion request */
#define CMD_SDONE 0x20 /* special command completion */
#define BTR_INT 0x10 /* block transfer - interrupt */
#define ISR_BUSY 0x08 /* interface busy */
#define DTREQ 0x04 /* data transfer request */
#define RW_REQ 0x02 /* r/w request */
#define DBUS_ENA 0x01 /* data bus enabled */

/* commands */
#define PMPRESET 0x7 /* software reset */
#define PMSMODE 0x8 /* set mode */
#define PMPMODE 0x9 /* read mode */
#define PMPSETP 0xC /* specify parameters */
#define PMPSPINUPW 0x82 /* sequence up disk and wait */
#define PMPRDEVPMS 0x85 /* read device parms */
#define PMPSPINDN 0x81 /* sequence down disk */
#define PMNWRITE 0x42 /* write data, retries disabled */

#define PMNREAD 0x43 /* read data, retries disabled */
#define PMWRITE 0x52 /* write data, retries enabled */
#define PMREAD 0x53 /* read data, retries enabled */
#define PPMWRK 0x62 /* write tape file mark */
#define PPMRMRK 0x63 /* read tape file mark */
#define PMPVERIFY 0x64 /* verify tape mark */
#define PMPREWIND 0x6A /* rewind tape */
#define PMPERASE 0x6F /* erase tape */
#define PMPCBTI 0x01 /* clear "block transfer intr" */
#define PMPKTXFER 0xB0 /* transfer packet (used for format) */
#define PMPKTRST 0xB8 /* read packet status */
#define PMPKTRRES 0xB1 /* resume packet operation */
#define PMPKTABRT 0xBF /* abort packet */
#define PMADVANCE 0xC0 /* advance file marks */
#define PMPRETEEN 0xC1 /* retention tape */

/* results */
#define NPMRES 6 /* number of result registers */
#define PMCTYPE 0x30 /* error completion type mask (r0) */
#define PMPICOMP 0x10 /* init complete (r0) */
#define PMPCCERR 0x11 /* ECC error (r0) */
#define PMPDMTOUT 0x33 /* timeout error (r0) */
#define PMNH(r) (int)((r) >> 4) /* no. heads (r1 after PMPSPINUP) */
#define PMNC(r1,r2) (int)((((r1)&0xF)<<8)|(r2)) /* no. cyls (r1,r2 after PMPSPINUP) */
#define PMNS(r) (int)(r) /* no. heads (r3 after PMPSPINUP) */

/* parameters */
#define PMPLOGSECT 0x40 /* log. sector mode (p1) */
#define PMPSEL1 0x01 /* parm select 1 (p1) */
#define PMP1PARMS 0x07 /* parm select 1 parameters (p2)
    disable high performance /
    block transfer intr enabled /
    block transfer timer disable */
#define PMPSEL0 0x00 /* parm select 0 (p1) */
#define PMP0PARMS 0x03 /* parm select 0 parameters (p2)
    auto. defect management enabled /
    init. complete intr disabled /
    command complete intr enabled /
    enable parity */

#define PMPCARD 0x4000 /* offset to next card */
#define PMPHISEL 0x2000 /* offset to hi select space */
/* lo sel space (boot ROM and parity) */
#define pmlsel(u) (0xFC0000 + ((u) * PMPCARD))
#define pmpaddr(u) (((struct pm_base *)pmlsel(u))
/* hi sel space (data handling) */
#define pmaddr(u) (((struct pm_base *)pmlsel(u)) + PMPHISEL)

/* interface control bits (added to pmaddr) */
#define PM_N 0x200 /* no device cycle */
#define PM_I 0x100 /* interrupt enable */
#define PM_B 0x080 /* byte mode */
#define PM_W 0x040 /* waiting (while U wait) */
#define PM_P 0x020 /* parity checking enabled */

#define pmNaddr(p) (((struct pm_base *)(((long)p) + PM_N))
#define pmIaddr(p) (((struct pm_base *)(((long)p) + PM_I))
#define pmBaddr(p) (((struct pm_base *)(((long)p) + PM_B))
#define pmWaddr(p) (((struct pm_base *)(((long)p) + PM_W))
#define pmPaddr(p) (((struct pm_base *)(((long)p) + PM_P))
#define pmBWaddr(p) (((struct pm_base *)(((long)p) + PM_B + PM_W))
#define pmBIaddr(p) (((struct pm_base *)(((long)p) + PM_B + PM_I))
#define pmBWBIaddr(p) (((struct pm_base *)(((long)p) + PM_B + PM_W + PM_I))

/*
 * Format disc packet parameters structure
 */
struct pmfmtparms {
    unsigned char pm_opcode; /* operation code = PMPFMT */
#define PMPFMT 0x02 /* opcode for format */
    unsigned char pm_devsel; /* device select = 0 (no daisy chaining) */
    unsigned char pm_scntl; /* sector control: FBD, media type */
#define PMPFBD 0x80 /* fill byte disable */
    unsigned char pm_fill; /* sector control: fill byte */
    unsigned short pm_ssize; /* sector control: sector size */
    unsigned char pm_dcntl; /* defect control: DMD, # spare sectors */
};

```

```
#define PMDMD      0x80          /* defect mapping disable */
unsigned char pm_ncyl;        /* defect control: # alternate cylinders */
unsigned char pm_cif;        /* interleave control: cyl interleave factor */
unsigned char pm_hif;        /* interleave control: head interleave factor */
unsigned char pm_sif;        /* interleave control: sect interleave factor */
unsigned char pm_sitl;       /* interleave control:
                               sector interleave table length */
};

/*
 * Packet status report structure (used by packet-based format command)
 */
struct pmfmtstat {
unsigned char pm_pid;        /* packet ID */
unsigned char pm_pstate;    /* packet state */
#define PMPKTCOMP 0x0D      /* packet completed, not resumeable */
unsigned char pm_tdf;       /* termination device flag */
unsigned char pm_pristat;   /* primary termination status */
unsigned char pm_xx[32];    /* more status - unused */
};

/*
 * Header structure
 * Used by the lisa office system. It's 4 bytes longer than the
 * header for the profile, to include the size of the disk, but
 * otherwise the same (see profile.h).
 */
struct pmheader {
unsigned short pm_version;   /* unused */
unsigned short pm_volume;   /* unused */
unsigned short pm_fileid;   /* must be 0xAAAA for the boot block */
unsigned short pm_cs1;      /* unused */
unsigned int pm_cs2;        /* unused */
unsigned short pm_relpage;  /* unused */
unsigned int pm_nblk;       /* unused */
unsigned short pm_pblk;     /* unused */
unsigned int pm_size;       /* unused - size of disk */
};
#define HDRSIZE (sizeof (struct pmheader))
#define FILEID 0xAAAA       /* file id for boot block */

#define TIMELIMIT 500000    /* timeout of about 5 secs. */

#define NPM 3
```

```
/* Disk Header Layout
 * Used by the lisa office system
 */
struct proheader {
    unsigned short ph_version; /* OS version number */
    unsigned short ph_datastk:2, /* kind of info in this block */
    ph_reserved:6,
    ph_volume:8; /* disk volume number */
    unsigned short ph_fileid; /* type of file */
    unsigned short ph_havcsum:1, /* whether ph_chksum is valid */
    ph_dataused:15; /* valid byte count */
    unsigned int ph_abbrev:24, /* page in filesystem */
    ph_chksum:8; /* checksum */
    unsigned short ph_relpage; /* page in file */
    unsigned int ph_forward:24, /* next block */
    ph_bckhigh:8; /* high byte of prev block */
    unsigned short ph_bcklow; /* low word of prev block */
};

#define HDRSIZE (sizeof (struct proheader))

struct proidblk {
    char pi_devn[13]; /* device name */
    char pi_dnum[3]; /* device number (currently 0) */
    short pi_revsn; /* micro code revision number */
    char pi_blkcnt[3]; /* number of useable blocks */
    char pi_blksize[2]; /* number of bytes per block */
    char pi_spavail; /* number of spare sectors avail */
    char pi_spalloc; /* number of spare sectors allocated */
    char pi_bbcnt; /* number of bad blocks below */
    char pi_blist[486]; /* list of bad and spared sectors */
};
```

```
/*
 * (C) 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with Unisoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by Unisoft.
 */

/* This is the command value for the console ioctl which
 * jumps to the start of unix. It is only compiled into
 * the installation unix sunix.
 */
#define RESTART ('R'<<8)      /* restart unix */
```



```

/*
 * Zilog Z8530 SCC Chip
 *
 * Copyright 1981 UniSoft Corporation
 */

/* read register 0 bits */
#define RORXRDY 0x01 /* receiver is ready */
#define ROZZERO 0x02 /* zero count */
#define ROTXRDY 0x04 /* transmitter is ready */
#define RODCD 0x08 /* data carrier detect */
#define ROSYNC 0x10 /* sync */
#define ROCTS 0x20 /* clear to send */
#define ROUNDERRUN 0x40 /* transmitter underrun */
#define ROBREAK 0x80 /* break */

/* read register 1 bits */
#define RIALLSENT 0x01 /* Async mode: all data cleared transmitter */
#define RIRRESIDUE 0x0E /* residue bits (see tables) */
#define RIPARERR 0x10 /* parity error */
#define RLOVRERR 0x20 /* lost characters due to overrun */
#define RIFRMERR 0x40 /* CRC or framing error */
#define RIENDFRAME 0x80 /* SDLC: end of frame */

/* write register 0 - command register */
#define WONULL 0x00 /* null code */
#define WOPOINTHIGH 0x08 /* select second bank of registers */
#define WOREXT 0x10 /* reset external/status interrupt */
#define WOABORT 0x18 /* send abort */
#define WORXINT 0x20 /* enable interrupt on next Rx char */
#define WORXPND 0x28 /* reset transmitter interrupt pending */
#define WORERROR 0x30 /* error reset */
#define WORIOUS 0x38 /* reset highest interrupt under service */
#define WORXCRC 0x40 /* reset Rx crc checker */
#define WOTXCRC 0x80 /* reset Tx crc generator */
#define WOTXURUN 0xC0 /* reset Tx underrun/eom latch */

/* write register 1 - xmit/rcv interrupt and data xfer mode */
#define WLEXTIEN 0x01 /* external interrupt enable */
#define WITXIEN 0x02 /* transmitter interrupt enable */
#define WIPSC 0x04 /* parity is a special condition */
#define WIRXDI 0x00 /* Rx int disable */
#define WIRXIFIRST 0x08 /* Rx interrupt on first char or special */
#define WIRXIFALL 0x10 /* Rx interrupt on all chars or special */
#define WIRXISC 0x18 /* Rx interrupt on special condition only */
#define W1WDMARCV 0x20 /* wait/dma request on receive/transmit */
#define W1WDMARF 0x40 /* wait/dma request function */
#define W1WDMARE 0x80 /* wait/dma request enable */

/* write register 2 - interrupt vector */

/* write register 3 - receive parameters and control */
#define W3RXENABLE 0x01 /* Rx enable */
#define W3SCLI 0x02 /* sync character load inhibit */
#define W3ASM 0x04 /* address search mode (sdlc) */
#define W3XCRC 0x08 /* Rx crc enable */
#define W3EHUNT 0x10 /* enter hunt mode */
#define W3AUTOENABLES 0x20 /* auto enables */
#define W35BIT 0x00 /* 5 bit data */
#define W36BIT 0x80 /* 6 bit data */
#define W37BIT 0x40 /* 7 bit data */
#define W38BIT 0xC0 /* 8 bit data */

/* write register 4 - xmit/rcv misc parameters and modes */
#define W4PARENABLE 0x01 /* parity enable */
#define W4PAREVEN 0x02 /* even parity */
#define W4ENSYNC 0x00 /* enable sync modes */
#define W41STOP 0x04 /* 1 stop bit */
#define W415STOP 0x08 /* 1.5 stop bits */
#define W42STOP 0x0C /* 2 stop bits */
#define W48SYNC 0x00 /* 8 bit sync character */
#define W416SYNC 0x10 /* 16 bit sync character */
#define W4SDLC 0x20 /* sdhc mode (01111110 flag) */
#define W4EXTSYNC 0x30 /* external sync mode */
#define W4CLK1 0x00 /* X 1 clock mode */
#define W4CLK16 0x40 /* X 16 clock mode */

#define W4CLK32 0x80 /* X 32 clock mode */
#define W4CLK64 0xC0 /* X 64 clock mode */

/* write register 5 - transmit parameter and control */
#define W5TXCRC 0x01 /* Tx crc enable */
#define W5RTS 0x02 /* rts */
#define W5CRC16 0x04 /* crc-16/sdlc */
#define W5TXENABLE 0x08 /* transmitter enable */
#define W5BREAK 0x10 /* send break */
#define W55BIT 0x00 /* 5 or less bit data */
#define W56BIT 0x40 /* 6 bit data */
#define W57BIT 0x20 /* 7 bit data */
#define W58BIT 0x60 /* 8 bit data */
#define W5DTR 0x80 /* data terminal ready */

/* write register 6 - sync chars or sdhc address field */

/* write register 7 - sync chars or sdhc address flag */

/* write register 8 - transmit buffer */

/* write register 9 - master interrupt control */
#define W9VIS 0x01 /* vector includes status */
#define W9NV 0x02 /* no vector */
#define W9DLC 0x04 /* disable lower chain */
#define W9MIE 0x08 /* master interrupt enable */
#define W9HIGHSTATUS 0x10 /* status high/low */
#define W9BRESET 0x40 /* channel b reset */
#define W9ARESET 0x80 /* channel a reset */
#define W9HRESET 0xC0 /* force hardware reset */

/* write register 10 - misc transmitter/receiver control bits */
#define W10BITSYN 0x01 /* 6 bit/8 bit sync */
#define W10LOOP 0x02 /* loop mode */
#define W10AUR 0x04 /* abort/flag on underrun */
#define W10MARK 0x08 /* mark/flag idle */
#define W10APOLL 0x10 /* go active on poll */
#define W10NRZ 0x00 /* nrz */
#define W10NRZI 0x20 /* nrzi */
#define W10FMI 0x40 /* fm1 */
#define W10FMO 0x60 /* fm0 */
#define W10CRCPRESET 0x80 /* crc preset 1/0 */

/* write register 11 - clock control mode */
#define W11OXTAL 0x00 /* crystal output */
#define W11OXCLK 0x01 /* transmit clock */
#define W11OBR 0x02 /* baud rate generator output */
#define W11ODPLL 0x03 /* dp11 output */
#define W11TRxC 0x04 /* TRxC 0/1 */
#define W11TRTxC 0x00 /* transmit clock RTxC pin */
#define W11TRRxC 0x08 /* transmit clock TRxC pin */
#define W11TR 0x10 /* transmit clock baud rate generator */
#define W11TDPPLL 0x18 /* transmit clock dp11 output */
#define W11RRTxC 0x00 /* receive clock RTxC pin */
#define W11RRRxC 0x20 /* receive clock TRxC pin */
#define W11RBR 0x40 /* receive clock baud rate generator */
#define W11RDPLL 0x60 /* receive clock dp11 output */
#define W11XTAL 0x80 /* crystal */

/* write register 12 - lower byte of baud rate generator */

/* write register 13 - upper byte of baud rate generator */

/* write register 14 - misc control bits */
#define W14BRGE 0x01 /* baud rate generator enable */
#define W14BRINT 0x02 /* baud rate generator internal */
#define W14DTRREQ 0x04 /* dtr/req function */
#define W14AUTOECHO 0x08 /* auto echo */
#define W14LOCALLB 0x10 /* local loop back */
#define W14NULL 0x00 /* null command */
#define W14ESM 0x20 /* enter search mode */
#define W14RMC 0x40 /* reset missing clock */
#define W14DPLL 0x60 /* disable dp11 */
#define W14SBRGEN 0x80 /* set source baud rate generator */
#define W14SRTxC 0xA0 /* set source RTxC */
#define W14SFM 0xC0 /* set fm mode */

```

```
#define W14SNR2I      0xE0    /* set nrzi mode */

struct device {
    char    dum1;
    char    csr;
    char    dum2[3];
    char    data;
};

/* structure for storage of port reset code and baud generator speed
 */
struct scoline {
    char reset;
    long speed;
};
```

```

/*
 * Sony definitions
 *
 * (C) 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with Unisoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by Unisoft.
 */

struct s_iob {
    char dum00;    unsigned char gobyte;
    char dum01;    unsigned char cmd;
#define mask cmd
    char dum02;    unsigned char drive;
    char dum03;    unsigned char side;
    char dum04;    unsigned char sector;
    char dum05;    unsigned char track;
    char dum06;    unsigned char unused;
    char dum07;    unsigned char confirm;
    char dum08;    unsigned char status;
    char dum09;    unsigned char format;
    char dum11;    unsigned char type;
    char dum12;    unsigned char error;
    char dum13;    unsigned char seek;
    char dum14[23]; unsigned char rom_id;        /* FCC031 */
    char dum15[15]; unsigned char diskIn;       /* FCC041 */
    char dum16[7];  unsigned char drv_connect;  /* FCC049 */
    char dum17[21]; unsigned char intr_status;  /* FCC05F */
    char dum18[49]; unsigned char d_strt_bitslip;
    char dum19;    unsigned char d_end_bitslip;
    char dum20;    unsigned char d_checksum;
    char dum21;    unsigned char a_strt_bitslip;
    char dum22;    unsigned char a_end_bitslip;
    char dum23;    unsigned char a_wrng_sector;
    char dum24;    unsigned char a_wrng_track;
    char dum25;    unsigned char a_checksum;
    char dum26[88]; unsigned char cmd_index;
};

struct sn_hdr {
    unsigned short version;    /* unused */
    unsigned short volume;     /* unused */
    unsigned short fileid;     /* must be 0xAAAA for the boot block */
    unsigned short relpg;      /* unused */
    unsigned short dum1;       /* unused */
    unsigned short dum2;       /* unused */
};
#define FILEID 0xAAAA          /* file id for boot block */

#define SNIOB ((struct s_iob *)0xFCC000)
#define SN_HDRBUF 0xFCC3E8    /* 12-byte header of 524-byte buffer */
#define SN_DATABUF 0xFCC400    /* real 512-byte buffer */
#define SN_CMD_BASE 0xFCC103   /* where the old commands are */

#define NSN 1
#define SN_MAXBN 800

/* SONY COMMANDS */
#define SN_READ 0
#define SN_WRITE 1
#define SN_EJECT 2
#define SN_FORMAT 3
#define SN_VERIFY 4
#define SN_FMTTRK 5
#define SN_VFYTRK 6
#define SN_RDBRUT 7
#define SN_WRTBRUT 8
#define SN_CLAMP 9

#define SN_SHAKE 0x80
#define SN_CMD 0x81
#define SN_SEEK 0x83
#define SN_CALL 0x84
#define SN_CLRST 0x85

#define SN_STMASK 0x86
#define SN_CLRINT 0x87
#define SN_WAITRM 0x88
#define SN_GOAWAY 0x89
#define SN_CLEARMSK 0x77

/* ROM revision number indicates twiggly vs. sony and fast vs. slow timer. */
#define ROMMASK 0xE0          /* look at these bits only */
#define ROMTW 0x80           /* 0=twiggies, 1=sony */
#define ROMSLOW 0x20         /* for Lisa 2s, 0=fast, 1=slow */

```

```
/*
 * Sun1 definitions
 *
 * (C) 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with Unisoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by Unisoft.
 */

/*
 * This block is returned when a "get drive parameters" command is
 * issued.
 */
struct sdparam {
    char fwmsg[31];          /* firmware message */
    unsigned char revision; /* revision number */
    unsigned char romv;     /* ROM version */
    /* next four bytes are for entire disk */
    unsigned char spt;      /* sectors per track */
    unsigned char tpc;      /* tracks per cylinder */
    unsigned char cpd_lsb;  /* cylinders per drive (least sig byte) */
    unsigned char cpd_msb;  /* cylinders per drive (most sig byte) */
    /* next three bytes are for specific virtual drive */
    unsigned char nblk_lsb; /* capacity in 512-byte blks (least sig byte) */
    unsigned char nblk_nsb; /* capacity in 512-byte blks (next sig byte) */
    unsigned char nblk_msb; /* capacity in 512-byte blks (most sig byte) */
    char fill[88];         /* fill to 128 bytes */
};

/*
 * This block is used to issue a copy command which copies from one
 * virtual drive to another. The tape unit is virtual drive 9, and
 * the disk can be divided into virtual drives 1-7. If the command
 * returns a non-zero status then the amount that was not copied is
 * specified in 10 bytes corresponding to the bytes from srcvd to
 * mcount. These blocks can be transferred using normal reads and
 * writes (in start/stop mode).
 */
struct copyparam {
    unsigned char command; /* copy command = 0xE3 */
    unsigned char srcvd;   /* source virtual drive */
    unsigned char lsrcstart; /* source starting sector */
    unsigned char msrstart; /* source starting sector */
    unsigned char fill1;   /* upper byte of starting sector */
    unsigned char destvd;  /* destination virtual drive */
    unsigned char ldeststart; /* destination starting sector */
    unsigned char mdeststart; /* destination starting sector */
    unsigned char fill2;   /* upper byte of starting sector */
    unsigned char lcount;  /* sector count for copy */
    unsigned char mcount;  /* sector count for copy */
    unsigned char chksum;  /* all 12 bytes add to zero */
};

#define TAPEVD 9
```

```
/*
 * (C) 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with Unisoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by Unisoft.
 */

#define PRNSWAP 2400      /* swap size for ProFile */
#define CVNSWAP 3959    /* swap size for Corvus */
#define PMNSWAP 4000    /* swap size for Priam */
```

```

/* @(#)acct.c 1.2 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/system.h"
#include "sys/acct.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/inode.h"
#include "sys/file.h"

/*
 * Perform process accounting functions.
 */

sysacct()
{
    register struct inode *ip;
    register struct a {
        char *fname;
    } *uap;
    static aclock;

    uap = (struct a *)u.u_ap;
    if (aclock || !suser())
        return;
    aclock++;
    switch (uap->fname) {
    case NULL:
        if (acctp) {
            plock(acctp);
            iput(acctp);
            acctp = NULL;
        }
        break;
    default:
        if (acctp) {
            u.u_error = EBUSY;
            break;
        }
        ip = namei(uap->fname, 0);
        if (ip == NULL)
            break;
        if ((ip->i_mode & IFMT) != IFREG)
            u.u_error = EACCES;
        else
            (void) access(ip, IWRITE);
        if (u.u_error) {
            iput(ip);
            break;
        }
        acctp = ip;
        prele(ip);
    }
    aclock--;
}

/*
 * On exit, write a record on the accounting file.
 */
acct(st)
{
    register struct inode *ip;
    register struct user *up;
    register struct acct *ap;
    off_t siz;

    if ((ip=acctp) == NULL)
        return;
    up = &u;
    plock(ip);
    ap = &acctbuf;
    bcopy((caddr_t)up->u_comm, (caddr_t)ap->ac_comm, sizeof(ap->ac_comm));
    ap->ac_btime = up->u_start;
    ap->ac_utime = compress(up->u_utime);

```

```

    ap->ac_stime = compress(up->u_stime);
    ap->ac_etime = compress(lbolt - up->u_ticks);
    ap->ac_mem = compress(up->u_mem);
    ap->ac_io = compress(up->u_ioch);
    ap->ac_rw = compress(up->u_lor+up->u_iow);
    ap->ac_uid = up->u_ruid;
    ap->ac_gid = up->u_rgid;
    ap->ac_tty = up->u_ttyp ? up->u_ttyd : NODEV;
    ap->ac_stat = st;
    ap->ac_flag = up->u_acflag;
    siz = ip->i_size;
    up->u_offset = siz;
    up->u_base = (caddr_t)ap;
    up->u_count = sizeof(acctbuf);
    up->u_segflg = 1;
    up->u_error = 0;
    up->u_limit = (daddr_t)5000;
    up->u_fmode = FWRITE;
    writel(ip);
    if (up->u_error)
        ip->i_size = siz;
    prele(ip);
}

/*
 * Produce a pseudo-floating point representation
 * with 3 bits base-8 exponent, 13 bits fraction.
 */
compress(t)
register time_t t;
{
    register exp = 0, round = 0;

    while (t >= 8192) {
        exp++;
        round = t&04;
        t >>= 3;
    }
    if (round) {
        t++;
        if (t >= 8192) {
            t >>= 3;
            exp++;
        }
    }
    return((exp<<13) + t);
}

```

```
/*      af.c      4.7      82/10/17      */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/sysm.h"
#include "net/misc.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "net/mbuf.h"
#include "net/protosw.h"
#include "net/af.h"

/*
 * Address family support routines
 */
int      null_hash(), null_netmatch();
#define AFNULL \
    { null_hash,      null_netmatch }

#ifdef INET
extern int inet_hash(), inet_netmatch();
#define AFINET \
    { inet_hash,      inet_netmatch }
#else
#define AFINET AFNULL
#endif

#ifdef PUP
extern int pup_hash(), pup_netmatch();
#define AFPUP \
    { pup_hash,      pup_netmatch }
#else
#define AFPUP AFNULL
#endif

struct afswitch afswitch[AF_MAX] = {
    AFNULL, AFNULL, AFINET, AFINET, AFPUP,
    AFNULL, AFNULL, AFNULL, AFNULL, AFNULL,
    AFNULL
};

/*ARGSUSED*/
null_hash(addr, hp)
    struct sockaddr *addr;
    struct afhash *hp;
{
    hp->afh_nethash = hp->afh_hosthash = 0;
}

/*ARGSUSED*/
null_netmatch(a1, a2)
    struct sockaddr *a1, *a2;
{
    return (0);
}
```

```

/* @(#)alloc.c 1.4 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/mount.h"
#include "sys/filsys.h"
#include "sys/fblk.h"
#include "sys/buf.h"
#include "sys/inode.h"
#include "sys/file.h"
#include "sys/ino.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/var.h"

/*
 * alloc will obtain the next available free disk block from the free list
 * of the specified device.
 * The super block has up to NICFREE remembered free blocks;
 * the last of these is read to obtain NICFREE more . . .
 *
 * no space on dev x/y -- when the free list is exhausted.
 */
struct buf *
alloc(dev)
dev_t dev;
{
    register daddr_t bno;
    register struct filsys *fp;
    register struct buf *bp;

    fp = getfs(dev);
    while(fp->s_flock)
        (void) sleep((caddr_t)&fp->s_flock, PINOD);
    do {
        if (fp->s_nfree <= 0)
            goto nospace;
        bno = fp->s_free[--fp->s_nfree];
        if (bno == 0)
            goto nospace;
    } while (badblock(fp, bno, dev));
    if (fp->s_nfree <= 0) {
        fp->s_flock++;
        bp = bread(dev, bno);
        if (u.u_error == 0) {
            fp->s_nfree = (bp->b_un.b_fblk)->df_nfree;
            bcopy((caddr_t)(bp->b_un.b_fblk)->df_free,
                (caddr_t)fp->s_free, sizeof(fp->s_free));
        }
        brelse(bp);
        /*
         * Prevent "dups in free"
         */
        bp = getblk(dev, SUPERB);
        bcopy((caddr_t)fp, bp->b_un.b_addr, sizeof(struct filsys));
        fp->s_fmod = 0;
        fp->s_time = time;
        bwrite(bp);
        fp->s_flock = 0;
        wakeup((caddr_t)&fp->s_flock);
    }
    if (fp->s_nfree <= 0 || fp->s_nfree > NICFREE) {
        prdev("Bad free count", dev);
        goto nospace;
    }
    bp = getblk(dev, bno);
    clrbuf(bp);
    if (fp->s_tfree) fp->s_tfree--;
    fp->s_fmod = 1;
    return(bp);
}

nospace:
    fp->s_nfree = 0;

```

```

        fp->s_tfree = 0;
        delay(v.v_hz<<2);
        prdev("no space", dev);
        u.u_error = ENOSPC;
        return(NULL);
    }

/*
 * place the specified disk block back on the free list of the
 * specified device.
 */
free(dev, bno)
dev_t dev;
daddr_t bno;
{
    register struct filsys *fp;
    register struct buf *bp;

    fp = getfs(dev);
    fp->s_fmod = 1;
    while(fp->s_flock)
        (void) sleep((caddr_t)&fp->s_flock, PINOD);
    if (badblock(fp, bno, dev))
        return;
    if (fp->s_nfree <= 0) {
        fp->s_nfree = 1;
        fp->s_free[0] = 0;
    }
    if (fp->s_nfree >= NICFREE) {
        fp->s_flock++;
        bp = getblk(dev, bno);
        (bp->b_un.b_fblk)->df_nfree = fp->s_nfree;
        bcopy((caddr_t)fp->s_free,
            (caddr_t)(bp->b_un.b_fblk)->df_free,
            sizeof(fp->s_free));
        fp->s_nfree = 0;
        bwrite(bp);
        fp->s_flock = 0;
        wakeup((caddr_t)&fp->s_flock);
    }
    fp->s_free[fp->s_nfree++] = bno;
    fp->s_tfree++;
    fp->s_fmod = 1;
}

/*
 * Check that a block number is in the range between the I list
 * and the size of the device.
 * This is used mainly to check that a
 * garbage file system has not been mounted.
 *
 * bad block on dev x/y -- not in range
 */
badblock(fp, bn, dev)
register struct filsys *fp;
daddr_t bn;
dev_t dev;
{
    if (bn < fp->s_ysize || bn >= fp->s_fsize) {
        prdev("bad block", dev);
        return(1);
    }
    return(0);
}

/*
 * Allocate an unused I node on the specified device.
 * Used with file creation.
 * The algorithm keeps up to NICINOD spare I nodes in the
 * super block. When this runs out, a linear search through the
 * I list is instituted to pick up NICINOD more.
 */
struct inode *
ialloc(dev, mode, nlink)
dev_t dev;

```



```

{
    register struct filsys *fp;
    register struct inode *ip;
    register i;
    register struct buf *bp;
    struct dinode *dp;
    ino_t ino;
    daddr_t adr;

loop:
    fp = getfs(dev);
    while(fp->s_ilock)
        (void) sleep((caddr_t)&fp->s_ilock, PINOD);
    if (fp->s_ninode > 0
        && (ino = fp->s_inode[--fp->s_ninode])) {
        ip = iget(dev, ino);
        if (ip == NULL)
            return(NULL);
        if (ip->i_mode == 0) {
            /* found inode: update now to avoid races */
            ip->i_mode = mode;
            ip->i_nlink = nlink;
            ip->i_flag |= IACC|IUPD|ICHG|ISYN;
            ip->i_uid = u.u_uid;
            ip->i_gid = u.u_gid;
            ip->i_size = 0;
            for (i=0; i<NADDR; i++)
                ip->i_addr[i] = 0;
            if (fp->s_tinode) fp->s_tinode--;
            fp->s_fmod = 1;
            iupdat(ip, &time, &time);
            return(ip);
        }
        /*
         * Inode was allocated after all.
         * Look some more.
         */
        iupdat(ip, &time, &time);
        iput(ip);
        goto loop;
    }
    fp->s_ilock++;
    fp->s_ninode = NICINOD;
    ino = Fsinos(dev, fp->s_inode[0]);
    for(adr = Fsitod(dev, ino); adr < fp->s_ishize; adr++) {
        bp = bread(dev, adr);
        if (u.u_error) {
            brelse(bp);
            ino += Fsinopb(dev);
            continue;
        }
        dp = bp->b_un.b_dino;
        for(i=0; i<Fsinopb(dev); i++,ino++,dp++) {
            if (fp->s_ninode <= 0)
                break;
            if (dp->di_mode == 0)
                fp->s_inode[--fp->s_ninode] = ino;
        }
        brelse(bp);
        if (fp->s_ninode <= 0)
            break;
    }
    fp->s_ilock = 0;
    wakeup((caddr_t)&fp->s_ilock);
    if (fp->s_ninode > 0) {
        fp->s_inode[fp->s_ninode-1] = 0;
        fp->s_inode[0] = 0;
    }
    if (fp->s_ninode != NICINOD) {
        fp->s_ninode = NICINOD;
        goto loop;
    }
    fp->s_ninode = 0;
    prdev("Out of inodes", dev);
    u.u_error = ENOSPC;
    fp->s_tinode = 0;

    return(NULL);
}
/*
 * Free the specified I node on the specified device.
 * The algorithm stores up to NICINOD I nodes in the super
 * block and throws away any more.
 */
ifree(dev, ino)
dev_t dev;
ino_t ino;
{
    register struct filsys *fp;

    fp = getfs(dev);
    fp->s_tinode++;
    if (fp->s_ilock)
        return;
    fp->s_fmod = 1;
    if (fp->s_ninode >= NICINOD) {
        if (ino < fp->s_inode[0])
            fp->s_inode[0] = ino;
        return;
    }
    fp->s_inode[fp->s_ninode++] = ino;
}
/*
 * getfs maps a device number into a pointer to the incore super block.
 * The algorithm is a linear search through the mount table.
 * A consistency check of the in core free-block and i-node counts.
 *
 * bad count on dev x/y -- the count
 * check failed. At this point, all
 * the counts are zeroed which will
 * almost certainly lead to "no space"
 * diagnostic
 * panic: no fs -- the device is not mounted.
 * this "cannot happen"
 */
struct filsys *
getfs(dev)
dev_t dev;
{
    register struct mount *mp;
    register struct filsys *fp;

    for(mp = &mount[0]; mp < (struct mount *)v.va_mount; mp++)
        if (mp->m_flags == MINUSE && mp->m_dev == dev) {
            fp = mp->m_bufp->b_un.b_filsys;
            if (fp->s_nfree > NICFREE || fp->s_ninode > NICINOD) {
                prdev("bad count", dev);
                fp->s_nfree = 0;
                fp->s_ninode = 0;
            }
            return(fp);
        }
    panic("no fs");
    return(NULL);
}
/*
 * update is the internal name of 'sync'. It goes through the disk
 * queues to initiate sandbagged IO; goes through the I nodes to write
 * modified nodes; and it goes through the mount table to initiate modified
 * super blocks.
 */
update()
{
    register struct inode *ip;
    register struct mount *mp;
    register struct filsys *fp;
    register struct user *up;
    static struct inode uinode;

    if (uinode.i_flag)

```

```
    return;
up = &u;
uinode.i_flag++;
uinode.i_mode = IFBLK;
for(mp = &mount[0]; mp < (struct mount *)v.ve_mount; mp++)
    if (mp->m_flags && MINUSE) {
        fp = mp->m_bufp->b_un.b_filsys;
        if (fp->s_fmod==0 || fp->s_ilock!=0 ||
            fp->s_flock!=0 || fp->s_ronly!=0)
            continue;
        fp->s_fmod = 0;
        fp->s_time = time;
        uinode.i_rdev = mp->m_dev;
        up->u_error = 0;
        up->u_offset = SUPERBOFF;
        up->u_count = sizeof(struct filsys);
        up->u_base = (caddr_t)fp;
        up->u_segflg = 1;
        up->u_fmode = FWRITE|FSYNC;
        writei(&uinode);
    }
for(ip = &inode[0]; ip < (struct inode *)v.ve_inode; ip++)
    if ((ip->i_flag&ILOCK)==0 && ip->i_count!=0
        && (ip->i_flag&(IACC|IUPD|ICHG))) {
        ip->i_flag |= ILOCK;
        ip->i_count++;
        iupdat(ip, &time, &time);
        iput(ip);
    }
bflush(NODEV);
uinode.i_flag = 0;
}
```

```

/* @(#)bio.c 1.7 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/mmu.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/sysinfo.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/buf.h"
#include "sys/iobuf.h"
#include "sys/conf.h"
#include "sys/proc.h"
#include "sys/seg.h"
#include "sys/var.h"
#include "sys/scat.h"

/*
 * swap IO headers.
 */
struct buf      swbuf[NSWB];

/*
 * 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 one 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
 *   bread
 *   breada
 * Eventually the buffer must be released, possibly with the
 * side effect of writing it out, by using one of
 *   bwrite
 *   bdwrite
 *   bawrite
 *   brelease
 */

/*
 * Unlink a buffer from the available list and mark it busy.
 * (internal interface)
 */
#define notavail(bp) \
{\
    register s;\
\
    s = spl6();\
    bp->av_back->av_forw = bp->av_forw;\
    bp->av_forw->av_back = bp->av_back;\
    bp->b_flags |= B_BUSY;\
    bfreelist.b_bcount--;\
    splx(s);\
}

/*
 * 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. Actually the latter is always true because devices
 * don't yet return specific errors.
 */
#define geterror(bp) \
{\
    if (bp->b_flags&B_ERROR)\
        if ((u.u_error = bp->b_error)==0)\
            u.u_error = EIO;\
}

/*
 * Read in (if necessary) the block and return a buffer pointer.

```

```

*/
struct buf *
bread(dev, blkno)
dev_t dev;
daddr_t blkno;
{
    register struct buf *bp;

    sysinfo.lread++;
    bp = getblk(dev, blkno);
    if (bp->b_flags&B_DONE)
        return(bp);
    bp->b_flags |= B_READ;
    bp->b_bcount = FsBSIZE(dev);
    (*bdevsw[bmajor(dev)].d_strategy)(bp);
    u.u_ior++;
    sysinfo.bread++;
    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)
dev_t dev;
daddr_t blkno, rablkno;
{
    register struct buf *bp, *rabp;

    bp = NULL;
    if (!incore(dev, blkno)) {
        sysinfo.lread++;
        bp = getblk(dev, blkno);
        if ((bp->b_flags&B_DONE) == 0) {
            bp->b_flags |= B_READ;
            bp->b_bcount = FsBSIZE(dev);
            (*bdevsw[bmajor(dev)].d_strategy)(bp);
            u.u_ior++;
            sysinfo.bread++;
        }
    }
    if (rablkno && bfreelist.b_bcount>1 && !incore(dev, rablkno)) {
        rabp = getblk(dev, rablkno);
        if (rabp->b_flags & B_DONE)
            brelease(rabp);
        else {
            rabp->b_flags |= B_READ|B_ASYNC;
            rabp->b_bcount = FsBSIZE(dev);
            (*bdevsw[bmajor(dev)].d_strategy)(rabp);
            u.u_ior++;
            sysinfo.bread++;
        }
    }
    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;

    sysinfo.lwrite++;
    flag = bp->b_flags;
    bp->b_flags &= ~(B_READ | B_DONE | B_ERROR | B_DELRWI);
    (*bdevsw[bmajor(bp->b_dev)].d_strategy)(bp);
    u.u_iow++;

```

```

        sysinfo.bwrite++;
        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.
 */
bwrite(bp)
register struct buf *bp;
{
    sysinfo.lwrite++;
    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;
    }
}

/* Put buffer on freelist, at the beginning if B_AGE, otherwise at the end. */
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++;
    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)
register dev_t dev;
daddr_t blkno;
{
    register struct buf *bp;
    register struct buf *dp;

    blkno = FsLTOP(dev, blkno);
    dp = bhash(dev, blkno);
    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)==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.
 */
struct buf *
getblk(dev, blkno)
register dev_t dev;
daddr_t blkno;
{
    register struct buf *bp;
    register struct buf *dp;

    blkno = FsLTOP(dev, blkno);
loop:
    SPL0();
    dp = bhash(dev, blkno);
    if (dp == NULL)
        panic("devtab");
    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;
            syswait.iowait++;
            (void) sleep((caddr_t)bp, PRIBIO+2);
            syswait.iowait--;
            goto loop;
        }
        SPL0();
        notavail(bp);
        return(bp);
    }
    SPL6();
    if (bfreelist.av_forw == &bfreelist) {
        bfreelist.b_flags |= B_WANTED;
        (void) sleep((caddr_t)&bfreelist, PRIBIO+1);
        goto loop;
    }
    SPL0();
    bp = bfreelist.av_forw;
#ifdef OLD
    notavail(bp);
    if (bp->b_flags & B_DELWRI) {
        bp->b_flags |= B_ASYNC;
        bwrite(bp);
        goto loop;
    }
}
#else
    if (bp->b_flags & B_DELWRI) {

```

```

        bflush(NODEV);
        goto loop;
    }
    notavail(bp);
#endif
    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;
    bp->b_bcount = FsBSIZE(dev);
    return(bp);
}

/*
 * get an empty block,
 * not assigned to any particular device
 */
struct buf *
geteblk()
{
    register struct buf *bp;
    register struct buf *dp;

loop:
    SPL6();
    while (bfreelist.av_forw == &bfreelist) {
        bfreelist.b_flags |= B_WANTED;
        (void) sleep((caddr_t)&bfreelist, PRIBIO+1);
    }
    SPL0();
    dp = &bfreelist;
    bp = bfreelist.av_forw;
    notavail(bp);
    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;
    bp->b_bcount = SBUF_SIZE;
    return(bp);
}

/*
 * Wait for I/O completion on the buffer; return errors
 * to the user.
 */
iowait(bp)
register struct buf *bp;
{
    syswait.iowait++;
    SPL6();
    while ((bp->b_flags&B_DONE)==0)
        (void) sleep((caddr_t)bp, PRIBIO);
    SPL0();
    syswait.iowait--;
    geterror(bp);
}

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

/*
 * Zero the core associated with a buffer.
 */
clrbuf(bp)
struct buf *bp;
{
    clear((caddr_t)bp->b_un.b_words, (int)bp->b_bcount);
    bp->b_resid = 0;
}

/*
 * swap I/O
 */
swap(blkno, coreaddr, count, rdflg)
daddr_t blkno;
register coreaddr, count;
{
    static struct buf *sbp;
    register struct buf *bp;
    register int c;

#ifdef SWAPTRACE
    printf("SWAP %s %d disk=0x%x core=0x%x\n",
        (rdflg==B_READ)?"IN":"OUT", count, blkno, coreaddr);
#endif
    syswait.swap++;
    if (sbp==NULL)
        sbp = &swbuf[0];
    bp = sbp++;
    if (sbp > &swbuf[NSWB-1])
        sbp = &swbuf[0];
    SPL6();
    while (bp->b_flags&B_BUSY) {
        bp->b_flags |= B_WANTED;
        (void) sleep((caddr_t)bp, PSWP+1);
    }
    bp->b_flags = B_BUSY | B_PHYS | rdflg;
    SPL0();
    bp->b_dev = swapdev;
#ifdef NONSCATLOAD
    bp->b_un.b_addr = (caddr_t)ctob(coreaddr);
    while (count > 0) {
        if (count <= btoc(MAXCOUNT))
            c = count;
        else
            c = btoc(MAXCOUNT);
        bp->b_bcount = ctob(c);
        bp->b_blkno = swplo+blkno;
        (*bdevsw[(short)bmajor(swapdev)].d_strategy)(bp);
        u.u_iosw++;
        if (rdflg) {
            sysinfo.swapin++;
            sysinfo.bswapin += ctod(c);
        } else {
            sysinfo.swapout++;
            sysinfo.bswapout += ctod(c);
        }
    }
    SPL6();
    while((bp->b_flags&B_DONE)==0)
        (void) sleep((caddr_t)bp, PSWP);
    SPL0();
    bp->b_un.b_addr += ctob(c);
    bp->b_flags &= ~B_DONE;

```

```

    if (bp->b_flags & B_ERROR)
        panic("IO err in swap");
    count -= c;
    blkno += ctod(c);
}
#else
#define sindex coreaddr
while (count > 0) {
    if (sindex == SCATEND) {
        printf("swap error:swapping beyond process\n");
        break;
    }
    c = memcontig(sindex, count);
    bp->b_un.b_addr = (caddr_t)ctob(ixtoc(sindex));
    bp->b_bcount = ctob(c);
    bp->b_blkno = swplo+blkno;
#ifdef SWAPTRACE
    printf("    SWAP %s %d disk=0x%x sindex=0x%x\n",
        (rdflg==B_READ)?"IN":"OUT", c, blkno, sindex);
#endif
    (*bdevsw[(short)bmajor(swapdev)].d_strategy)(bp);
    u.u_iosw++;
    if (rdflg) {
        sysinfo.swapin++;
        sysinfo.bswapin += ctod(c);
    } else {
        sysinfo.swapout++;
        sysinfo.bswapout += ctod(c);
    }
    SPL6();
    while((bp->b_flags&B_DONE)==0)
        (void) sleep((caddr_t)bp, PSWP);
    SPL0();
    bp->b_flags &= ~B_DONE;
    if (bp->b_flags & B_ERROR)
        panic("IO err in swap");
    count -= c;
    blkno += ctod(c);
    while (c-- > 0 && sindex != SCATEND)
        sindex = scatmap[sindex].sc_index;
}
#endif
if (bp->b_flags&B_WANTED)
    wakeup((caddr_t)bp);
bp->b_flags &= ~(B_BUSY|B_WANTED|B_PHYS);
syswait.swap--;
#ifdef NONSCATLOAD
    return(sindex);
#endif
}
/*
 * 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;

    SPL6();
    for (bp = bfreelist.av_forw; bp != &bfreelist;) {
        if (bp->b_flags&B_DELWRI && (dev == NODEV||dev==bp->b_dev)) {
            bp->b_flags |= B_ASYNC;
            notavail(bp);
            bwrite(bp);
            SPL6();
            bp = bfreelist.av_forw;
        } else {
            if (bp->av_forw) bp = bp->av_forw;
            else panic("bflush: bad free list\n");
        }
    }
    SPL0();
}

```

```

}
/*
 * Raw I/O. The arguments are
 * The strategy routine for the device
 * A buffer header, sometimes of a special type owned by the
 * device, and sometimes from the physio pool of headers.
 * The device number
 * Read/write flag
 * Essentially all the work is computing physical addresses and
 * validating them.
 */
physio(strat, bp, dev, rw)
register struct buf *bp;
int (*strat)();
{
    register struct user *up;
    register struct proc *p;
    register unsigned base;
    register unsigned limit;
    register unsigned dsstart, dsend;
    int hpf;

    up = &u;
    p = up->u_procp;
    base = (unsigned)up->u_base;

    dsstart = v.v_ustart + ctob(stoc(ctos(up->u_tsize)));
    dsend = dsstart + ctob(up->u_dsize);
    limit = base + up->u_count - 1;

    /*
     * Check that transfer is either entirely in the
     * virtual data space or in the virtual stack space
     */
    if (limit < base) /* wraparound, base < 0, count <= 0 */
        goto bad;
    if (base >= dsstart && limit < dsend)
        goto cont;
    if (base >= v.v_uend - ctob(up->u_ssize) && limit < v.v_uend)
        goto cont;
    if (rw != B_READ && base >= v.v_ustart &&
        limit < v.v_ustart + ctob(up->u_tsize))
        goto cont;
    if (chkphys((int)base, limit))
        goto cont;

bad:
    up->u_error = EFAULT;
    return;

cont:
    if (rw)
        sysinfo.phread++;
    else
        sysinfo.phwrite++;
    syswait.physio++;
#ifdef NONSCATLOAD
    if ((p->p_flag&SCONTIG)==0) {
        p->p_flag |= SSWAPIT;
        if (runout) {
            runout = 0;
            wakeup((caddr_t)&runout);
        }
        if (runin) {
            runin = 0;
            wakeup((caddr_t)&runin);
        }
        SPL6();
        while ((p->p_flag&SCONTIG)==0)
            (void) sleep((caddr_t)scatmap, PRIBIO);
    }
#endif
    hpf = (bp == NULL);
    SPL6();
    if (hpf) {
        while ((bp = pfreelist.av_forw) == NULL) {

```

```

/* @(#)clist.c 1.2 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/tty.h"

getc(p)
register struct clist *p;
{
    register struct cblock *bp;
    register int c;

#ifdef mc68000
    /*
     * Note use of d6
     */
#endif
#ifdef lint
    register int s = 0;
#endif

    asm(" movw    sr,d6");
    asm(" movw    #0x2600,sr");

#else
    register int s;
    s = spl6();
#endif

    if (p->c_cc > 0) {
        p->c_cc--;
        bp = p->c_cf;
        c = bp->c_data[bp->c_first++]&0377;
        if (bp->c_first == bp->c_last) {
            if ((p->c_cf = bp->c_next) == NULL)
                p->c_cl = NULL;
            bp->c_next = cfreelist.c_next;
            cfreelist.c_next = bp;
            if (cfreelist.c_flag) {
                cfreelist.c_flag = 0;
                wakeup((caddr_t)&cfreelist);
            }
        } else
            c = -1;
    }
#ifdef mc68000
    asm(" movw    d6,sr");
#else
    splx(s);
#endif
    return(c);
}

putc(c, p)
register struct clist *p;
{
    register struct cblock *bp, *obp;

#ifdef mc68000
    /*
     * Note use of d7
     */
#endif
#ifdef lint
    register int s = 0;
#endif

    asm(" movw    sr,d7");
    asm(" movw    #0x2600,sr");

#else
    register int s;
    s = spl6();
#endif

    if ((bp = p->c_cl) == NULL || bp->c_last == (char)cfreelist.c_size) {
        obp = bp;
        if ((bp = cfreelist.c_next) == NULL) {
#ifdef mc68000
            asm(" movw    d7,sr");
#endif
        } else

```

```

        splx(s);
    }
#endif
    return(-1);
}
cfreelist.c_next = bp->c_next;
bp->c_next = NULL;
bp->c_first = 0; bp->c_last = 0;
if (obp == NULL)
    p->c_cf = bp;
else
    obp->c_next = bp;
p->c_cl = bp;
}
bp->c_data[bp->c_last++] = c;
p->c_cc++;
#endif
mc68000
asm(" movw    d7,sr");
#else
    splx(s);
#endif
    return(0);
}

struct cblock *
getc(f)
{
    register struct cblock *bp;
    register struct chead *cf;

#ifdef mc68000
    /*
     * Note use of d7
     */
#endif
#ifdef lint
    register int s = 0;
#endif

    asm(" movw    sr,d7");
    asm(" movw    #0x2600,sr");

#else
    register int s;
    s = spl6();
#endif

    cf = &cfreelist;
    if ((bp = cf->c_next) != NULL) {
        cf->c_next = bp->c_next;
        bp->c_next = NULL;
        bp->c_first = 0;
        bp->c_last = cf->c_size;
    }
#ifdef mc68000
    asm(" movw    d7,sr");
#endif
    splx(s);
#endif
    return(bp);
}

putc(f, bp)
register struct cblock *bp;
{
    register struct chead *cf;

#ifdef mc68000
    /*
     * Note use of d7
     */
#endif
#ifdef lint
    register int s = 0;
#endif

    asm(" movw    sr,d7");
    asm(" movw    #0x2600,sr");

    register int s;

```

```

    s = spl6();
#endif

    cf = &cfreelist;
    bp->c_next = cf->c_next;
    cf->c_next = bp;
    if (cf->c_flag) {
        cf->c_flag = 0;
        wakeup((caddr_t)cf);
    }
#endif mc68000
    asm(" movw    d7,sr");
#else
    splx(s);
#endif
}

struct cblock *
getcb(p)
register struct clist *p;
{
    register struct cblock *bp;

#ifdef mc68000
    /*
     * Note use of d7
     */
#endif lint
    register int s = 0;
#endif

    asm(" movw    sr,d7");
    asm(" movw    #0x2600,sr");
#else
    register int s;
    s = spl6();
#endif

    if ((bp = p->c_cf) != NULL) {
        p->c_cc -= bp->c_last - bp->c_first;
        if ((p->c_cf = bp->c_next) == NULL)
            p->c_cl = NULL;
    }
#ifdef mc68000
    asm(" movw    d7,sr");
#else
    splx(s);
#endif
    return(bp);
}

putcb(bp, p)
register struct cblock *bp;
register struct clist *p;
{
#ifdef mc68000
    /*
     * Note use of d7
     */
#endif lint
    register int s = 0;
#endif

    asm(" movw    sr,d7");
    asm(" movw    #0x2600,sr");
#else
    register int s;
    s = spl6();
#endif

    if (p->c_cl == NULL)
        p->c_cf = bp;
    else
        p->c_cl->c_next = bp;
    p->c_cl = bp;
    bp->c_next = NULL;
}

```

```

    p->c_cc += bp->c_last - bp->c_first;
#endif mc68000
    asm(" movw    d7,sr");
#else
    splx(s);
#endif
}

#ifdef notdef
getcb(p, cp, n)
struct clist *p;
register char *cp;
register n;
{
    register struct cblock *bp;
    register char *op;
    register on;
    register char *acp = cp;

    while (n) {
        if ((bp = p->c_cf) == NULL)
            break;
        op = &bp->c_data[bp->c_first];
        on = bp->c_last - bp->c_first;
        if (n >= on) {
            bcopy(op, cp, on);
            cp += on;
            n -= on;
            if ((p->c_cf = bp->c_next) == NULL)
                p->c_cl = NULL;
            bp->c_next = cfreelist.c_next;
            cfreelist.c_next = bp;
        } else {
            bcopy(op, cp, n);
            bp->c_first += n;
            cp += n;
            n = 0;
            break;
        }
    }
    n = cp - acp;
    p->c_cc -= n;
    return(n);
}
#endif

#ifdef notdef
putcb(p, cp, n)
struct clist *p;
register char *cp;
register n;
{
    register struct cblock *bp, *obp;
    register char *op;
    register on;
    register char *acp = cp;

    while (n) {
        if ((bp = p->c_cl) == NULL || bp->c_last == cfreelist.c_size) {
            obp = bp;
            if ((bp = cfreelist.c_next) == NULL)
                break;
            cfreelist.c_next = bp->c_next;
            bp->c_next = NULL;
            bp->c_first = 0; bp->c_last = 0;
            if (obp == NULL)
                p->c_cf = bp;
            else
                obp->c_next = bp;
            p->c_cl = bp;
        }
        op = &bp->c_data[bp->c_last];
        on = cfreelist.c_size - bp->c_last;
        if (n >= on) {
            bcopy(cp, op, on);
            cp += on;
        }
    }
}

```



```
        bp->c_last += on;
        n -= on;
    } else {
        bcopy(cp, op, n);
        cp += n;
        bp->c_last += n;
        n = 0;
        break;
    }
    n = cp - acp;
    p->c_cc += n;
    return(n);
}
#endif
```

```

/* @(#)clock.c 1.1 */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/mmu.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/sysinfo.h"
#include "sys/callo.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/proc.h"
#include "sys/text.h"
#include "sys/psl.h"
#include "sys/var.h"
#include "sys/reg.h"
#ifdef UCB_NET
#include "net/misc.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "net/if.h"
#include "net/in_system.h"
#endif

#ifdef VIRTUAL451
#define args buserr
#define a_ps ber_sr
#define a_pc ber_pc
#define a_dev ber_dev
#endif

/*
 * clock is called straight from
 * the real time clock interrupt.
 *
 * Functions:
 *   reprime clock
 *   implement callouts
 *   maintain user/system times
 *   maintain date
 *   profile
 *   alarm clock signals
 *   jab the scheduler
 */

#define PRF_ON 01
extern prfstat;

time_t time, lbolt;

#ifdef UCB_NET
/*
 * Protoslow is like lbolt, but for slow protocol timeouts, counting
 * up to (hz/PR_SLOWHZ), then causing a pfslowtimo().
 * Protofast is like lbolt, but for fast protocol timeouts, counting
 * up to (hz/PR_FASTHZ), then causing a pffasttimo().
 */
extern int protoslow;
extern int protofast;
extern int ifnetslow;
extern short netoff;
#endif

clock(ap)
struct args *ap;
{
    register struct callo *p1, *p2;
    register struct user *up;
    register struct proc *pp;
    register a, ps;
    static short lticks;
    static rqlen, sqlen;

    /*
     * if panic stop clock

```

```

*/
    if (panicstr) {
        clkstop();
        return;
    }

    up = &u;
    ps = ap->a_ps;
    if (up->u_stack[0] != STKMAGIC)
        panic("Interrupt stack overflow");

#ifdef UCB_NET
/*
 * Time moves on for protocols.
 */
    if (!netoff) {
        --protoslow; --protofast; --ifnetslow;
        if (protoslow <= 0 || protofast <= 0 || ifnetslow <= 0) {
            schednetisr(NETISR_CLOCK);
        }
    }
#endif

    /*
     * callouts
     * if none, just continue
     * else update first non-zero time
     */

    if (callout[0].c_func == NULL)
        goto out;
    p2 = &callout[0];
    while (p2->c_time <= 0 && p2->c_func != NULL)
        p2++;
    p2->c_time--;

    /*
     * if ps is high, just return
     */

    if (BASEPRI(ps))
        goto out;

    /*
     * if any callout active, update first non-zero time
     * then process necessary callouts
     */

    spltty();
    if (callout[0].c_time <= 0) {
        p1 = &callout[0];
        while (p1->c_func != 0 && p1->c_time <= 0) {
            (*p1->c_func)(p1->c_arg);
            p1++;
        }
        p2 = &callout[0];
        while (p2->c_func = p1->c_func) {
            p2->c_time = p1->c_time;
            p2->c_arg = p1->c_arg;
            p1++;
            p2++;
        }
    }

out:
    if (prfstat & PRF_ON)
        prfintr((caddr_t)ap->a_pc, ps);
    if (USERMODE(ps)) {
        a = CPU_USER;
        up->u_utime++;
        if (up->u_prof.pr_scale)
            addupc((unsigned)ap->a_pc, &up->u_prof, 1);
    } else {
        if (ap->a_dev != 0) { /* dev has old idleflg in it */
            if (syswait.lowait+syswait.swap+syswait.physio) {
                a = CPU_WAIT;

```

```

        if (syswait.iowait)
            sysinfo.wait[W_IO]++;
        if (syswait.swap)
            sysinfo.wait[W_SWAP]++;
        if (syswait.physio)
            sysinfo.wait[W_PIO]++;
    } else
        a = CPU_IDLE;
} else {
    a = CPU_KERNEL;
    up->u_stime++;
}
}
sysinfo.cpu[a]++;
pp = up->u_procp;
if (pp->p_stat==SRUN) {
    up->u_mem += (unsigned)(v.v_usize+procsize(pp));
    if (pp->p_textp) {
        a = pp->p_textp->x_ccount;
        if (a==0 || a==1)
            up->u_mem += pp->p_textp->x_size;
        else
            up->u_mem +=
                (unsigned short)pp->p_textp->x_size /
                (short)a;
    }
}
if (pp->p_cpu < 80)
    pp->p_cpu++;
lbolt++; /* time in ticks */
if (--lticks <= 0) {
    if (BASEPRI(ps))
        return;
    lticks += v.v_hz;
    ++time;
    if ((time & 3) == 0) /* entry to load average */
        loadav();
    runrun++;
    rqlen = 0;
    sqlen = 0;
    for(pp = &proc[0]; pp < (struct proc *)v.ve_procp; pp++)
    if (pp->p_stat) {
        if (pp->p_time != 127)
            pp->p_time++;
        if (pp->p_clktim)
            if (--pp->p_clktim == 0)
                psignal(pp, SIGALRM);
        pp->p_cpu >>= 1;
        if (pp->p_pri >= (PUSER-NZERO)) {
            pp->p_pri = (pp->p_cpu>>1) + PUSER +
                pp->p_nice - NZERO;
        }
        if (pp->p_stat == SRUN)
            if (pp->p_flag & SLOAD)
                rqlen++;
            else
                sqlen++;
    }
    if (rqlen) {
        sysinfo.runque += rqlen;
        sysinfo.runocc++;
    }
    if (sqlen) {
        sysinfo.swpqe += sqlen;
        sysinfo.swpocc++;
    }
    if (runin!=0) {
        runin = 0;
        setrun(&proc[0]);
    }
}
#endif VIRTUAL451
if (runout!=0) {
    runout = 0;
    setrun(&proc[0]);
}
#endif VIRTUAL451

```

```

    }
}
/*
 * timeout is called to arrange that fun(arg) is called in tim/HZ seconds.
 * An entry is sorted into the callout structure.
 * The time in each structure entry is the number of HZ's more
 * than the previous entry. In this way, decrementing the
 * first entry has the effect of updating all entries.
 *
 * The panic is there because there is nothing
 * intelligent to be done if an entry won't fit.
 */
timeout(fun, arg, tim)
int (*fun)();
caddr_t arg;
int tim;
{
    register struct callout *p1, *p2;
    register int t;
    int s;

    t = tim;
    p1 = &callout[0];
    s = spl7();
    while(p1->c_func != 0 && p1->c_time <= t) {
        t -= p1->c_time;
        p1++;
    }
    if (p1 >= (struct callout *)v.ve_call-1)
        panic("Timeout table overflow");
    p1->c_time -= t;
    p2 = p1;
    while(p2->c_func != 0)
        p2++;
    while(p2 >= p1) {
        (p2+1)->c_time = p2->c_time;
        (p2+1)->c_func = p2->c_func;
        (p2+1)->c_arg = p2->c_arg;
        p2--;
    }
    p1->c_time = t;
    p1->c_func = fun;
    p1->c_arg = arg;
    splx(s);
}

#define PDELAY (PZERO-1)
delay(ticks)
{
    extern wakeup();
    int s;

    if (ticks<=0)
        return;
    s = spl7();
    timeout(wakeup, (caddr_t)u.u_procp+1, ticks);
    (void) sleep((caddr_t)u.u_procp+1, PDELAY);
    splx(s);
}

/*
 * From here down is load average code
 */
struct lavnum {
    unsigned short high;
    unsigned short low;
};

struct lavnum avenrun[3];

/*
 * Constants for averages over 1, 5, and 15 minutes
 * when sampling at 4 second intervals.
 * (Using 'fixed-point' with 16 binary digits to right)
 */

```

```
struct lavnum cexp[3] = {
    { 61309, 4227 }, /* (x = exp(-1/15) * 65536) , 1 - x */
    { 64667, 869 }, /* (x = exp(-1/75) * 65536) , 1 - x */
    { 65245, 291 }, /* (x = exp(-1/225) * 65536) , 1 - x */
};

/* called once every four seconds */
loadav()
{
    register struct lavnum *avg;
    register struct lavnum *rcexp;
    register unsigned int j;
    register unsigned short nrun;
    register struct proc *p;

    nrun = 0;
    for (p = &proc[0]; p < (struct proc *)v.ve_proc; p++) {
        if (p->p_flag & SSYS)
            continue;
        if (p->p_stat) {
            switch (p->p_stat) {
                case SSLEEP:
                case SSTOP:
                    if (p->p_pri <= PZERO)
                        nrun++;
                    break;
                case SRUN:
                case SIDL:
                    nrun++;
                    break;
            }
        }
    }
}

/*
 * Compute a tenex style load average of a quantity on
 * 1, 5 and 15 minute intervals.
 * (Using 'fixed-point' with 16 binary digits to right)
 */
avg = avenrun;
rcexp = cexp;
for (; avg < &avenrun[3]; avg++, rcexp++) {
    j = ((avg->low * rcexp->high + 32768) >> 16)
        + (avg->high * rcexp->high)
        + (nrun * rcexp->low);
    avg->low = j & 65535;
    avg->high = j >> 16;
}
}
```

```

/*
 * (C) 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with Unisoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by Unisoft.
 *
 * co.c - "console" (ie, bitmap screen and keyboard) driver for the lisa.
 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/file.h"
#include "sys/tty.h"
#include "sys/termio.h"
#include "sys/conf.h"
#include "sys/sysinfo.h"
#include "sys/var.h"
#include "sys/reg.h"
#include "setjmp.h"
#include "sys/ioctl.h"
#include "sys/kb.h"
#include "sys/al_ioctl.h"
#ifdef SUNIX
#include "sys/reboot.h"
#endif
#include "sys/mmu.h"
#include "sys/cops.h"
#include "sys/l2.h"

int coproc();
extern int co_cnt;
extern struct tty co_tty[];
extern struct ttyptr co_ttptr[];

extern char bmbck, bmnorm;
extern char *bmscrn; /* pointer to screen -- initialized in bminit */

/* calls to the putc routine are made indirectly through
 * the te_putc pointer which is used to
 * keep track of the current state for escape character
 * processing, ie, although initialized to point to
 * the normal putc, an escape character causes other
 * functions to process the next character(s)
 */
extern int vt_putc();
int (*te_putc)()=vt_putc;
#ifdef SUNIX
extern caddr_t start;
#endif
*/

struct device {
    char csr; /* Command status register */
    char idum[2]; /* fillers */
    char dbuf; /* data buffer */
};

/* ARGSUSED */
coopen(dev, flag)
register dev;
{
    register struct tty *tp;

    if (dev >= co_cnt) {
        u.u_error = ENXIO;
        return;
    }

    tp = co_ttptr[dev].tt_tty;
    tp->t_index = dev;
    SPL6();
    if ((tp->t_state & (ISOPEN|WOPEN)) == 0) {
        tp->t_proc = coproc;
        ttinit(tp);
        tp->t_state = WOPEN | CARR_ON;
        if (dev == CONSOLE) {
            tp->t_iflag = ICRNL | ISTRIP;
            tp->t_oflag = OPOST | ONLCR | TAB3;
            tp->t_lflag = ISIG | ICANON | ECHO | ECHOK;
            tp->t_cflag = sspeed | CS8 | CREAD | HUPCL;
        }
    }
    SPL0();
    (*linesw[tp->t_line].l_open)(tp);
}

/* ARGSUSED */
coclose(dev, flag)
{
    register struct tty *tp;

    tp = co_ttptr[dev].tt_tty;
    (*linesw[tp->t_line].l_close)(tp);
}

coread(dev)
{
    struct tty *tp;

    tp = co_ttptr[dev].tt_tty;
    (*linesw[tp->t_line].l_read)(tp);
}

cowrite(dev)
{
    struct tty *tp;

    tp = co_ttptr[dev].tt_tty;
    (*linesw[tp->t_line].l_write)(tp);
}

/* ARGSUSED */
coioctl(dev, cmd, arg, mode)
{
    int i;

    switch (cmd) {
    case AL_SEVOL:
        l2_bvol = arg & 7;
        break;
    case AL_SBPITCH:
        l2_bpitch = arg & 0x1FFF;
        break;
    case AL_SBTIME:
        if (arg <= 0)
            arg = 1;
        if (arg > 10 * v.v_hz) /* limit to 10 seconds */
            arg = 10 * v.v_hz;
        l2_btime = arg;
        break;
    case AL_SDIMTIME:
        l2_dtime = arg;
        l2_dtrap = lbolt + l2_dtime;
        break;
    case AL_SDIMCONT:
        l2_dimcont = (~arg) & 0xFF;
        break;
    case AL_SDIMRATE:
        l2_crate = arg;
        break;
    case AL_SCONTRAST:
        l2_defcont = (~arg) & 0xFF;
    }
}

```

```

        l2_desired = l2_defcont;
        l2ramp(0);
        break;
case AL_SREPWAIT:
    kb_repwait = arg;
    break;
case AL_SREPDELAY:
    kb_repdelay = arg;
    break;
case AL_GBVOL:
    i = l2_bvol;
    if (copyout((caddr_t)&i, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GBPITCH:
    if (copyout((caddr_t)&l2_bpitch, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GBTIME:
    if (copyout((caddr_t)&l2_bttime, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GDIMTIME:
    if (copyout((caddr_t)&l2_dtime, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GDIMCONT:
    i = (-l2_dimcont) & 0xFF;
    if (copyout((caddr_t)&i, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GDIMRATE:
    if (copyout((caddr_t)&l2_crate, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GCONTRAST:
    i = (-l2_defcont) & 0xFF;
    if (copyout((caddr_t)&i, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GREPWAIT:
    i = kb_repwait & 0xFFFF;
    if (copyout((caddr_t)&i, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GREPDELAY:
    i = kb_repdelay & 0xFFFF;
    if (copyout((caddr_t)&i, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GBMADDR:
    if (copyout((caddr_t)&bmscrn, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_REVVIDEO:
    if (arg > 0)
        i = 0;
    else if (arg == 0)
        i = -1;
    else
        i = (bmbck)? 0 : -1;
    if (bmbck != i) {
        bmswitch();
        bmsinv();
    }
    bmbck = i;
    bmnorm = bmbck;
    break;
#endif SUNIX
case RESTART: /* jump to the start of unix */
    reinit();
    ((int (*)( ))0xC000)();
    break;
#endif SUNIX
default:
    (void) ttiocom(co_ttptr[0].tt_tty, cmd, arg, mode);

```

```

        break;
    }
}
coproc(tp, cmd)
register struct tty *tp;
{
    register struct cblock *tbuf;
    extern ttrstrt();

    switch (cmd) {
case T_TIME:
    tp->t_state &= ~TIMEOUT;
    goto start;

case T_WFLUSH:
    tbuf = &tp->t_tbuf;
    tbuf->c_size -= tbuf->c_count;
    tbuf->c_count = 0;
    /* fall through */
case T_RESUME:
    tp->t_state &= ~TTSTOP;
    goto start;

case T_OUTPUT:
start:
    if (tp->t_state & (TTSTOP|TIMEOUT|BUSY))
        break;
    tbuf = &tp->t_tbuf;
    if ((tbuf->c_ptr == 0) || (tbuf->c_count == 0)) {
        if (tbuf->c_ptr)
            tbuf->c_ptr -= tbuf->c_size;
        if (!(CPRES & (*linesw[tp->t_line].l_output)(tp)))
            break;
    }
    (*te_putc)((*tbuf->c_ptr++) & 0x7F);
    tbuf->c_count--;
    sysinfo.xmint++; /* this is the xmit interrupt */
    splx(spl());
    goto start;

case T_SUSPEND:
    tp->t_state |= TTSTOP;
    break;

case T_BLOCK:
    break;

case T_RFLUSH:
    if (!(tp->t_state & TBLOCK))
        break;
    /* fall through */

case T_UNBLOCK:
    break;

case T_BREAK:
    break;
    }
}

cointr(dev)
{
    register struct cblock *cbp;
    register int c, lcnt, flg;
    struct tty *tp;
    register char ctmp;
    char lbuf[3];

    sysinfo.rcvint++;
    c = kb_chrbuf;
    tp = co_ttptr[dev].tt_tty;
    if (tp->t_rbuf.c_ptr == NULL)
        return;
#endif NULLDEBUG
#ifdef NULLDEBUG

```

```

if( c == 0x00 ) {
    sccdebug();
    return;
}
#endif
if (tp->t_iflag & IXON) {
    ctmp = c & 0177;
    if (tp->t_state & TTSTOP) {
        if (ctmp == CSTART || tp->t_iflag & IXANY)
            (*tp->t_proc)(tp, T_RESUME);
    } else {
        if (ctmp == CSTOP)
            (*tp->t_proc)(tp, T_SUSPEND);
    }
    if (ctmp == CSTART || ctmp == CSTOP)
        return;
}
/*
 * Check for errors
 */
lcnt = 1;
flg = tp->t_iflag;
if (flg&ISTRIP)
    c &= 0177;
else {
    if (c == 0377 && flg&PARMRK) {
        lbuf[1] = 0377;
        lcnt = 2;
    }
}
/*
 * Stash character in r_buf
 */
cbp = &tp->t_rbuf;
if (lcnt != 1) {
    lbuf[0] = c;
    while (lcnt) {
        *cbp->c_ptr++ = lbuf[--lcnt];
        if (--cbp->c_count == 0) {
            cbp->c_ptr -= cbp->c_size;
            (*linesw[tp->t_line].l_input)(tp);
        }
    }
    if (cbp->c_size != cbp->c_count) {
        cbp->c_ptr -= cbp->c_size - cbp->c_count;
        (*linesw[tp->t_line].l_input)(tp);
    }
} else {
    *cbp->c_ptr = c;
    cbp->c_count--;
    (*linesw[tp->t_line].l_input)(tp);
}
}

/*
 * This version of putchar writes directly to the bitmap display
 * for those last-ditch situations when you just have to get stuff to the CRT.
 */
coputchar(c)
register c;
{
    (*te_putc)(c & 0x7F);
    if (c == '\n')
        (*te_putc)('\r');
}

```

```

/*
 * Configuration information
 */

/* #define     DISK_0 1 */

#define NBUF      30
#define NINODE    50
#define NFILE     60
#define NMOUNT    8
#define CMAPSIZ  50 /* also in reinit.c */
#define SMAPSIZ  50 /* also in reinit.c */
#define CXMAPSIZ 50
#define NCALL     15
#define NPROC     30
#define NTEXT     20
#define NSVTEXT   20
#define NCLIST    100
#define STACKGAP  8
#define NSABUF    5
#define POWER     0
#define MAXUP     25
#define NHBUF     64
#define NPBUF     4
#define NFLOCK    200
#define X25LINKS  1
#define X25BUFS   256
#define X25MAPS   30
#define X25BYTES  (16*1024)
#define CSIBNUM   20
#define VPMBSZ   8192
#define MMSG      1
#define MSGMAP    100
#define MSGMAX   8192
#define MSGMNB   16384
#define MSGMNI   50
#define MSGSSZ   8
#define MSGTQL   40
#define MSGSEG   1024
#define SEMA      1
#define SEMMAP   10
#define SEMMNI   10
#define SEMMNS   60
#define SEMMNU   30
#define SEMMSL   25
#define SEMOPM   10
#define SEMUME   10
#define SEMVMX   32767
#define SEMAEM   16384
#define SHMEM    1
#define SHMMAX   (128*1024)
#define SHMMIN   1
#define SHMMNI   100
#define SHMRK    16
#define SHMALL   512
#define STIHBUF  (ST_0*4)
#define STOHBUF  (ST_0*4)
#define STNPRNT  (ST_0>>2)
#define STIBSZ   8192
#define STOBSZ   8192

#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/conf.h"
#include "sys/cpuid.h"
#include "sys/space.h"
#include "sys/io.h"
#include "sys/termio.h"
#include "sys/reg.h"
#include "sys/scc.h"
#include "sys/pport.h"
#include "sys/swapsz.h"

extern nodev(), nulldev();
extern proopen(), proread(), prowrite(), prostrategy(), proprint(), proioctl();
extern snbopen(), snbopen(), snbclose(), sncclose(), snread(), snwrite(), snstrategy(), snprint(), sni
extern cvopen(), cvread(), cvwrite(), cvstrategy(), cvprint();
extern pmopen(), pmread(), pmwrite(), pmstrategy(), pmprint(), pmioctl();
extern coopen(), coclose(), coread(), cowrite(), coloctl();
extern syopen(), syread(), sywrite(), syioctl();
extern mmread(), mmwrite();
extern scopen(), scclose(), scread(), scwrite(), scioclt();
extern erropen(), errclose(), errread();
extern proread(), prowrite(), proioctl();
extern ejioclt();
extern msopen(), msclose(), msread(), msioctl();
extern lpopen(), lpclose(), lpwrite(), lpioctl();
extern skopen(), skclose(), skwrite();
extern rtcread(), rtcwrite();
extern teopen(), teclose(), teread(), tewrite(), teioclt();

#ifdef UCB_NET
extern int ptsopen(), ptsclose(), ptsread(), ptswrite();
extern int ptccopen(), ptcclose(), ptccread(), ptccwrite();
extern int ptsioclt(), ptcioclt();
#endif

struct bdevsw bdevsw[] = {
    proopen, nulldev, prostrategy, proprint, /* 0 */
    snbopen, snbclose, snstrategy, snprint, /* 1 */
    cvopen, nulldev, cvstrategy, cvprint, /* 2 */
    pmopen, nulldev, pmstrategy, pmprint, /* 3 */
};

struct cdevsw cdevsw[] = {
    coopen, coclose, coread, cowrite, coloclt, 0, /* 0 */
    syopen, nulldev, syread, sywrite, syioclt, 0, /* 1 */
    nulldev, nulldev, mmread, mmwrite, nodev, 0, /* 2 */
    erropen, errclose, errread, nodev, nodev, 0, /* 3 */
    scopen, scclose, scread, scwrite, scioclt, 0, /* 4 */
    proopen, nulldev, proread, prowrite, proioctl, 0, /* 5 */
    snbopen, sncclose, snread, snwrite, snioclt, 0, /* 6 */
    nulldev, nulldev, nodev, nodev, ejioclt, 0, /* 7 */
    lpopen, lpclose, nodev, lpwrite, lpioctl, 0, /* 8 */
    msopen, msclose, msread, nodev, msioctl, 0, /* 9 */
    skopen, skclose, nodev, skwrite, nodev, 0, /* 10 */
    cvopen, nulldev, cvread, cvwrite, nulldev, 0, /* 11 */
    pmopen, nulldev, pmread, pmwrite, pmioctl, 0, /* 12 */
    nulldev, nulldev, rtcread, rtcwrite, nulldev, 0, /* 13 */
    teopen, teclose, teread, tewrite, teioclt, 0, /* 14 */
};

#ifdef UCB_NET
    nodev, nodev, nodev, nodev, nodev, 0, /* 15 */
    nodev, nodev, nodev, nodev, nodev, 0, /* 16 */
    nodev, nodev, nodev, nodev, nodev, 0, /* 17 */
    nodev, nodev, nodev, nodev, nodev, 0, /* 18 */
    nodev, nodev, nodev, nodev, nodev, 0, /* 19 */
    ptccopen, ptcclose, ptccread, ptccwrite, ptcioclt, 0, /* 20 */
    ptsopen, ptsclose, ptsread, ptswrite, ptsioclt, 0, /* 21 */
#endif

};

int bdevcnt = sizeof(bdevsw)/sizeof(bdevsw[0]);
int cdevcnt = sizeof(cdevsw)/sizeof(cdevsw[0]);

#ifdef UNIX /* Sony (installation) root filesystem */
dev_t rootdev = makedev(1, 0);
dev_t pipedev = makedev(1, 0);
dev_t dumpdev = makedev(1, 0);
/* nswap and swapdev are set in lisainit in config.c */
dev_t swapdev = makedev(0, 1);
daddr_t swpio = 0;
int nswap = PRNSWAP;
#else UNIX /* ProFile root filesystem */
#define ROOTBASE 0 /* (port * 16) for port=0,1,2,4,5,7, or 8 */
dev_t rootdev = makedev(0, ROOTBASE);
dev_t pipedev = makedev(0, ROOTBASE);
dev_t dumpdev = makedev(0, ROOTBASE);

```



```

dev_t swapdev = makedev(0, ROOTBASE + 1);
daddr_t swplo = 0;
int nswap = PRNSWAP;
#endif SUNIX

int (*dump)() = nulldev;
int dump_addr = 0x0000;

int (*pwr_clr[])() = {
    (int (*)())0
};

int (*dev_init[])() = {
    (int (*)())0
};

#ifdef SCC_CONSOLE
int scputchar();
int (*putchar)() = scputchar;
#else
int coputchar();
int (*putchar)() = coputchar;
#endif

#ifdef UCB_NET
#define PTC_DEV 20
int ptc_dev = PTC_DEV;
#endif

int co_cnt = 1;
struct tty co_tty[1];

struct ttyptr co_ttptr[] = {
    1, &co_tty[0], /* tt_addr field not used */
    0,
};

int sc_cnt = NSC;
struct tty sc_tty[NSC];
char sc_modem[NSC];

struct ttyptr sc_ttptr[] = {
    0xFCD240, &sc_tty[1],
    0xFCD242, &sc_tty[0],
    0,
};

struct scline sc_line[] = {
    W9BRESET, (4000000/16), /* clock frequency b */
    W9ARESET, (4000000/16), /* clock frequency a */
};

#if NTE != 0
int te_cnt = NTE;
struct tty te_tty[NTE];
char te_dparam[NTE];
char te_modem[NTE];

struct ttyptr te_ttptr[NTE+1]; /* +1 for pstat */
#endif

/*
 * pointers to ttyptr structures for terminal monitoring programs
 */
struct ttyptr *tty_stat[] = {
    co_ttptr,
    sc_ttptr,
#ifdef NTE != 0
    te_ttptr,
#endif
    0
};

/*
 * tty output low and high water marks

```

```

*/
#define TTHIGH
#ifdef TLOW
#define M 1
#define N 1
#endif
#ifdef TTHIGH
#define M 3
#define N 1
#endif
int tthiwat[16] = {
    0*M, 60*M, 60*M, 60*M, 60*M, 60*M, 60*M, 120*M,
    120*M, 180*M, 180*M, 240*M, 240*M, 240*M, 100*M, 100*M,
};
int ttlowat[16] = {
    0*N, 20*N, 20*N, 20*N, 20*N, 20*N, 20*N, 40*N,
    40*N, 60*N, 60*N, 80*N, 80*N, 80*N, 50*N, 50*N,
};

/*
 * Default terminal characteristics
 */
char ttcchar[NCC] = {
    CINTR,
    CQUIT,
    CERASE,
    CKILL,
    CEOF,
    0,
    0,
    0
};

#ifdef lint
/* LINTLIBRARY */
forlint()
{
    bminit();
    nmikey();
    llintr((struct args *)0);
    kbintr();
    scintr((struct args *)0);
    pmintr((struct args *)0);
    ebintr(0);
    netintr();
}
#endif

#ifdef UCB_NET
#include <net/misc.h>
#include <net/ubavar.h>
extern struct uba_driver ebdriver;
struct uba_device ubdinit[] = {
    /* driver, unit, addr, flags*/
    { ebdriver, 0, (caddr_t)5, 0x59002908 }, /* net 89 */
    0
};

int iff_noarp = 0; /* 0 -> do ARP; not 0 -> no ARP */
#endif

```

```

/*
 * This file contains
 * 1. oem modifiable configuration personality parameters
 * 2. oem modifiable system specific kernel personality code
 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/map.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/proc.h"
#include "sys/buf.h"
#include "sys/iobuf.h"
#include "sys/reg.h"
#include "sys/file.h"
#include "sys/inode.h"
#include "sys/seg.h"
#include "sys/acct.h"
#include "sys/sysinfo.h"
#include "sys/var.h"
#include "sys/ipc.h"
#include "sys/shm.h"
#include "sys/termio.h"

#include "sys/conf.h"
#include "sys/cops.h"
#include "sys/pport.h"
#include "sys/local.h"
#include "sys/l2.h"
#include "sys/kb.h"
#include "sys/swapsz.h"

/*char oemmsg[] = "UniSoft Systems distribution system release 1.5";*/
char oemmsg[] = "UniSoft Systems pre-distribution system (release 1.5)";

int sspeed = B9600; /* default console speed */
int parityno = 28; /* parity interrupt vector */
int cmask = CMASK; /* default file creation mask */
int cdlimit = CDLIMIT; /* default file size limit */
char slot[NSLOTS]; /* card ID numbers for expansion cards */

/*
 * Kernel initialization functions.
 * Called from main.c while at spl7 in the kernel.
 */
oem7init() /* alias (formerly) "lisainit" */
{
#ifdef SUNIX
    int dev;
    extern dev_t swapdev;
    extern int nswap;
#endif
#ifdef SUNIX
    extern struct rtime rtime;
    extern int pmvect[];
    extern int tevect[];
    register short *sidp; /* slot ID pointer */
    register slotid, i;
    register long *ip;

    l2init(); /* setup the COPS ports */

    /* This mess disables the verticle retrace interrupt, for now.
    */
    do {
        VRON = 1;
    } while ((STATUS & S_VR) != 0);
    do {
        VROFF = 1;
    } while ((STATUS & S_VR) == 0);
#endif
}

/* Some of the initialization requires that interrupts be enabled to
 * pick up coded sequences from the keyboard cops. If interrupts were
 * masked out then the time returned by READCLOCK would fill the
 * buffer and KBENABLE, which also returns a value, would have trouble.
 */
SPL1(); /* ok, do it to me */
l2copscommand(MOUSEOFF); /* shut off mouse interrupts */
l2copscommand(READCLOCK); /* get time of day */
l2copscommand(KBENABLE); /* enable keyboard */

sninit(); /* Sony initialization */

/* Wait 'til the clock data (from READCLOCK) and keyboard ID (from
 * KBENABLE) have come in, and the keyboard is back in NORMALWAIT */
while (kb_state);
time = rtime.rt_tod;

SPL7(); /* it should be at level 7 for the rest (?) */

/* Find out what's in each of the expansion slots.
 */
for (i = 0, sidp = SLOTS; i < NSLOTS; i++, sidp++) {
    slot[i] = 0xFF; /* not supported */
    slotid = *sidp & SLOTMASK;
    if (!slotid) {
        if (iocheck((caddr_t)(STDIO+i*0x4000+1))) {
            printf("Expansion slot %d: quad serial card\n",
                i+1);
            if (teinit(i) == 0) {
                /*
                 * point to interrupt vector,
                 * set tecmar quad serial board inter loc,
                 * and initialize hdwr
                 */
                ip = &((long *) 0)[EXPIVECT+devtoslot(i)];
                *ip = (long)tevect + (long)(devtoslot(i)<<2);
            }
        }
        continue;
    }
    printf("Expansion slot %d: ", i+1);
    switch (slotid) {
        case ID_APLNET:
            printf("aplnet card\n");
            break;
        case ID_PRO:
            printf("ProFile card\n");
            break;
        case ID_2PORT:
            printf("two port card\n");
            slot[i] = PR0; /* valid */
            break;
        case ID_PRIAM:
            printf("Priam card\n");
            ip = &((long *) 0)[EXPIVECT+devtoslot(i)]; /* point to int vector */
            *ip = (long)pmvect + (long)(devtoslot(i)<<2); /* set to Priam intr */
            if (pmcinit(i) == 0) /* initialize controller */
                slot[i] = PM3; /* valid */
            break;
        default:
            printf("card ID 0x%x\n", slotid);
    }
}
scinit(); /* SCC serial initialization */
#ifdef UCB_NET
netinit();
#endif
/* Now enable the verticle retrace interrupt, used for the system clock.
 */
do {
    VRON = 1;
} while ((STATUS & S_VR) != 0);
#ifdef SUNIX
SPL0();
/* This is the first unix booted during installation so find swapdev. */

```

```

if (rootdev == makedev(SN1, 0)) {
    while (chkdev(dev = getdevnam()))
        printf("Unable to use that device\nTry again:\n");
    printf("\n\nswapdev = 0x%x\n", dev);
    swapdev = dev;
    if (major(dev) == PR0) nswap = PRNSWAP;
    else if (major(dev) == PM3) nswap = PMNSWAP;
    else if (major(dev) == CV2) nswap = CVNSWAP;
    else panic("cannot determine size of swapdev");
}
#endif SUNIX
}
/*
 * Kernel initialization functions.
 * Called from main.c while at spl0 in the kernel.
 */
oem0init()
{
}

/*
 * parityerror()
 * Called from trap for parity error traps via
 * interrupt level "parityno" (conf.c).
 * Should return non-zero for fatal errors.
 * Should return zero for a transient warning error.
 */
parityerror()
{
    printf("parity error\n");
    return(-1);
}

/*
 * reboot the system
 * called from reboot function
 */
doboot()
{
    kb state = SHUTDOWN; /* SHUTDOWN (see kb.c)*/
    SPL7(); /* extreme priority */
    rom_mon(); /* return to the ROM monitor */
    /*NOTREACHED*/
}

/*
 * OEM supplied subroutine called on process exit
 */
/* ARGSUSED */
oemexit(p)
register struct proc *p;
{
#ifdef lint
    /* for lint use p */
    p->p_flag++;
#endif
}

struct device_d *pro_da[NPPDEVS] = {
    /* DEV Description */
    PPADDR, /* 0x00 parallel port */
    (struct device_d *) (STDIO+0x2000), /* 0x10 FPC port 0 slot 1 */
    (struct device_d *) (STDIO+0x2800), /* 0x20 FPC port 1 slot 1 */
    (struct device_d *) (STDIO+0x3000), /* 0x30 FPC port 2 slot 1 !!!*/
    (struct device_d *) (STDIO+0x6000), /* 0x40 FPC port 0 slot 2 */
    (struct device_d *) (STDIO+0x6800), /* 0x50 FPC port 1 slot 2 */
    (struct device_d *) (STDIO+0x7000), /* 0x60 FPC port 2 slot 2 !!!*/
    (struct device_d *) (STDIO+0xA000), /* 0x70 FPC port 0 slot 3 */
    (struct device_d *) (STDIO+0xA800), /* 0x80 FPC port 1 slot 3 */
    (struct device_d *) (STDIO+0xB000), /* 0x90 FPC port 2 slot 3 !!!*/
};
int (*pi_fnc[NPPDEVS])(); /* slots for interrupt handler addresses */

/* Set the interrupt handler for a given parallel port controller.
 */

```

```

setppint(addr, fnc)
struct device_d *addr;
int (*fnc)();
{
    register int i;
    extern int cvint(), prointr(), lpintr();

    for (i=0; i<NPPDEVS; i++)
        if (pro_da[i] == addr) { /* found dev number */
            if (pi_fnc[i]) { /* in use */
                if (pi_fnc[i] == fnc) /* same handler */
                    return 0;
                if (pi_fnc[i] == prointr)
                    printf("ALREADY assigned to profile\n");
                else if (pi_fnc[i] == lpintr)
                    printf("ALREADY assigned to lp\n");
                else if (pi_fnc[i] == cvint)
                    printf("ALREADY assigned to corvus\n");
                else
                    printf("Assigned to unknown handler at 0x%x\n",pi_fnc[i]);
                break;
            }
            pi_fnc[i] = fnc;
            return 0;
        }
    return 1;
}

/* Free the interrupt handler slot for a given controller.
 */
freeppin(addr)
struct device_d *addr;
{
    register int i;

    for (i=0; i<NPPDEVS; i++)
        if (pro_da[i] == addr) {
            pi_fnc[i] = 0;
            return;
        }
}

/*
 * ppintr - handle interrupt from parallel port controllers
 */
ppintr(ap)
struct args *ap;
{
    register int i, j;
    register char a;
    register struct device_d *dp;
    int (*fnc)(), ebintr(), prointr(), cvint(), lpintr();
    extern char lpflg[];

    if((i = ap->a_dev) == 0) { /* special case for pp 0 */
        if(fnc == pi_fnc[i]) {
            fnc(i);
            return;
        }
    }
    j = i + 2;
    while (i < j) {
        dp = pro_da[i];
        if ((a = dp->d_ifr) & FCA1) {
            asm("nop");
            dp->d_ifr = a; /* reset interrupt */
            if (fnc == pi_fnc[i]) {
                if (fnc == lpintr)
                    lpflg[i] = 0;
                else if (fnc != ebintr &&
                    fnc != prointr &&
                    fnc != cvint) {
                    printf("pi_fnc[%d] = 0x%x invalid!\n",
                        i, fnc);
                    return;
                }
            }
        }
        i = j;
    }
}

```

```

        fnc(i);
        return;
    }
#endif INTDUMP
    ppdump(i,dp);
#endif INTDUMP
    return;
}
i++;
}
}

#ifdef INTDUMP
ppdump(n, p)
register struct device_d *p;
{
    printf("pport %d: ",n);
    printf("ifr=%x acr=%x pcr=%x ddra=%x ddrb=%x irb=%x\n",
        p->d_ifr&0xFF, p->d_acr&0xFF, p->d_pcr&0xFF, p->d_ddra&0xFF,
        p->d_ddrb&0xFF, p->d_irb&0xFF);
}
#endif INTDUMP

/*
 * called from clock if there's a panic in progress
 */
clkstop()
{
    VROFF = 1;          /* disable vertical retrace intr */
}

nmikey()
{
    int i;
    register short status;

    /* added 7/25/84 to provide more info than "NMI key".
     * (taken from section 2.8 of Lisa Theory of Operations)
     */
    printf("non-maskable interrupt: ");
    status = STATUS;
    if (status & S_SMEMERR)
        printf("soft memory error\n");
    else if (status & S_HMEMERR)
        printf("hard memory error\n");
    else
        printf("power failure/keyboard reset\n");
}

#ifdef HOWFAR
showbus();
#endif HOWFAR
for (i=0x00000; i>0; i--) ; /* delay */
}

#ifdef SUNIX
/* Get swap device name
 */
getdevnam ()
{
    char *p, *gets();
    int unit, dev;

retry:
    printf("\n\nWhere is the swap area?\n");
    printf("Enter: 'p' for builtin disk or a profile disk\n");
    printf("        'c' for Corvus disk\n");
    printf("        'pm' for Priam disk\n");
    p = gets();
    switch (p[0]) {
    case 'p':
        dev = PR0;
        if (p[1] == 'm')
            dev = PM3;
        break;
    case 'c':
        dev = CV2;
        break;

```

```

    default:
        printf("Invalid input. Try again.\n");
        goto retry;
    }
    printf("Where will the disk be?\n");
    if ((dev == PR0) || (dev == CV2)) {
        printf("Enter: '0' for builtin port\n");
        printf("        '1' for Expansion Slot 1, Bottom Port\n");
        printf("        '2' for Expansion Slot 1, Top Port\n");
        printf("        '4' for Expansion Slot 2, Bottom Port\n");
        printf("        '5' for Expansion Slot 2, Top Port\n");
        printf("        '7' for Expansion Slot 3, Bottom Port\n");
        printf("        '8' for Expansion Slot 3, Top Port\n");
        p = gets();
        switch (p[0]) {
        case '0':
        case '1':
        case '2':
        case '4':
        case '5':
        case '7':
        case '8':
            unit = p[0] - '0';
            break;
        default:
            printf("Invalid input. Try again.\n");
            goto retry;
        }
    } else { /* dev == PM3 */
        printf("Enter: '0' for Slot 1\n");
        printf("        '1' for Slot 2\n");
        printf("        '2' for Slot 3\n");
        p = gets();
        switch (p[0]) {
        case '0':
        case '1':
        case '2':
            unit = p[0] - '0';
            break;
        default:
            printf("Invalid input. Try again.\n");
            goto retry;
        }
    }
    return makedev(dev, (unit<<4) | 1);
}

chkdev(d)
{
    return(*bdevsw[bmajor(d)].d_open)(minor(d), FREAD | FWRITE);
}

/*
 * This version of getchar reads directly from the keyboard in order to get
 * swapdev when the parallel port is not available. It will not work once
 * the console has been formally opened.
 */
char kb_getchr;
cogetchar()
{
    SPL0();
    while(kb_state) ; /* wait for kb driver to finish special cmd */
    kb_getchr = 1; /* wait flag */
    while (kb_getchr) ; /* wait for it to happen */
    return kb_chrbuf;
}

/* Kernel get string routine.
 * Useful for getting information from the console before the system
 * comes up. The getchar routine will not work once the console has
 * been opened.
 */
int (*getchar)() = cogetchar;
extern int (*putchar)();

char getsbuf[100];

```

```
char *
gets ()
{
    register char *p;
    register char c;
    extern short kb_keycount;

    p = getsbuf;
    while (c = (*getchar)()) {
        switch (c) {
            case '\r':
            case '\n':
                goto out;
            case '\b':
                if (p > getsbuf) {
                    p--;
                }
                break;
            case '@':
            case 'X'&0x1F: /* line kill */
                if (p > getsbuf) {
                    p = getsbuf;
                    c = '\n'; /* echo a newline */
                }
                break;
            default:
                *p++ = c;
        }
        (*putchar)(c);
        if (p >= getsbuf + sizeof(getsbuf)) {
            printf("\nInput line too long, try again ...\n");
            p = getsbuf;
        }
    }
out:
    *p = '\0';
    (*putchar)('\n');
    return getsbuf;
}
#endif SUNIX
```

```

#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/sysm.h"
#include "sys/sysinfo.h"
#include "sys/callo.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/ipc.h"
#include "sys/shm.h"
#include "sys/proc.h"
#include "sys/text.h"
#include "sys/psl.h"
#include "sys/var.h"
#include "sys/context.h"
#include "sys/map.h"

/* #define HOWFAR */
extern struct shminfo shminfo; /* shared memory info structure */

struct context cxhdr; /* head of context structure */

/*
 * Allocate a new context freeing one if necessary
 */
struct context *
cxalloc()
{
    register struct context *cx;

    /*
     * search for unused context
     */
    for (cx = cxhdr.cx_forw; cx != &cxhdr; cx = cx->cx_forw)
        if (cx->cx_proc == 0)
            return(cxunlink(cx));

    /*
     * return the context on top of the queue
     */
    cx = cxhdr.cx_forw;
    cxrfree(cx);
    return(cxunlink(cx));
}

/*
 * Find first used context and free it
 */
cxfree()
{
    register struct context *cx;

    for (cx = cxhdr.cx_forw; cx != &cxhdr; cx = cx->cx_forw) {
        if (cx->cx_proc) {
            cxrrelse(cx);
            return(0);
        }
    }
    return(-1);
}

/*
 * Initialize the context structure linked list
 */
cxinit()
{
    register struct context *cx;
    register i;

    i = USERCX;
    cx = &cxhdr;
    cx->cx_forw = cx->cx_back = cx;
    for (cx = &context[0]; cx < &context[NUMUCONTX]; cx++) {
        cx->cx_num = cxntocx(i++);
        cx->cx_back->cx_forw = cx;
        cx->cx_forw->cx_back = cx;
        backp = (struct context **)&cxhdr.cx_forw;
        (*backp)->cx_back = cx;
        cx->cx_forw = *backp;
        *backp = cx;
        cx->cx_back = &cxhdr;
    }

    /*
     * Release the context associated with
     * a given context structure
     */
    cxrfree(cx);
    register struct context *cx;
    {
        register struct proc *p;
        register struct cxphys *cxp;
        register struct cxshm *cxs;
        int i;

        if ((p = cx->cx_proc) == NULL)
            return;
        if (cx->cx_dsize > 0) {
#ifdef HOWFAR
            printf("Freeing %d data segments at %d for pid %d\n",
                cx->cx_dsize, cx->cx_daddr, p->p_pid);
#endif
            if (cx->cx_daddr == 0)
                printf("cxrfree error. cx_daddr = 0\n");
            mfree(cxmap, (short)cx->cx_dsize, (short)cx->cx_daddr);
            cx->cx_dsize = 0;
            cx->cx_daddr = 0;
        }
        cxp = &cx->cx_phys[0];
        for (i=0; i<v.v_phys; i++) {
            if (cxp->cx_physize) {
                mfree(cxmap, (short)cxp->cx_physize,
                    (short)cxp->cx_phaddr);
                cxp->cx_physize = 0;
                cxp->cx_phaddr = 0;
            }
            cxp++;
        }
        cxs = &cx->cx_shm[0];
        for (i=0; i < shminfo.shmseg; i++) {
            if (cxs->cx_shmsize) {
                mfree(cxmap, (short)cxs->cx_shmsize,
                    (short)cxs->cx_shmaddr);
                cxs->cx_shmsize = 0;
                cxs->cx_shmaddr = 0;
            }
            cxs++;
        }
        cx->cx_proc = 0;
        p->p_context = 0;
    }
}

```

```

}

/*
 * Put a context buffer onto the tail of the context queue
 */
cxtail(cx)
register struct context *cx;
{
    register struct context **backp;

    backp = (struct context **) &cxhdr.cx_back;
    (*backp)->cx_forw = cx;
    cx->cx_back = *backp;
    *backp = cx;
    cx->cx_forw = &cxhdr;
}

/*
 * Release context resources associated
 * with a text segment.
 */
cxtxfree(xp)
register struct text *xp;
{
    register struct proc *p;

    for (p = &proc[0]; p < (struct proc *)v.ve_proc; p++)
        if (p->p_textp == xp)
            cxrelse(p->p_context);
    if (xp->x_cxaddr == 0 || xp->x_size == 0)
        return;
    mfree(cxmap, (short)ctos(xp->x_size), (short)xp->x_cxaddr);
    xp->x_cxaddr = 0;
}

/*
 * unlink a context structure from the queue
 */
cxunlink(cx)
register struct context *cx;
{
    cx->cx_back->cx_forw = cx->cx_forw;
    cx->cx_forw->cx_back = cx->cx_back;
    return(cx);
}

/*
 * Free all shared text segments
 */
txfree()
{
    register struct text *xp;
    register struct proc *p;
    int n;

    n = 0;
    for (p = &proc[0]; p < (struct proc *)v.ve_proc; p++) {
        if ((xp = p->p_textp) != NULL) {
            cxrelse(p->p_context);
            if (xp->x_cxaddr && xp->x_size) {
                mfree(cxmap, (short)ctos(xp->x_size),
                    (short)xp->x_cxaddr);
                xp->x_cxaddr = 0;
                n++;
            }
        }
    }
    return(n);
}

```

```

/*#define HOWFAR*/
#define INTSON      /* defined for an interrupting disk */

/*
 * Corvus Disk System
 */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/utsname.h"
#include "sys/buf.h"
#include "sys/eelog.h"
#include "sys/erec.h"
#include "sys/iobuf.h"
#include "sys/system.h"
#include "sys/var.h"
#include "sys/altblk.h"
#include "sys/diskformat.h"
#include "setjmp.h"
#include "sys/cops.h"
#include "sys/pport.h"
#include "sys/d_profile.h"
#include "sys/cv.h"
#include "sys/swapsz.h"

#ifdef notdef
/* defined in d_profile.h */
#define logical(x)      (minor(x) & 7)      /* eight logicals per phys */
#define interleave(x)  (minor(x) & 0x8)     /* interleave bit for swapping */
#define physical(x)    ((minor(x) & 0xF0) >> 4) /* 10 physical devs */
#endif
#define cv_addr(d)      (prodata[d].pd_da)
#define cvwait(a)       while(((a)->d_irb & ST_BUSY) == 0)

/*
 * the total space on the corvus h series is:
 * 306 cylinders are there but corvus reserves 2
 * 304 cylinders * 20 sectors per track * 6 heads =
 * 36480
 *
 * The first 100 blocks are reserved for the boot program and
 * are inaccessible via unix.
 */
#define MAXBOOT 100
struct cv_sizes {
    daddr_t sz_offset;
    daddr_t sz_size;
}
cv_sizes[] = {
    CVNSWAP+101, 32420, /* a: root filesystem */
    101, CVNSWAP, /* b: swap area (3959 blocks) */
    0, 0, /* c: unused */
    0, 0, /* d: unused */
    0, 0, /* e: unused */
    0, 0, /* f: unused */
    0, 0, /* g: unused */
    101, 1000000 /* h: filesystem using entire disk */
};

struct iostat cvstat[NPPDEVS];
struct iobuf cvtab = tabinit(CV2,cvstat); /* active buffer header */
struct buf cvrbuf;

/*
 * cvopen - check for existence of controller
 */
cvopen(dev)
register dev;
{
    register punit;
    register struct device_d *devp;
    int cvint();

```

```

extern char slot[];

punit = physical(dev);
if (punit) {
    if (!PPOK(punit) || (slot[PPSLOT(punit)] != PR0)) {
        u.u_error = ENXIO;
        return 1;
    }
}
devp = pro_da[punit];
u.u_error = 0;

if (iocheck(&devp->d_ifr)) { /* board there ? */
    if (cv_addr(punit) != devp) { /* not already setup */
        if (setppint((cv_addr(punit) = devp),cvint))
            goto fail;
        if (cvinit(&prodata[punit])) {
            freeppin(devp);
            goto fail;
        }
    }
} else {
fail:
    u.u_error = ENXIO;
    cv_addr(punit) = (struct device_d *)0;
    return 1;
}
return 0;
}

/*
 * cvinit - initialize drive first time
 */
cvinit(p)
register struct prodata *p;
{
    register struct device_d *devp = p->pd_da;
    register char irb;
    register char zero = 0;
    int pl;

    pl = spl6();
    if (devp == PPADDR) {
        devp->d_ddrb &= 0x5C; /* port B bits: 0,1,5,7 to in, 2,3,4,6 to out */
        devp->d_pcr = 0x6B; /* set controller CA2 pulse mode strobe */
        devp->d_ddra = zero; /* set port A bits to input */
        devp->d_irb |= CMD|DRW; /* set command = false set direction = in */
        devp->d_ddrb |= 0x7C;
        devp->d_irb &= -DEN; /* set enable = true */
    } else {
        devp->d_pcr = 0x6B; /* set controller CA2 pulse mode strobe */
        devp->d_ddra = zero; /* set port A bits to input */
        devp->d_irb |= CMD|DRW; /* set command = false set direction = in */
        devp->d_ddrb = 0x7C;
    }
}

#ifdef INTSON
devp->d_ier = FIRQ|FCAL;
irb = devp->d_irb;
#endif lint
pl = irb;
#endif lint
p->pd_state = SCMD;
#endif INTSON
splx(pl);
return 0;
}

cvstrategy(bp)
register struct buf *bp;
{
    register punit, lunit, bn;

    punit = physical(bp->b_dev);
    lunit = logical(bp->b_dev);
    bn = bp->b_blkno + cv_sizes[lunit].sz_offset;
    if (bp->b_blkno < 0 || bn <= MAXBOOT) {

```



```

#ifdef HOWFAR
    prdev("cvstrategy: illegal blkno", bp->b_dev);
    printf("blkno=%d bcount=%d\n", bp->b_blkno, bp->b_bcount);
#endif HOWFAR

    bp->b_flags |= B_ERROR;
    iodone(bp);
    return;
}
cvstat[punit].io_ops++;
#ifdef INTSON
    bp->b_resid = bn; /* resid for disksort */
    SPL6();
    disksort(&cvtab, bp);
#else INTSON
    bp->av_forw = (struct buf *)NULL; /* last of all bufs */
    if (cvtab.b_actf == NULL)
        cvtab.b_actf = bp; /* empty - put on front */
    else
        cvtab.b_actl->av_forw = bp; /* else put at end */
    cvtab.b_actl = bp;
#endif INTSON
    if (cvtab.b_active == 0)
        cvstart();
#ifdef INTSON
    SPL0();
#else INTSON
    while (cvtab.b_active)
        cvint();
#endif INTSON
    return;
}

cvstart()
{
    register struct buf *bp;
    register lunit, offset, bn;
    register struct device_d *addr;

loop:
    if ((bp = cvtab.b_actf) == (struct buf *)NULL)
        return;
    if (cvtab.b_active == 0) {
        bp->b_resid = bp->b_bcount;
        cvtab.b_active = 1;
    }
    lunit = logical(bp->b_dev);
    blkacty |= (1<<CV2);
    offset = bp->b_bcount - bp->b_resid;
    bn = bp->b_blkno + btod(offset); /* logical block number */
    if (bp->b_resid < BSIZE || bn >= cv_sizes[lunit].sz_size) {
        next:
#ifdef HOWFAR
        if (bp->b_resid != 0)
            printf("Unix cvstart: blkno=%d resid=%d bn=%d\n",
                bp->b_blkno, bp->b_resid, bn);
#endif HOWFAR

        blkacty &= ~(1<<CV2);
        cvtab.b_active = 0;
        if (cvtab.b_errcnt) {
            logberr(&cvtab, 0); /* errlog non-fatal errors */
            cvtab.b_errcnt = 0;
        }
        addr = cv_addr(physical(bp->b_dev));
        addr->d_ifr = addr->d_ibr; /* reset intr */
        cvtab.b_actf = bp->av_forw;
        iodone(bp);
        goto loop;
    }
    if (cwrw(minor(bp->b_dev), bn + cv_sizes[lunit].sz_offset, BSIZE,
        bp->b_flags&B_READ, bp->b_un.b_addr + offset) < 0) {
        bp->b_flags |= B_ERROR;
        logberr(&cvtab, 1); /* log fatal error */
        goto next;
    }
    return;
}

```

```

caddr_t cv_buf;
int cv_count;

cwrw(dev, blkno, n, flag, buff)
register dev;
register daddr_t blkno;
register n, flag;
register caddr_t buff;
{
    register punit;
    register struct device_d *addr;

    punit=physical(dev);
    cv_buf = buff;
    cv_count = n;
    if (cvop(punit, 4, flag==B_READ ? N_R512 : N_W512, (blkno>>16 & 0xf)+1,
        blkno & 0xff, blkno>>8 & 0xff) < 0)
        goto bad;

    if (flag == B_WRITE && cvw(punit, buff, n) < 0)
        goto bad;
    addr = cv_addr(punit);
    addr->d_ddra = 0x00; /* data direction port A bits to input */
    addr->d_ibr |= 0x08; /* bidirectional driver to input */
    return 0;

bad:
#ifdef HOWFAR
    printf("cwrw: %s error unit=%d blkno=%d n=%d buf=0x%x\n",
        flag==B_READ?"read":"write", punit, blkno, n, buff);
#endif HOWFAR
    return -1;
}

/* VARARGS3 */
cvop(unit, na, a)
{
#ifdef INTSON
    register s;
#endif INTSON
    register int *ap;
    register struct device_d *addr = cv_addr(unit);

    addr->d_ddra = 0xff; /* port A to output */
    addr->d_ibr &= ~0x08; /* bidirectional driver to output */

    ap = &a;
    if (na-- > 0) {
#ifdef INTSON
        s = spl7();
#endif INTSON
        cvwait(addr);
        addr->d_ira = *ap++;
#ifdef INTSON
        splx(s);
#endif INTSON
    }
    for (; na > 0; na--, ap++) {
        cvwait(addr);
        addr->d_ira = *ap;
    }
    return 0;
}

cvint(punit)
int punit;
{
    register struct device_d *addr;
    register struct buf *bp;
    register char status;

    (void) spl6();
    addr = cv_addr(punit);
    if (cvtab.b_active == 0) {
#ifdef HOWFAR
        printf("cvint: b_active == 0\n");

```

```

#endif HOWFAR
    if (addr == PPADDR)
        addr->d_ifr = addr->d_ifr;          /* reset intr */
    return;
}
    if (bp = cvtab.b_actf) == (struct buf *)NULL) {
#ifdef HOWFAR
        printf("cvint: b_actf == NULL\n");
#endif HOWFAR
    if (addr == PPADDR)
        addr->d_ifr = addr->d_ifr;          /* reset intr */
    return;
}
    cvwait(addr);
    if (addr == PPADDR)
        addr->d_ifr = addr->d_ifr;          /* reset intr */
    if (status = addr->d_ira) {
err:
        printf("%s error: /dev/c%d%c blkno=%d\n",
            bp->b_flags&B_READ?"read":"write", punit,
            'a'+logical(bp->b_dev), bp->b_blkno);
        cvlog(status);
        cv_count = 0;
        if (++cvtab.b_errcnt > NRETRY) {
            bp->b_flags |= B_ERROR;
            logberr(&cvtab, 1);          /* log fatal error */
            blkacty &= ~(1<<CV2);
            cvtab.b_errcnt = 0;
            cvtab.b_actf = bp->av_forw;
            cvtab.b_active = 0;
            iodone(bp);
        }
    } else if (bp->b_flags&B_READ)
        if (cvt(punit, (char *)cv_buf, cv_count) < 0)
            goto err;
/*
 * because a single buffer can take several io operations,
 * we leave it to cvstart() to figure out when it's done
 */
    bp->b_resid -= cv_count;
    cvstart();
}

cvlog(status);
register status;
{
    register struct buf *bp;
    register struct device_d *addr;
    register bn, punit, lunit;
    struct deverreg cvreg[2];

    bp = cvtab.b_actf;
    punit = physical(bp->b_dev);
    lunit = logical(bp->b_dev);
    cvtab.io_stp = &cvstat[lunit];
    addr = cv_addr(punit);
    cvreg[0].draddr = (long)&(addr->d_ira);
    cvreg[0].drvalue = status;
    cvreg[0].drname = "cv status";
    cvreg[0].drbits = "Corvus disk status code";
    cvreg[1].draddr = (long)0;
    cvreg[1].drvalue = cv_count;
    cvreg[1].drname = "count";
    cvreg[1].drbits = "byte count of transfer";
    bn = bp->b_blkno + btod(bp->b_bcount - bp->b_resid) + cv_sizes[lunit].sz_offset;
    fmberr(&cvtab,
        (unsigned)punit,
        (unsigned)0,          /* cylinder */
        (unsigned)0,          /* track */
        (unsigned)bn,        /* sector */
        (long)(sizeof(cvreg)/sizeof(cvreg[0])), /* regcnt */
        &cvreg[0],&cvreg[1]);
}

cvt(unit, buff, n)
register char *buff;

```

```

register n;
{
    register char *ira;
    register struct device_d *addr = cv_addr(unit);

    ira = &(addr->d_ira);
    cvwait(addr);
    for ( ; n > 0; n--) {
        if ((addr->d_irb & ST_HTOC) == 0)
            break;
        *ira = *buff++;
    }
    for (;;) {
        cvwait(addr);
        if ((addr->d_irb & ST_HTOC) == 0)
            break;
        *ira = 0;
    }
    cv_count -- n;
    if (n > 0) {
#ifdef HOWFAR
        printf("cvw: %d bytes short\n", n);
#endif HOWFAR
        cvlog(0);
        return -1;
    }
    return 0;
}

cvt(unit, buff, n)
register char *buff;
register n;
{
    register char *ira;
    register struct device_d *addr = cv_addr(unit);

    cvwait(addr);
    ira = &(addr->d_ira);
    for ( ; n > 0; n--) {
        if (addr->d_irb & ST_HTOC)
            break;
        *buff++ = *ira;
    }
    cv_count -- n;
    if (n > 0) {
#ifdef HOWFAR
        printf("cvt: %d bytes short\n", n);
#endif HOWFAR
        return -1;
    }
    for (;;) {
        cvwait(addr);
        if (addr->d_irb & ST_HTOC)
            break;
        n = *ira;
    }
    return 0;
}

/*****
#ifdef INTSON
#else INTSON
/* wait for controller to host direction or timeout */ /*
cvctoh(a)
register struct device_d *a;
{
    register i;

    for (i = 20; i-- > 0;);
    i = 100000;
    do
        while (--i > 0 && ((a->d_irb&ST_BUSY) == 0));
    while (i > 0 && (a->d_irb & ST_HTOC));
    if (i <= 0) {
        printf("cvctoh: timeout\n");
        return -1;
    }
}

```

```
    }
    return 0;
}
#endif INTSON
*****/

cvread(dev)
dev_t dev;
{
    physio(cvstrategy, &cvrbuf, dev, B_READ);
}

cvwrite(dev)
dev_t dev;
{
    physio(cvstrategy, &cvrbuf, dev, B_WRITE);
}

cvprint(dev, str)
char *str;
{
    printf("%s on cv drive %d, slice %d\n", str, (dev>>4)&0xF, dev&7);
}
}
```

```

#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/sysinfo.h"
#include "sys/callo.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/proc.h"
#include "sys/text.h"
#include "sys/ipc.h"
#include "sys/shm.h"
#include "sys/psl.h"
#include "sys/var.h"
#include "sys/seg.h"
#include "sys/context.h"
#include "sys/map.h"
#include "sys/errno.h"
#include "sys/scat.h"

typedef int mem_t;
short segoff; /* mmu segment offset */
extern struct shmid_ds *shm_shmem[]; /* ptrs to attached segments */
extern struct shmpt_ds shm_pte[]; /* segment attach points */
extern struct shminfo shminfo; /* shared memory info structure */

/* #define DUMPMML */
/* #define HOWFAR */
/* #define TRACEALL */

/*
 * Load the user hardware page map.
 */
sureg()
{
    register struct user *up;
    register struct phys *ph;
    register struct shmid_ds *sp;
    register short *addr;
    register a, i, j, page;
    struct text *tp;
    struct proc *p;

    up = &u;
    p = up->u_proc;
    tp = p->p_textp;

#ifdef HOWFAR
    printf("sureg:p_addr=0x%x, tsize=%d dsize=%d ssize=%d\n",
        p->p_addr, up->u_ptsize, up->u_pdsz, up->u_pssz);
#endif
    SEG1_1 = 1;
    /* SEG2_0 = 1; */
    clearmmu();

    addr = (short *)vtoseg(v.v_ustart);
    if (tp != NULL) {
        page = tp->x_caddr + segoff;
        /* map a max of NPAGEPERSEG (256) 512-byte pages per segment */
        for (i = up->u_ptsize; i > 0; i -= a) {
            a = min(NPAGEPERSEG, (unsigned)i);
            setmmu((short *)((int)addr | ACCSEG), page);
            setmmu((short *)((int)addr | ACCCLIM),
                ((up->u_xrw==RO)?ASRO:ASRW) | ((256-a) & 0xFF));
            page += a;
            addr = (short *)((long)addr + (1 << SEGSHIFT));
        }
        addr = (short *)vtoseg(ctob(stoc(ctos(btoc(v.v_ustart)+up->u_ptsize))););
    } else
        if (up->u_ptsize != 0)
            addr = (short *)((long)addr +
                ctob(stoc(ctos(up->u_ptsize))));

    /* set up data segment */

```

```

        page = p->p_addr + v.v_usize + segoff;
        /* map a max of NPAGEPERSEG (256) 512-byte pages per segment */
        for (i = up->u_pdsz; i > 0; i -= a) {
            a = min(NPAGEPERSEG, (unsigned)i);
            setmmu((short *)((int)addr | ACCSEG), page);
            setmmu((short *)((int)addr | ACCCLIM), ASRW | ((256-a) & 0xFF));
            page += a;
            addr = (short *)((long)addr + (1 << SEGSHIFT));
        }

        /* set up stack segment */
        addr = (short *)vtoseg(v.v_uend);
        page += up->u_pssz; /* stack is right after data */
        for (i = up->u_pssz; i > 0; i -= a) {
            addr = (short *)((long)addr - (1 << SEGSHIFT));
            a = min(NPAGEPERSEG, (unsigned)i);
            page -= NPAGEPERSEG;
            setmmu((short *)((int)addr | ACCSEG), page);
            setmmu((short *)((int)addr | ACCCLIM), ASRWS | (a-1));
        }

        /* set up phys() */
        for (ph = &u.u_phys[0]; ph < &u.u_phys[v.v_phys]; ph++) {
            if (ph->u_phsize) {
                page = (ph->u_phpaddr >> PAGESHIFT) + segoff;
                addr = (short *)vtoseg(ph->u_phladdr);
                for (i = ph->u_phsize; i > 0; i -= a) {
                    a = min(NPAGEPERSEG, (unsigned)i);
                    /* if ((getmmu(vtoseg(addr) | ACCCLIM) & PROTMASK) == ASINVAL) { */
                    if ((getmmu((short *)((int)addr | ACCCLIM) & PROTMASK) == ASINVAL) {
                        setmmu((short *)((int)addr | ACCSEG), page);
                        setmmu((short *)((int)addr | ACCCLIM), ASRW | ((256-a) & 0xFF));
                    }
                    page += a;
                    addr = (short *)((long)addr + (1 << SEGSHIFT));
                }
                /* dumpmml(1); /***** DEBUG *****/
            }
        }

        /* set up shared memory */
        for (i = (p - proc) * shminfo.shmseg; /* index of first shm_shmem[] */
            i < ((p - proc) + 1) * shminfo.shmseg; i++) {
            sp = shm_shmem[i];
            if (sp == NULL)
                /* no more shared mem segments this process */
                continue;
            /* shm_scat is starting physical click number */
            page = sp->shm_scat + segoff;
            addr = (short *)vtoseg(shm_pte[i].shm_segbeg);
            for (j = btoc(sp->shm_segz); j > 0; j -= a) {
                a = min(NPAGEPERSEG, (unsigned)j);
                if ((getmmu((short *)((int)addr | ACCCLIM) & PROTMASK) == ASINVAL) {
                    setmmu((short *)((int)addr | ACCSEG), page);
                    setmmu((short *)((int)addr | ACCCLIM),
                        ASRW | ((256-a) & 0xFF));
                }
                page += a;
                addr = (short *)((long)addr + (1 << SEGSHIFT));
            }
        }

        setmmu(addr, data)
        register short *addr;
        register data;
        {
            int s;

#ifdef TRACEALL
#ifdef HOWFAR
            if (data != ASINVAL)
                if (((int)addr & ACCSEG) == ACCSEG)
                    printf("setmmu:addr=0x%x, data=0x%x (0x%x)\n", addr, data, data<<9);
#endif
#endif
        }
        else

```

```

    printf("setmmu:addr=0x%x prot=0x%x, length=0x%x (%d)\n",
           addr, data & PROTMASK, data, 256 - (data & 0xFF));
#endif
#endif
    s = spl7();
    SETUP_1 = 1;
    *addr = data;
    SETUP_0 = 1;
    splx(s);
}

getmmu(addr)
register short *addr;
{
    register data;
    int s;

    s = spl7();
    SETUP_1 = 1;
    data = *addr;
    SETUP_0 = 1;
    splx(s);
    data &= 0xFFFF;
#ifdef TRACEALL
#ifdef HOWFAR
if (((int)addr & ACCSEG) == ACCSEG)
    printf("getmmu:addr=0x%x, data=0x%x (0x%x)\n", addr, data, data<<9);
else
    printf("getmmu:addr=0x%x prot=0x%x, length=0x%x (%d)\n",
           addr, data & PROTMASK, data, 256 - (data & 0xFF));
#endif
#endif
    return(data);
}

clearmmu()
{
    register data = 0xC00;          /* ASINVAL (d7) */
    register inc = 0x20000;        /* address increment (d6) */
    register s = 0;                /* saved priority (d5) */
    register short i = 32-1;      /* loop counter (d4) */
    register char *addr = (char *)0x8000; /* address ACCLIM (a5) */

#ifdef lint
    *addr = (char)i;
    *addr = (char)inc;
    *addr = (char)data;
#endif
    s = spl7();
    SETUP_1 = 1;
    asm("loop:");
    asm(" movw  d7,a5@");
    asm(" addl  d6,a5");
    asm(" movw  d7,a5@");
    asm(" addl  d6,a5");
    asm(" movw  d7,a5@");
    asm(" addl  d6,a5");
    asm(" movw  d7,a5@");
    asm(" addl  d6,a5");
    asm(" movw  d7,a5@");
    asm(" addl  d6,a5");
    asm(" dbra  d4,loop");
    SETUP_0 = 1;
    splx(s);
}

/*
 * In V7, Set up software prototype segmentation
 * registers to implement the 3 pseudo
 * text,data,stack segment sizes passed
 * as arguments.
 * The argument sep specifies if the
 * text and data+stack segments are to
 * be separated.
 * The last argument determines whether the text
 * segment is read-write or read-only.
 *
 * u.u_ptsize etc replace the proto entries on the pcp11. They
 */
}

/* are used by sureg to set up the page map.
 */
/* ARGSUSED */
estabur(nt, nd, ns, sep, xrw)
unsigned nt, nd, ns;
{
#ifdef HOWFAR
printf("estabur:nt=%d nd=%d ns=%d rw=%d\n", nt, nd, ns, xrw);
#endif
    if (verureg(nt, nd, ns, xrw))
        return(-1);
    sureg();
    return(0);
}

/*
 * verify user registers can be set up
 */
verureg(nt, nd, ns, xrw)
register unsigned nt;
unsigned nd, ns;
{
    register int s;

    /*
     * check for sufficient number of segment registers
     */
    if (ctos(nt) + ctos(nd) + ctos(ns) > ctos(btoc(v.v_uend-v.v_ustart)))
        goto bad;

    s = nd + ns + v.v_usize;
    if (nt == 0) { /* non shared text */
        if (s > maxmem)
            goto bad;
    } else { /* shared text */
        if (nt + s <= maxmem) /* text+data can fit in largest hole */
            goto ok;
        goto bad;
    }
}

ok:
    u.u_ptsize = nt; /* essentially these pass args to sureg */
    u.u_pdsize = nd;
    u.u_pssize = ns;
    u.u_xrw = xrw;
    return(0);

bad:
#ifdef HOWFAR
printf("verureg failure:nt=%d nd=%d ns=%d\n", nt, nd, ns);
#endif
    u.u_error = ENOMEM;
    return(-1);
}

#ifdef DUMPMM
char *mmu_codes[] = {
    "UNPREDICT-0", /* 0 */
    "UNPREDICT-1", /* 1 */
    "UNPREDICT-2", /* 2 */
    "UNDEFINED", /* 3 */
    "RO stack", /* 4 */
    "RO", /* 5 */
    "RW stack", /* 6 */
    "RW", /* 7 */
    "UNPREDICT-8", /* 8 */
    "IO", /* 9 */
    "UNPREDICT-A", /* A */
    "UNPREDICT-B", /* B */
    "INVALID", /* C */
    "UNPREDICT-D", /* D */
    "UNPREDICT-E", /* E */
    "SPIO", /* F */
};

/*
 * dumpmm(system)
 */

```

```

*      dump the memory management registers
*      if system is non-zero, also dump system registers
*/
dumpmm(system)
int system;
{
    printf("p_addr=0x%x tsize=0x%x dsize=0x%x ssize=0x%x\n",
           u.u_procp->p_addr, u.u_ptsize, u.u_pdsiz, u.u_pssize);
    if (system)
        dumpmm1(0);
    dumpmm1(1);
}

dumpmm1(space)
{
    register i, addr, prot, j, len;

    printf("Context %d mmu registers\n", space);
    printf("seg logical physical (clicks) permission\n");
    for (i = 0; i < 128; i++) {
        if (space == 0)
            SEG1_0 = 1;
        else
            SEG1_1 = 1;
        /* SEG2_0 = 1; */
        addr = getmmu((i << SEGSHIFT) | ACCSEG);
        prot = getmmu((i << SEGSHIFT) | ACCLIM);
        if ((prot & PROTMASK) == ASINVAL)
            continue;
        addr -= segoff;
        len = prot & 0xFF;
        if (prot & 0x100) {          /* data or stack segment */
            j = i << SEGSHIFT;
            len = 256 - len;
            printf("0x%x 0x%x-0x%x 0x%x-0x%x (%d) %s\n", i, j,
                   j+ctob(len), addr, addr+len, len,
                   mmu_codes[(prot&PROTMASK)>>8]);
        } else {
            len++;
            j = (i+1) << SEGSHIFT;
            addr += NPAGEPERSEG;
            printf("0x%x 0x%x-0x%x 0x%x-0x%x (%d) %s\n", i,
                   j-ctob(len), j, addr-len, addr, len,
                   mmu_codes[(prot&PROTMASK)>>8]);
        }
    }
}
#endif DUMPMM
/*
* check the size of a process
*/
chksize(nt, nd, ns)
register unsigned nt, nd, ns;
{
    if (nt + nd + ns + v.v_usize < maxmem)
        return(0);
    u.u_error = ENOMEM;
    return(1);
}

```

```
/*
 * generalized seek sort for disk
 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/utsname.h"
#include "sys/buf.h"
#include "sys/eelog.h"
#include "sys/erec.h"
#include "sys/iobuf.h"

#define b_cylin b_resid

disksort(dp, bp)
register struct iobuf *dp;
register struct buf *bp;
{
    register struct buf *ap;
    struct buf *tp;

    ap = dp->b_actf;
    if(ap == NULL) {
        dp->b_actf = bp;
        dp->b_actl = bp;
        bp->av_forw = NULL;
        return;
    }
    tp = NULL;
    for(; ap != NULL; ap = ap->av_forw) {
        if ((bp->b_flags&B_READ) && (ap->b_flags&B_READ) == 0) {
            if (tp == NULL)
                tp = ap;
            break;
        }
        if ((bp->b_flags&B_READ) == 0 && (ap->b_flags&B_READ))
            continue;
        if(ap->b_cylin <= bp->b_cylin)
            if(tp == NULL || ap->b_cylin >= tp->b_cylin)
                tp = ap;
    }
    if(tp == NULL)
        tp = dp->b_actl;
    bp->av_forw = tp->av_forw;
    tp->av_forw = bp;
    if(tp == dp->b_actl)
        dp->b_actl = bp;
}
```

```
/* @(#)err.c 1.1 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/buf.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/file.h"
#include "sys/utname.h"
#include "sys/eelog.h"
#include "sys/erec.h"

static short logging;

erropen(dev, flg)
{
    if(logging) {
        u.u_error = EBUSY;
        return;
    }
    if((flg&FWRITE) || dev != 0) {
        u.u_error = ENXIO;
        return;
    }
    if(suser()) {
        logstart();
        logging++;
    }
}

/* ARGSUSED */
errclose(dev, flg)
{
    logging = 0;
}

/* ARGSUSED */
errread(dev)
{
    register struct errhdr *eup;
    register n;
    struct errhdr *geterec();

    if(logging == 0)
        return;
    eup = geterec();
    n = min((unsigned)eup->e_len, u.u_count);
    if (copyout((caddr_t)eup, u.u_base, n))
        u.u_error = EFAULT;
    else
        u.u_count -= n;
    freeslot(eup);
}
```



```

#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/buf.h"
#include "sys/conf.h"
#include "sys/map.h"
#include "sys/utsname.h"
#include "sys/elog.h"
#include "sys/erec.h"
#include "sys/err.h"
#include "sys/iobuf.h"
#include "sys/var.h"

typedef int mem_t;

int blkacty;

errinit()
{
    register struct err *errp;

    errp = &err;
    if(errp->e_nslot) {
        mapinit(errp->e_map, (errp->e_nslot+3)/2);
        mfree(errp->e_map, (mem_t)errp->e_nslot, (mem_t)1);
    }
    errp->e_org = errp->e_ptrs;
    errp->e_nxt = errp->e_ptrs;
}

struct errhdr *
geteslot(size)
{
    register ns, *p;
    register struct errhdr *ep;
    int n, sps;

    ns = (size+sizeof(struct errhdr)+sizeof(struct errslist)-1)
        /sizeof(struct errslist);
    sps = spl7();
    n = malloc(err.e_map, (mem_t)ns);
    splx(sps);
    if(n == 0)
        return(NULL);
    ep = (struct errhdr *)(&err.e_slot[--n]);
    ns *= sizeof(struct errslist)/sizeof(int);
    p = (int *)ep;
    do {
        *p++ = 0;
    } while(--ns);
    ep->e_len = size + sizeof(struct errhdr);
    return(++ep);
}

freeslot(ep)
register struct errhdr *ep;
{
    register ns, sps;

    ns = (ep->e_len+sizeof(struct errslist)-1)/sizeof(struct errslist);
    sps = spl7();
    mfree(err.e_map, (mem_t)ns,
        (mem_t)(((struct errslist *)ep)-err.e_slot+1));
    splx(sps);
}

struct errhdr *
geterec()
{
    register sps;
    register struct errhdr *ep;
    register struct err *errp;

    errp = &err;
    sps = spl7();
    while(*errp->e_org == NULL)
        (void) sleep((caddr_t)&errp->e_org,PZERO+1);
    ep = *errp->e_org;
    *errp->e_org++ = NULL;
    if(errp->e_org >= &errp->e_ptrs[errp->e_nslot])
        errp->e_org = errp->e_ptrs;
    splx(sps);
    return(ep);
}

puterec(ep, type)
register struct errhdr *ep;
{
    register sps;
    register struct err *errp;

    errp = &err;
    (--ep)->e_type = type;
    ep->e_time = time;
    sps = spl7();
    *errp->e_nxt++ = ep;
    if(errp->e_nxt >= &errp->e_ptrs[errp->e_nslot])
        errp->e_nxt = errp->e_ptrs;
    splx(sps);
    wakeup((caddr_t)&errp->e_org);
}

logstart()
{
    register sps;
    register struct estart *ep;
    register struct bdevsw *bdp;
    register struct err *errp;
    extern nodev();

    errp = &err;

    sps = spl7();
    for(errp->e_org = &errp->e_ptrs[errp->e_nslot-1];
        errp->e_org >= errp->e_ptrs; errp->e_org--)
        if(*errp->e_org != NULL) {
            freeslot(*errp->e_org);
            *errp->e_org = NULL;
        }
    errp->e_org = errp->e_ptrs;
    errp->e_nxt = errp->e_ptrs;
    ep = (struct estart *)geteslot(sizeof(struct estart));
    splx(sps);
    if(ep == NULL)
        return;
    ep->e_name = utsname;
    for(bdp = &bdevsw[bdevcnt-1]; bdp >= bdevsw; bdp--)
        if(bdp->d_strategy != nodev)
            ep->e_bconf |= 1 << (&bdevsw[0]-bdp);
    ep->e_bconf = blkacty;
    puterec((struct errhdr *)ep, E_GOTS);
}

logtchg(nt)
time_t nt;
{
    register struct etimchg *ep;

    if((ep = (struct etimchg *)geteslot(sizeof(struct etimchg))) != NULL) {
        ep->e_ntime = nt;
        puterec((struct errhdr *)ep, E_TCHG);
    }
}

logstray(addr)
physadr addr;
{
    register struct estray *ep;

    if((ep = (struct estray *)geteslot(sizeof(struct estray))) != NULL) {
        ep->e_saddr = addr;
    }
}

```

```

        ep->e_sbacty = blkacty;
        puterec((struct errhdr *)ep,E_STRAY);
    }
}

logparity(addr)
register paddr_t addr;
{
    register struct eparity *ep;

    if((ep = (struct eparity *)geteslot(sizeof(struct eparity))) != NULL) {
        ep->e_parreg = addr;
        puterec((struct errhdr *)ep,E_PRTY);
    }
}

/*
 * fmberr() is used by block device drivers to build up a valid
 * eblock structure to be sent to the error log.
 *
 * dp          the address of the io queue item.
 * unit        the Physical Device error report field
 *             the Logical Device field is the minor device number
 * cyl         the cylinder number
 * trk         the track number
 * sector      the sector number
 * regcnt      the number of following register structures
 * regs        is the address of an array of structures each of which
 *             contain the elements described in struct deverreg.
 */

/* VARARGS7 */
fmberr(dp, unit, cyl, trk, sector, regcnt, regs)
register struct iobuf *dp;
unsigned unit;
unsigned cyl;
unsigned trk;
unsigned sector;
long regcnt;
struct deverreg *regs;
{
    register struct eblock *ep;
    register struct buf *bp;
    register struct deverreg **dr;
    register char *str1;
    register char *pp;
    register short argc;
    register short nn;
    struct br { /* just used to generate addr after eblock */
        struct eblock eb;
        char cregs[1];
    };
    struct iostat *iosp;
    extern char *longcopy();

    if(dp->io_erec != NULL) {
        dp->io_erec->e_rtry++;
        return;
    }

    /* count the length of the values and strings */
    nn = 0;
    argc = regcnt;
    dr = &regs;
    while (argc-- > 0) {
        nn += sizeof((*dr)->draddr);
        nn += sizeof((*dr)->drvalue);
        nn += strlen((*dr)->drname) + 1; /* + null */
        nn += strlen((*dr)->drbits) + 1; /* + null */
        nn += (nn & 1); /* round to even number of bytes */
        dr++;
    }
    isop = dp->io_stp;
    /* want sizeof eblock to the next long address */
    if((ep = (struct eblock *)
        geteslot(sizeof(struct eblock) + nn)) == NULL) {
        isop->io_unlog++;
        return;
    }
    nn = major(dp->b_dev);
    bp = dp->b_actf;
    ep->e_dev = makedev(nn, (bp==NULL)?minor(dp->b_dev):minor(bp->b_dev));
    ep->e_bacty = blkacty;
    ep->e_stats.io_ops = isop->io_ops;
    ep->e_stats.io_misc = isop->io_misc;
    ep->e_stats.io_unlog = isop->io_unlog;
    ep->e_pos.unit = unit;
    ep->e_pos.cyl = cyl;
    ep->e_pos.trk = trk;
    ep->e_pos.sector = sector;
    if(bp != NULL) {
        ep->e_bflags = (bp->b_flags & B_READ) ? E_READ : E_WRITE;
        if(bp->b_flags & B_PHYS)
            ep->e_bflags |= E_PHYS;
        if(bp->b_flags & B_MAP)
            ep->e_bflags |= E_MAP;
        ep->e_bnum = bp->b_blkno;
        ep->e_bytes = bp->b_bcount;
        ep->e_memadd = paddr(bp);
    }
    else
        ep->e_bflags = E_NOIO;
    ep->e_nreg = regcnt;
    pp = &{((struct br *)ep)->cregs[0]};
    dr = &regs;
    while(--regcnt >= 0) {
        /* copy out the number values */
        pp = longcopy((char *)&((*dr)->draddr),pp);
        pp = longcopy((char *)&((*dr)->drvalue),pp);
        /* copy out the strings themselves */
        str1 = (*dr)->drname;
        while (*str1) {
            *pp++ = *str1++;
        }
        /* copy the terminating null too */
        *pp++ = '\0';
        str1 = (*dr)->drbits;
        while (*str1) {
            *pp++ = *str1++;
        }
        *pp++ = '\0';
        dr++;
    }
    dp->io_erec = ep;
}

logberr(dp,error)
register struct iobuf *dp;
{
    register struct eblock *ep;

    if((ep = dp->io_erec) == NULL)
        return;
    if(error)
        ep->e_bflags |= E_ERROR;
    puterec((struct errhdr *)ep,E_BLK);
    dp->io_erec = NULL;
}

/* may not be on long address boundary when copied to b2,
   avoiding any alignment problems on some machines? */
char *
longcopy(b1,b2)
register char *b1,*b2;
{
    register int ii;

    for (ii=0; ii < sizeof(long); ii++) {
        *b2++ = *b1++;
    }
    return(b2);
}

```

```

/* @(#)fio.c 1.4 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/filsys.h"
#include "sys/file.h"
#include "sys/conf.h"
#include "sys/inode.h"
#include "sys/mount.h"
#include "sys/var.h"
#include "sys/acct.h"
#include "sys/sysinfo.h"
#ifdef UCB_NET
#include "net/misc.h"
#include "net/socketvar.h"
#endif

/*
 * Convert a user supplied file descriptor into a pointer
 * to a file structure.
 * Only task is to check range of the descriptor.
 */
struct file *
getf(f)
register int f;
{
    register struct file *fp;

    if (0 <= f && f < NOFILE) {
        fp = u.u_ofile[f];
        if (fp != NULL)
#ifdef UCB_NET
            /* Mainly for net reset */
            if (fp->f_count == 0) {
                u.u_error = ENETDOWN;
                return (NULL);
            }
            else
#endif
                return(fp);
        }
        u.u_error = EBADF;
        return(NULL);
    }
}

/*
 * Internal form of close.
 * Decrement reference count on file structure.
 * Also make sure the pipe protocol does not constipate.
 *
 * Decrement reference count on the inode following
 * removal to the referencing file structure.
 * On the last close switch out to the device handler for
 * special files. Note that the handler is called
 * on every open but only the last close.
 */
closef(fp)
register struct file *fp;
{
    register struct inode *ip;
    int flag, fmt;
    dev_t dev;
    register int (*cfunc)();

    if (fp == NULL)
        return;
    flag = fp->f_flag;
#ifdef UCB_NET
    if ((flag & FSOCKET) == 0)
#endif
        unlock(fp->f_inode); /* file locking hook */

    if ((unsigned)fp->f_count > 1) {
#ifdef UCB_NET
        fp->f_flag &= ~FISUSER;
#endif
        fp->f_count--;
        return;
    }
#ifdef UCB_NET
    if (flag & FSOCKET) {
        int nouser = ((flag & FISUSER) == 0);

        u.u_error = 0; /* XXX */
        fp->f_flag &= ~FISUSER;
        soclose((struct socket *)fp->f_socket, nouser);
        if (nouser == 0 && u.u_error)
            return;
        fp->f_socket = 0;
        fp->f_count = 0;
        fp->f_next = ffreelist;
        ffreelist = fp;
        /*
         * the next line was in the ll code. Dont quite understand it,
         * but it cant hurt... (billn)
         */
        u.u_error = 0; /* so u.u_ofile always gets 0'd */
        return;
    }
#endif
    ip = fp->f_inode;
    plock(ip);
    dev = (dev_t)ip->i_rdev;
    fmt = ip->i_mode&IFMT;
    fp->f_count = 0;
    fp->f_next = ffreelist;
    ffreelist = fp;
    switch(fmt) {

    case IFCHR:
        cfunc = cdevsw[(short)major(dev)].d_close;
        break;

    case IFBLK:
        cfunc = bdevsw[bmajor(dev)].d_close;
        break;

    case IFIFO:
        closep(ip, flag);

    default:
        iput(ip);
        return;
    }
    for (fp = file; fp < (struct file *)v.va_file; fp++) {
        register struct inode *tip;

#ifdef UCB_NET
        if (fp->f_flag & FSOCKET)
            continue;
#endif
        if (fp->f_count) {
            tip = fp->f_inode;
            if (tip->i_rdev == dev &&
                (tip->i_mode&IFMT) == fmt)
                goto out;
        }
    }
    if (fmt == IFBLK) {
        register struct mount *mp;

        for (mp = mount; mp < (struct mount *)v.va_mount; mp++)
            if (mp->m_flags == MINUSE && mp->m_dev == dev)
                goto out;
        bflush(dev);
        (*cfunc)(minor(dev), flag);
        binval(dev);
    } else {

```

```

        prele(ip);
        (*cfunc)(minor(dev), flag);
    }
out:
    iput(ip);
}

/*
 * openi called to allow handler of special files to initialize and
 * validate before actual IO.
 */
openi(ip, flag)
register struct inode *ip;
{
    dev_t dev;
    register unsigned int maj;

    dev = (dev_t)ip->i_rdev;
    switch(ip->i_mode&IFMT) {

    case IFCHR:
        maj = major(dev);
        if (maj >= cdevcnt)
            goto bad;
        if (u.u_ttyp == NULL)
            u.u_ttyd = dev;
        (*cdevsw[(short)maj].d_open)(minor(dev), flag);
        break;

    case IFBLK:
        maj = bmajor(dev);
        if (maj >= bdevcnt)
            goto bad;
        (*bdevsw[maj].d_open)(minor(dev), flag);
        break;

    case IFIFO:
        openp(ip, flag);
        break;
    }
    return;

bad:
    u.u_error = ENXIO;
}

/*
 * Check mode permission on inode pointer.
 * Mode is READ, WRITE or EXEC.
 * In the case of WRITE, the read-only status of the file
 * system is checked. Also in WRITE, prototype text
 * segments cannot be written.
 * The mode is shifted to select the owner/group/other fields.
 * The super user is granted all permissions.
 */
access(ip, mode)
register struct inode *ip;
{
    register struct user *up;
    register m;

    up = &u;
    m = mode;
    if (m == IWRITE) {
        if (getfs(ip->i_dev)->s_ronly) {
            up->u_error = EROFS;
            return(1);
        }
        if (ip->i_flag&ITEXT)
            xrele(ip);
        if (ip->i_flag & ITEXT) {
            up->u_error = ETXTBSY;
            return(1);
        }
    }
    if (up->u_uid == 0)

```

```

        return(0);
    if (up->u_uid != ip->i_uid) {
        m >>= 3;
        if (up->u_gid != ip->i_gid)
            m >>= 3;
    }
    if ((ip->i_mode&m) != 0)
        return(0);

    up->u_error = EACCES;
    return(1);
}

/*
 * Look up a pathname and test if the resultant inode is owned by the
 * current user. If not, try for super-user.
 * If permission is granted, return inode pointer.
 */
struct inode *
owner()
{
    register struct inode *ip;

    ip = namei(uchar, 0);
    if (ip == NULL)
        return(NULL);
    if (u.u_uid == ip->i_uid || suser())
        if (getfs(ip->i_dev)->s_ronly)
            u.u_error = EROFS;
    if (!u.u_error)
        return(ip);
    iput(ip);
    return(NULL);
}

/*
 * Test if the current user is the super user.
 */
suser()
{
    if (u.u_uid == 0) {
        u.u_acflag |= ASU;
        return(1);
    }
    u.u_error = EPERM;
    return(0);
}

/*
 * Allocate a user file descriptor.
 */
ufalloc(i)
register i;
{
    register struct user *up;

    up = &u;
    for (; i < NOFILE; i++)
        if (up->u_ofile[i] == NULL) {
            up->u_rvall = i;
            up->u_pofile[i] = 0;
            return(i);
        }
    up->u_error = EMFILE;
    return(-1);
}

/*
 * Allocate a user file descriptor and a file structure.
 * Initialize the descriptor to point at the file structure.
 * no file -- if there are no available file structures.
 */
struct file *
falloc(ip, flag)

```

```
struct inode *ip;
{
    register struct file *fp;
    register i;

    i = ufalloc(0);
    if (i < 0)
        return(NULL);
    if ((fp->ffreelist) == NULL) {
        printf("no file\n");
        syserr.fileovf++;
        u.u_error = ENFILE;
        return(NULL);
    }
    ffreelist = fp->f_next;
    u.u_ofile[i] = fp;
    fp->f_count++;
    fp->f_inode = ip;
    fp->f_flag = flag;
    fp->f_offset = 0;
    return(fp);
}

struct file *ffreelist;
finit()
{
    register struct file *fp;
    register short i;

    ffreelist = fp = &file[0];
    i = v.v_file - 1 - 1;
    do {
        fp->f_next = fp+1;
        fp++;
    } while (--i != -1);
}
```

```
/* data and code stubs for loading without network */
```

```
#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/mmu.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/proc.h"
#include "sys/seg.h"
#include "sys/signal.h"
#include "sys/errno.h"
#include "sys/user.h"
#include "sys/system.h"
#include "sys/inode.h"
#include "sys/ino.h"
#include "sys/file.h"
#include "sys/conf.h"
#include "net/misc.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "sys/stat.h"
#include "sys/ioctl.h"
#include "net/ubavar.h"
#include "sys/map.h"
#include "net/if.h"
#include "net/in.h"
#include "net/in_system.h"
#include "net/ip.h"
#include "net/ip_var.h"
#include "sys/var.h"
```

```
/* data */
```

```
struct protosw protosw[1];
char netstak[3000];
char * svstak;
int ifnetslow;
int protofast;
int protoslow;
short netoff;
int netisr;
extern int ptc_dev;
extern int selwait;
extern u_short ip_id;
extern struct ipq ipq;
extern struct ipstat ipstat;
extern struct ifqueue rawintrl;
extern struct uba_device ubdinit[];
extern struct protosw *protoswLAST;
#ifdef INET
extern struct ifqueue ipintrl;
#endif
```

```
/* routines */
```

```
ssocket()
{
#ifdef lint
    ifnet++;
    ifnetslow++;
    protofast++;
    protoslow++;
    ptc_dev++;
    selwait++;
    ip_id++;
    ipq.ipq_ttl++;
    *protoswLAST++;
    ubdinit[0].ui_unit++;
    ipintrl.ifq_len++;
    rawintrl.ifq_len++;
    protosw[0].pr_type++;
    ipstat.ips_toosmall++;
#endif
    u.u_error = ENETDOWN;
}
netintr()
```

```
{
    netisr = 0;
}
/*ARGSUSED*/
soclose(so, exiting) struct socket *so; int exiting;
{
    u.u_error = ENETDOWN;
}
/*ARGSUSED*/
soreceive(so, asa) struct socket *so; struct sockaddr *asa;
{
    return(0);
}
/*ARGSUSED*/
sosend(so, asa) struct socket *so; struct sockaddr *asa;
{
    return(0);
}
/*ARGSUSED*/
sostat(so, sb) struct socket *so; struct stat *sb;
{
    return(0);
}
/*ARGSUSED*/
soioctl(so, cmd, cmdp) struct socket *so; int cmd; caddr_t cmdp;
{
    u.u_error = ENETDOWN;
}
sconnect()
{
    u.u_error = ENETDOWN;
}
ssend()
{
    u.u_error = ENETDOWN;
}
ssockad()
{
    u.u_error = ENETDOWN;
}
saccept()
{
    u.u_error = ENETDOWN;
}
netreset()
{
}
sethostname()
{
    u.u_error = ENETDOWN;
}
gethostname()
{
    u.u_error = ENETDOWN;
}
select()
{
    u.u_error = ENETDOWN;
}
sreceive()
{
    u.u_error = ENETDOWN;
}
/*ARGSUSED*/
ptswrite(dev) dev_t dev;
{
    u.u_error = ENETDOWN;
}
/*ARGSUSED*/
ptcioc1(dev, cmd, addr, flag) caddr_t addr; dev_t dev;
{
```

```
        u.u_error = ENETDOWN;
}
/*ARGSUSED*/
ptciocctl(dev, cmd, addr, flag) caddr_t addr; dev_t dev;
{
    u.u_error = ENETDOWN;
}
/*ARGSUSED*/
ptsopen(dev, flag) dev_t dev;
{
    u.u_error = ENETDOWN;
}
/*ARGSUSED*/
ptcopen(dev, flag) dev_t dev; int flag;
{
    u.u_error = ENETDOWN;
}
/*ARGSUSED*/
ptsread(dev) dev_t dev;
{
    u.u_error = ENETDOWN;
}
/*ARGSUSED*/
ptcread(dev) dev_t dev;
{
    u.u_error = ENETDOWN;
}
/*ARGSUSED*/
ptsclose(dev) dev_t dev;
{
    u.u_error = ENETDOWN;
}
/*ARGSUSED*/
ptcclose(dev) dev_t dev;
{
    u.u_error = ENETDOWN;
}
netinit()
{
}

#ifdef _NOTDEF
/* reference fnetlocal.c for inserting null interrupt associated stuff */
FAKE()
{
    fnetlocal();
}
#endif _NOTDEF
```

```
/* local defs for no network */

#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/var.h"
#include "errno.h"

#include "net/misc.h"
#include "net/mbuf.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "net/ubavar.h"

/* causes make to complain so this module gets included. "Called" from fnet.c */
fnetlocal(){}

/* ..driver structure */
struct uba_driver ebdriver;

/* interrupt routine */
ebintr(){}
```



```

/* if.c 4.25 83/02/10 */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/system.h"
#include "net/misc.h"
#include "net/socket.h"
#include "net/mbuf.h"
#include "net/protosw.h"
#include "net/if.h"
#include "net/af.h"

int ifqmaxlen = IFQ_MAXLEN;

/*
 * Network interface utility routines.
 *
 * Routines with if_ifwith* names take sockaddr **s as
 * parameters. Other routines take value parameters,
 * e.g. if_ifwithnet takes the network number.
 */

ifinit()
{
    register struct ifnet *ifp;

    for (ifp = ifnet; ifp; ifp = ifp->if_next)
        if (ifp->if_init) {
            (*ifp->if_init)(ifp->if_unit);
            if (ifp->if_snd.ifq_maxlen == 0)
                ifp->if_snd.ifq_maxlen = ifqmaxlen;
        }
    if_slowtimo();
}

#if vax
/*
 * Call each interface on a Unibus reset.
 */
ifubareset(uband)
{
    int uband;

    register struct ifnet *ifp;

    for (ifp = ifnet; ifp; ifp = ifp->if_next)
        if (ifp->if_reset)
            (*ifp->if_reset)(uband);
}
#endif

/*
 * Attach an interface to the
 * list of "active" interfaces.
 */
if_attach(ifp)
{
    struct ifnet *ifp;

    register struct ifnet **p = &ifnet;

    while (*p)
        p = &((*p)->if_next);
    *p = ifp;
}

/*
 * Locate an interface based on a complete address.
 */
/*ARGSUSED*/
struct ifnet *
if_ifwithaddr(addr)
{
    struct sockaddr *addr;

    register struct ifnet *ifp;

#define equal(a1, a2) \
    (bcmp((caddr_t)((a1)->sa_data), (caddr_t)((a2)->sa_data), 14) == 0)
    for (ifp = ifnet; ifp; ifp = ifp->if_next) {
        if (ifp->if_addr.sa_family != addr->sa_family)
            continue;
        if (equal(&ifp->if_addr, addr))
            break;
        if ((ifp->if_flags & IFF_BROADCAST) &&
            equal(&ifp->if_broadaddr, addr))
            break;
    }
    return (ifp);
}

/*
 * Find an interface on a specific network. If many, choice
 * is first found.
 */
struct ifnet *
if_ifwithnet(addr)
{
    register struct sockaddr *addr;

    register struct ifnet *ifp;
    register u_int af = addr->sa_family;
    register int (*netmatch)();

    if (af >= AF_MAX)
        return (0);
    netmatch = afswitch[af].af_netmatch;
    for (ifp = ifnet; ifp; ifp = ifp->if_next) {
        if (af != ifp->if_addr.sa_family)
            continue;
        if ((*netmatch)(addr, &ifp->if_addr))
            break;
    }
    return (ifp);
}

/*
 * As above, but parameter is network number.
 */
struct ifnet *
if_ifonnetof(net)
{
    register int net;

    register struct ifnet *ifp;

    for (ifp = ifnet; ifp; ifp = ifp->if_next)
        if (ifp->if_net == net)
            break;

    return (ifp);
}

/*
 * Find an interface using a specific address family
 */
struct ifnet *
if_ifwithaf(af)
{
    register int af;

    register struct ifnet *ifp;

    for (ifp = ifnet; ifp; ifp = ifp->if_next)
        if (ifp->if_addr.sa_family == af)
            break;

    return (ifp);
}

#endif notdef
/*
 * Mark an interface down and notify protocols of
 * the transition.
 * NOTE: must be called at splnet or equivalent.
 */
if_down(ifp)
    register struct ifnet *ifp;
{

```

```
    ifp->if_flags &= ~IFF_UP;
    pfcrtlinput(PRC_IFDOWN, (caddr_t)&ifp->if_addr);
}
#endif

/*
 * Handle interface watchdog timer routines. Called
 * from softclock, we decrement timers (if set) and
 * call the appropriate interface routine on expiration.
 */
if_slowtimo()
{
    register struct ifnet *ifp;

    for (ifp = ifnet; ifp; ifp = ifp->if_next) {
        if (ifp->if_timer == 0 || --ifp->if_timer)
            continue;
        if (ifp->if_watchdog)
            (*ifp->if_watchdog)(ifp->if_unit);
    }
    /* billn -- clock calls us in old...
    timeout(if_slowtimo, (caddr_t)0, hz / IFNET_SLOWHZ);
    */
}
```

```

/* 3com etherbox driver */

#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/var.h"
#include "errno.h"
#include "sys/config.h"

#include "net/misc.h"
#include "net/mbuf.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "net/ubavar.h"
#include "net/if.h"
#include "net/route.h"
#include "net/in.h"
#include "net/in_sysm.h"
#include "net/ip.h"
#include "net/ip_var.h"
#include "net/if_ether.h"

#define EBPUP_PUPTYPE 0x0400 /* PUP protocol */
#define EBPUP_IPTYPE 0x0800 /* IP protocol */

/*
 * The EBPUP_NTRAILER packet types starting at EBPUP_TRAIL have
 * (type-EBPUP_TRAIL)*512 bytes of data followed
 * by a PUP type (as given above) and then the (variable-length) header.
 */
#define EBPUP_TRAIL 0x1000 /* Trailer PUP */
#define EBPUP_NTRAILER 16

/*
 * 3Com Ethernet controller registers.
 */
#define EB_ACTADDR0 0 /* actual address byte 0 */
#define EB_ACTADDR1 1 /* actual address byte 0 */
#define EB_ACTADDR2 2 /* actual address byte 0 */
#define EB_ACTADDR3 3 /* actual address byte 0 */
#define EB_ACTADDR4 4 /* actual address byte 0 */
#define EB_ACTADDR5 5 /* actual address byte 0 */
#define EB_RCVCMD 6 /* receive command */
#define EB_XCSR 7 /* transmit csr */
#define EB_XBP_HI 8 /* transmit buffer pointer high byte */
#define EB_XBP_LO 9 /* transmit buffer pointer low byte */
#define EB_BBPCLEAR 10 /* buffer pointer clear(w) */
#define EB_PROM 10 /* address prom (r) */
#define EB_AUXCSR 11 /* auxiliary command/status */
#define EB_COLLNTR 12 /* collision counter */
#define EB_XMTBUF 13 /* transmit buffer */
#define EB_RCVBUFA 14 /* receive buffer a */
#define EB_RCVBUFB 15 /* receive buffer b */

/*
 * Transmit cmd reg bits
 */
#define EB_EIEOF 0x8 /* enable ints on end of frame */
#define EB_EI16COLL 4 /* enable ints on 16 collisions */
#define EB_EICOLL 2 /* enable ints on collisions */
#define EB_EIUNDER 1 /* enable ints on underflow */

/*
 * Transmit status reg bits
 */
#define EB_XREADY 0x8 /* ready for new frame */
#define EB_COLL16 4 /* 16 collisions detected on last xmit */
#define EB_COLL 2 /* collision occurred */
#define EB_UNDERFLOW 1 /* force underflow */

/*
 * Receive command reg bits
 */
#define EB_MULT1 0xC /* match station, multi, broadcast */

#define EB_STABROAD 0x80 /* match station, broadcast */
#define EB_PROMIS 0x40 /* match all packets */
#define EB_ANYGOOD 0x20 /* enable detection of any good frame */
#define EB_ANY 0x10 /* enable detection of any frame */
#define EB_SHORT 0x8 /* enable detection of short frames */
#define EB_DRIBBLE 4 /* enable detection of dribble error */
#define EB_CRC 2 /* enable detection of CRC error */
#define EB_OVFLO 1 /* enable detection of overflow error */
#define EB_RCVNORM (EB_STABROAD|EB_ANYGOOD|EB_DRIBBLE|EB_CRC|EB_OVFLO|EB_SHORT)

/*
 * Receive status reg bits
 */
#define EB_STALE 0x80 /* invalid packet here */
#define EB_SHORTERR 0x40 /* short frame */
#define EB_DRIBERR 0x20 /* dribble error */
#define EB_CRCERR 0x10 /* crc error */
#define EB_OVFLERR 0x8 /* overflow error */
#define EB_RCVERR 0xf8 /* error mask */
#define EB_RBUF 0x7 /* buffer pointer mask */

/*
 * Auxiliary command reg bits
 */
#define EB_EDLCRES 0x80 /* reset EDLC chip */
#define EB_SYSEI 0x40 /* enable system interrupts */
#define EB_RBBSW 0x20 /* talk to receive buffer b */
#define EB_RBASW 0x10 /* talk to receive buffer a */
#define EB_XBUFSW 0x8 /* talk to transmit buffer */
#define EB_XEOPDIS 2 /* create underflow for testing */
#define EB_POWINTR 1 /* clear power-on interrupt */

/*
 * Auxiliary status reg bits
 */
#define EB_XCVRUP 0x80 /* the integral transceiver is enabled */
#define EB_BBASW 4 /* receive buf b before a switch */
#define EB_IMASK 0x70 /* interesting bits for rcv. int. routine */

#define EBRDOFF 2 /* packet offset in read buffer */
#define EBMAXDOFF (2048-512) /* max packet offset (min size) */

#define NEB 1

/*
 * 3Com Ethernet Controller interface
 */
#define EBMTU 1500

int nulldev(), ebattach(), ebintr(), ebpoll();
struct uba_device *ebinfo[NEB];

struct uba_driver ebdriver = {
    nulldev, ebattach, (u_short *)0, ebinfo
};

#define EBUNIT(x) minor(x)

int ebinit(), eboutput(), ebwatch();
struct mbuf *ebget();

int ebwr_reg(), ebrd_reg();

extern struct ifnet loif;

/*
 * Ethernet software status per interface.
 *
 * Each interface is referenced by a network interface structure,
 * es_if, which the routing code uses to locate the interface.
 * This structure contains the output queue for the interface, its address, ...
 */
struct eb_softc {
    struct arpcom es_ac; /* common Ethernet structures */

```

```

#define es_if es_ac.ac_if /* network-visible interface */
#define es_enaddr es_ac.ac_enaddr /* hardware Ethernet address */
    short es_oactive; /* is output active? */
} eb_softc[NEB];

/* temp receive buffer */
char ebrbuf[EBMTU];

/* for polling */
short ebok = 0;

/* set low 3 bytes to, eg, DEC manuf. number for us to pretend to be DEC */
long eb_masq = 0;

/*
 * Interface exists: make available by filling in network interface
 * record. System will initialize the interface when it is ready
 * to accept packets.
 */
ebattach(md)
    struct uba_device *md;
{
    struct eb_softc *es = &eb_softc[md->ui_unit];
    register struct ifnet *ifp = &es->es_if;
    register pport = (int)md->ui_addr;
    struct sockaddr_in *sin;

    ifp->if_unit = md->ui_unit;
    ifp->if_name = "eb";
    ifp->if_mtu = EBMTU;
    ifp->if_net = md->ui_flags & 0xFF000000;

    if (!appleinit(pport))
        return;
    ebrset(pport);
    /*
     * Read the ethernet address from the box.
     */
    ebwr_reg(pport, EB_BBPCLER, 0); /* reset bus-buffer pointer */
    ebrd_setup(pport, EB_PROM); /* setup for read from prom */
    ebrd_data(pport, es->es_enaddr, 6);
    /* hack to change the manufacturer */
    if (eb_masq) {
        char *mp;

        mp = (char *) &eb_masq;
        mp++;
        bcopy(mp, es->es_enaddr, 3);
    }
    printf("Ethernet address = ");
    {
        char *p = &es->es_enaddr[0];
        int i, j = 0;
        char buf[14];

        for (i = 0; i < 6; i++) {
            buf[j++] = "0123456789ABCDEF"[((*p >> 4) & 0xf)];
            buf[j++] = "0123456789ABCDEF"[((*p++) & 0xf)];
        }
        buf[j++] = '\n';
        buf[j] = 0;
        printf(buf);
    }
    sin = (struct sockaddr_in *) &es->es_if.if_addr;
    sin->sin_family = AF_INET;
    /*
     * sin->sin_addr = arpmymaddr((struct arpcom *) 0);
     */

    /* this is a way to set addresses for now (without an ioctl()) */
    sin->sin_addr.s_addr = (u_long)md->ui_flags;
    ebsetaddr(ifp, sin);

    ifp->if_init = ebinit;
    ifp->if_output = eboutput;
    ifp->if_watchdog = ebwatch;

```

```

        if_attach(ifp);
    }

    ebwatch()
    {}

    /*
     * Initialization of interface; clear recorded pending
     * operations.
     */
    ebinit(unit)
        int unit;
    {
        struct eb_softc *es = &eb_softc[unit];
        register struct ifnet *ifp = &es->es_if;
        register struct sockaddr_in *sin;
        register pport;
        int i, s;
        char ebuf[6];

        sin = (struct sockaddr_in *) &ifp->if_addr;
        if (sin->sin_addr.s_addr == 0) /* address still unknown */
            return;
        if ((es->es_if.if_flags & IFF_RUNNING) == 0) {
            pport = (int) ebinf[info[unit]]->ui_addr;
            s = splimp();

            /* reset EDLC chip by toggling reset bit */
            ebwr_reg(pport, EB_AUXCSR, EB_EDLCRES);
            ebwr_reg(pport, EB_AUXCSR, 0);

            /* Initialize the address RAM */
            ebwr_reg(pport, EB_BBPCLER, 0); /* reset bus-buffer pointer */
            ebrd_setup(pport, EB_PROM);
            ebrd_data(pport, ebuf, 6);
            for (i = 0; i < 6; i++)
                ebwr_reg(pport, EB_ACTADDR0+i, ebuf[i]);

            /* Hang receive buffers and start any pending writes. */
            ebwr_reg(pport, EB_RVCMD, EB_RCVNORM); /* normal bits */
            ebwr_reg(pport, EB_AUXCSR, (EB_RBASW|EB_RBSSW|EB_SYSEI));
            es->es_oactive = 0;
            es->es_if.if_flags |= IFF_UP|IFF_RUNNING;
            if (es->es_if.if_snd.ifq_head)
                ebstart(unit);
            ebok = pport;
            ebpoll(pport);
            splx(s);
        }
        if_rtinit(&es->es_if, RTF_UP);
        arpattach(&es->es_ac);
        arpwhoas(&es->es_ac, &sin->sin_addr);
    }

    /*
     * Start or restart output on interface.
     * If interface is already active, then this is a retransmit
     * after a collision, and just restuff registers.
     * If interface is not already active, get another datagram
     * to send off of the interface queue, and map it to the interface
     * before starting the output.
     */
    ebstart(dev)
        register dev_t dev;
    {
        register int unit = EBUNIT(dev);
        register struct eb_softc *es = &eb_softc[unit];
        register int pport;
        register struct mbuf *m;
        register unsigned char csr;
        register x = spl6();

        pport = (int) ebinf[info[unit]]->ui_addr;
        if (es->es_oactive)
            goto restart;

```

```

IF_DEQUEUE(&es->es_if.if_snd, m);
if (m == 0) {
    es->es_oactive = 0;
    splx(x);
    return;
}
ebput(pport, m);
ebwr_reg(pport, EB_XCSR, (EB_EIOF|EB_EI16COLL));
csr = ebrd_reg(pport, EB_AUXCSR) & EB_IMASK;
ebwr_reg(pport, EB_AUXCSR, csr | (EB_SYSEI|EB_XBUFSW));

restart:
    splx(x);
    es->es_oactive = 1;
}

int bbuf;
int abuf;
int bbabuf;
int abbbuf;
int nopkt;
/*
 * Ethernet interface interrupt.  If received packet examine
 * packet to determine type.  If can't determine length
 * from type, then have to drop packet.  Otherwise decapsulate
 * packet based on type and pass to type specific higher-level
 * input routine.
 */
/* ARGSUSED */
ebintr(pport)
{
    register struct eb_softc *es;
    register int unit;
    register bits = 0;
    register unsigned char csr, xcscr;
    extern short netoff;

    if (netoff)
        return;
    for (unit = 0; unit < NEB; unit++) {
        es = &eb_softc[unit];
        csr = ebrd_reg(pport, EB_AUXCSR);
again:
        switch (csr & (EB_RBASW|EB_RBBSW|EB_BBASW)) {
            case EB_RBASW:
            case EB_RBASW|EB_BBASW:
                /* RBBSW == 0, receive B packet */
                bbuf++;
                ebread(es, pport, EB_RCVBUFB);
                bits |= (EB_RBBSW|EB_SYSEI);
                break;

            case EB_RBBSW:
            case EB_RBBSW|EB_BBASW:
                /* RBASW == 0, receive A packet */
                abuf++;
                ebread(es, pport, EB_RCVBUFA);
                bits |= (EB_RBASW|EB_SYSEI);
                break;

            case EB_BBASW:
                /* RBASW == 0, RBBSW == 0, BBASW, receive B, then A */
                bbabuf++;
                ebread(es, pport, EB_RCVBUFB);
                ebread(es, pport, EB_RCVBUFA);
                bits |= (EB_RBBSW|EB_RBASW|EB_SYSEI);
                break;

            case 0:
                /* RBASW == 0, RBBSW == 0, BBASW == 0, receive A, then B */
                abbbuf++;
                ebread(es, pport, EB_RCVBUFA);
                ebread(es, pport, EB_RCVBUFB);
                bits |= (EB_RBBSW|EB_RBASW|EB_SYSEI);
                break;
        }
    }
}

```

```

case EB_RBASW|EB_RBBSW:
case EB_RBASW|EB_RBBSW|EB_BBASW:
    /* no input packets */
    nopkt++;
    bits |= (EB_RBBSW|EB_RBASW|EB_SYSEI);
    break;

default:
    panic("ebintr: impossible value");
    /* NOT REACHED */
}
xcscr = ebrd_reg(pport, EB_AUXCSR);
if (xcscr != csr) {
    csr = xcscr;
    goto again;
}
ebwr_reg(pport, EB_AUXCSR, (int)((csr|bits)&EB_IMASK));

if (es->es_oactive) {
    if ((csr & EB_XBUFSW) == 0) {
        es->es_if.if_opackets++;
        es->es_oactive = 0;
        csr = ebrd_reg(pport, EB_XCSR);
        if (csr&0xff == 0xff)
            csr = ebrd_reg(pport, EB_XCSR);
        if (csr&0xff == 0xff) {
            printf("eb%d xmit status reg. botch\n",
                unit);
            goto cont;
        }
        if (csr & EB_COLL)
            /* clear counter */
            ebwr_reg(pport, EB_COLLNTR, 0);
        if (csr & EB_COLL16) {
            printf("eb%d: 16 collisions; resetting...\n",
                unit);
            ebreaset(pport);
            ebinit(unit);
        }
        if (es->es_if.if_snd.ifq_head)
            ebstart(unit);
    }
}

cont:
    ebportreset(pport);      /* clear pport */
}

ebread(es, pport, buf)
register struct eb_softc *es;
register int pport;
register int buf;
{
    register struct ether_header *eh;
    register struct mbuf *m;
    register int len, off, eboff;
    register unsigned char bytel, byte2;
    short resid;
    register struct ifqueue *inq;
    short x;

    es->es_if.if_ipackets++;
    ebwr_reg(pport, EB_BBPCLEAR, 0);      /* clear bus buffer pointer */
    bytel = ebrd_reg(pport, buf);        /* first byte of buffer */
    if (bytel & EB_RCVERR) {
        printf("ebrcv error %x\n", bytel&0xff);
        es->es_if.if_ierrors++;
        return;
    }
    byte2 = ebrd_reg(pport, buf);        /* second byte of buffer */
    eboff = (((bytel&EB_RBUF)<<8)|byte2) - EBRDOFF;

    /* get data */
    ebrd_setup(pport, buf);
    ebrd_data(pport, ebrbuf, eboff);
}

```

```

/*
 * Get input data length.
 * Get pointer to ethernet header (in input buffer).
 * Deal with trailer protocol: if type is PUP trailer
 * get true type from first 16-bit word past data.
 * Remember that type was trailer by setting off.
 */
len = eboff - sizeof (struct ether_header);
eb = (struct ether_header *)ebrbuf;
#define ebdataaddr(eb, off, type) ((type)(((caddr_t)((eb)+1)+(off))))

if (eb->ether_type >= EBPU_PUP_TRAIL &&
    eb->ether_type < EBPU_PUP_TRAIL+EBPU_PUP_NTRAILER) {
    off = (eb->ether_type - EBPU_PUP_TRAIL) * 512;
    if (off >= EBMTU)
        return; /* sanity */
    eb->ether_type = *ebdataaddr(eb, off, u_short *);
    resid = *(ebdataaddr(eb, off+2, u_short *));
    if (off + resid > len)
        return; /* sanity */
    len = off + resid;
} else
    off = 0;
if (len == 0)
    return;

/*
 * Put packet into mbufs. Off is nonzero if packet
 * has trailing header; ebget will then force this header
 * information to be at the front, but we still have to drop
 * the type and length which are at the front of any trailer data.
 */
m = ebget(ebrbuf, len, off);
if (m == 0)
    return;
if (off) {
    m->m_off += 2 * sizeof (u_short);
    m->m_len -= 2 * sizeof (u_short);
}
x = spl6();
switch (eb->ether_type) {

#ifdef INET
case EBPU_PUP_IPTYPE:
    schednetisr(NETISR_IP);
    inq = &ipintrq;
    break;

case ETHERPUP_ARPTYPE:
    arpinput(&es->es_ac, m);
    splx(x);
    return;
#endif

default:
    m_freem(m);
    splx(x);
    return;
}

if (IF_QFULL(inq)) {
    IF_DROP(inq);
    splx(x);
    m_freem(m);
    return;
}
IF_ENQUEUE(inq, m);
splx(x);
}

/*
 * Ethernet output routine.
 * Encapsulate a packet of type family for the local net.
 * Use trailer local net encapsulation if enough data in first
 * packet leaves a multiple of 512 bytes of data in remainder.
 */
/* If destination is this address or broadcast, send packet to
 * loop device to kludge around the fact that 3com interfaces can't
 * talk to themselves.
 */
eboutput(ifp, m0, dst)
    register struct ifnet *ifp;
    register struct mbuf *m0;
    register struct sockaddr *dst;
{
    int type, s, error;
    u_char edst[6];
    struct in_addr idst;
    register struct eb_softc *es = &eb_softc[ifp->if_unit];
    register struct mbuf *m = m0;
    register struct ether_header *eb;
    register int i;
    struct mbuf *mcopy = (struct mbuf *) 0; /* Null */

    s = splnet();
    switch (dst->sa_family) {

#ifdef INET
case AF_INET:
    idst = ((struct sockaddr_in *)dst)->sin_addr;
    if (!arpresolve(&es->es_ac, m, &idst, edst))
        return (0); /* if not yet resolved */
    if (in_lnaof(idst) == INADDR_ANY)
        mcopy = m_copy(m, 0, (int)M_COPYALL);
    type = EBPU_PUP_IPTYPE;
    goto gotttype;

#endif

case AF_UNSPEC:
    eb = (struct ether_header *)dst->sa_data;
    bcopy((caddr_t)eb->ether_dhost, (caddr_t)edst, sizeof (edst));
    type = eb->ether_type;
    goto gotttype;

default:
    printf("eb%d: can't handle af%d\n", ifp->if_unit,
        dst->sa_family);
    error = EAFNOSUPPORT;
    goto bad;
}

gotttype:
/*
 * Add local net header. If no space in first mbuf,
 * allocate another.
 */
if (m->m_off > MMAXOFF ||
    MMINOFF + sizeof (struct ether_header) > m->m_off) {
    m = m_get(M_DONTWAIT);
    if (m == 0) {
        error = ENOBUFS;
        goto bad;
    }
    m->m_next = m0;
    m->m_off = MMINOFF;
    m->m_len = sizeof (struct ether_header);
} else {
    m->m_off -= sizeof (struct ether_header);
    m->m_len += sizeof (struct ether_header);
}
eb = mtod(m, struct ether_header *);
bcopy((caddr_t)edst, (caddr_t)eb->ether_dhost, sizeof (edst));
eb->ether_type = htons((u_short)type);
bcopy((caddr_t)es->es_enaddr, (caddr_t)eb->ether_shost, 6);

/*
 * Queue message on interface, and start output if interface
 * not yet active.
 */
s = splimp();
if (IF_QFULL(&ifp->if_snd)) {
    IF_DROP(&ifp->if_snd);

```

```

        error = ENOBUFS;
        goto qfull;
    }
    IF_ENQUEUE(&ifp->if_snd, m);
    if (es->es_oactive == 0)
        ebstart(ifp->if_unit);
    splx(s);

gotlocal:
    return(mcopy ? looutput(&loif, mcopy, dst) : 0);

qfull:
    m0 = m;
bad:
    m_freem(m0);
    splx(s);
    return(error);
}

/*
 * Routine to copy from mbuf chain to transmitter
 * buffer in Multibus ebmemory.
 */
ebput(pport, m)
    register int pport;
    register struct mbuf *m;
{
    register struct mbuf *mp;
    register short off;
    register flag = 0;
    register unsigned len;
    register u_char *p;

    for (off = 2048, mp = m; mp = mp->m_next)
        off -= mp->m_len;
    if (off > EBMAXTOFF) /* enforce minimum packet size */
        off = EBMAXTOFF;

    if (off & 01) {
        off--;
        flag++;
    }

    /* load xmit buffer pointer */
    ebwr_reg(pport, EB_XBP_LO, off&0xfff);
    ebwr_reg(pport, EB_XBP_HI, ((off>>8)&7));
    ebwr_setup(pport, EB_XMTBUF);
    for (mp = m; mp; mp = mp->m_next) {
        len = mp->m_len;
        if (len == 0)
            continue;
        p = mtod(mp, u_char*);
        ebwr_data(pport, p, (short)len);
    }

    if (flag)
        ebwr_data(pport, "", 1);
    /* re-load xmit buffer pointer */
    ebwr_reg(pport, EB_XBP_LO, off&0xfff);
    ebwr_reg(pport, EB_XBP_HI, ((off>>8)&7));

    m_freem(m);
}

/*
 * Routine to copy from memory into mbufs.
 */
struct mbuf *
ebget(ebrbuf, totlen, off0)
    register u_char *ebrbuf;
    register int totlen, off0;
{
    register struct mbuf *m;
    struct mbuf *top = 0, **mp = &top;
    register int off = off0, len;
    register u_char *cp;

    cp = ebrbuf + sizeof (struct ether_header);
#ifdef DUMPIN
    {char *cp = ebrbuf; int i; int j; printf("ebget:\n"); for (j=0; j<6; j++) {for (i=0; i<16; i++) printf(
#endif
        while (totlen > 0) {
            u_char *mcp;

            MGET(m, 0);
            if (m == 0) {
                goto bad;
            }
            if (off) {
                len = totlen - off;
                cp = ebrbuf + sizeof (struct ether_header) + off;
            } else
                len = totlen;
            m->m_len = len = MIN(MLEN, len);
            m->m_off = MMINOFF;
            mcp = mtod(m, u_char *);
            bcopy(cp, mcp, len);
            cp += len;
            *mp = m;
            mp = &m->m_next;
            if (off == 0) {
                totlen -= len;
                continue;
            }
            off += len;
            if (off == totlen) {
                cp = ebrbuf + sizeof (struct ether_header);
                off = 0;
                totlen = off0;
            }
        }
        return (top);
bad:
    m_freem(top);
    return (0);
}

#ifdef notdef
/* ebdelay -- wait about tim secs */
ebdelay(tim)
    register tim;
{
    register i;

    while (tim--){
        i = 100000;
        while (i--)
            ;
    }
}
#endif

/* stuff for apple */
#include <sys/pport.h>
#include <sys/cops.h>
#include <sys/d_profile.h>
#include <sys/profile.h>

#define RC      0xa8      /* read cmd */
#define WC      0xa0      /* write cmd */
#define RD      0xb8      /* read data */
#define WD      0xb0      /* write data */
#define INPUT   0x00      /* ddra input */
#define OUTPUT  0xff      /* ddra output */

/* sometime, put the parallel port data somewhere besides the in disk driver */
extern struct device_d *pro_da[];

ebreset(pport)
    register pport; /* parallel port number */
{
    register struct device_d *dp = pro_da[pport];

```

```

#define THISISMAGIC
#ifdef THISISMAGIC
    /* reset 6522. */
    dp->d_ddra = INPUT;
    dp->d_irb = 0x18;
    dp->d_ddrb = 0xbc;
    dp->d_irb = RC;
    dp->d_pcr = 0xb;
    dp->d_ier = FIRQ|FCAL;      /* enable interrupts (I guess) */
#endif THISISMAGIC

    ebwr_reg(pport, EB_AUXCSR, 1);      /* clear power-on interrupt */
    ebportreset(pport);
}

/* setup a register to write to it */
ebwr_setup(pport, regno)
int pport, regno;
{
    register struct device_d *dp = pro_da[pport];

    dp->d_irb = WC;
    dp->d_ddra = OUTPUT;
    dp->d_ira = regno;
    dp->d_irb = WD;
}

/* setup a register to read from it */
ebrd_setup(pport, regno)
int pport, regno;
{
    register struct device_d *dp = pro_da[pport];

    dp->d_irb = WC;
    dp->d_ddra = OUTPUT;
    dp->d_ira = regno;
    dp->d_irb = RD;
    dp->d_ddra = INPUT;
}

/* write data to box. must have done a ebwr_setup() first */
ebwr_data(pport, p, nbytes)
int pport;
register short nbytes;
register char *p;
{
    register struct device_d *dp = pro_da[pport];

    if (nbytes)
        do {
            dp->d_ira = *p++;
        } while (--nbytes);
}

/* read data from box. must have done a ebrd_setup() first */
ebrd_data(pport, p, nbytes)
int pport;
register short nbytes;
register char *p;
{
    register struct device_d *dp = pro_da[pport];

    if (nbytes)
        do {
            *p++ = dp->d_ira;
        } while (--nbytes);
}

/* write a register */
ebwr_reg(pport, regno, byte)
int pport, regno, byte;
{
    register struct device_d *dp = pro_da[pport];

    dp->d_irb = WC;

```

```

    dp->d_ddra = OUTPUT;
    dp->d_ira = regno;
    dp->d_irb = WD;
    dp->d_ira = byte;
}

/* read a register */
ebrd_reg(pport, regno)
int pport, regno;
{
    register struct device_d *dp = pro_da[pport];

    dp->d_irb = WC;
    dp->d_ddra = OUTPUT;
    dp->d_ira = regno;
    dp->d_irb = RD;
    dp->d_ddra = INPUT;
    return (dp->d_ira & 0xff);
}

/* ebportreset - reset port to normal state after xmit command */
ebportreset(pport)
int pport;
{
    register struct device_d *dp = pro_da[pport];

    dp->d_irb = RC;
    dp->d_ddra = INPUT;
}

appleinit(i)
{
    register struct device_d *devp;
    int ebintr();
    extern char slot[];

    if (!(PPOK(i) || (slot[PPSLOT(i)] != PR0)) /* check slot # and type */)
        printf("ethernet init: port %d, ", i);
        if (slot[PPSLOT(i)] == PM3)
            printf("Priam card\n");
        else
            printf("card ID 0x%x\n", slot[PPSLOT(i)]);
        goto fail;
    }
    devp = pro_da[i];
    if (iocheck(&devp->d_ifr)) { /* board there ? */
        { asm("nop"); }
        if (prodata[i].pd_da != devp) { /* not already setup */
            if (setppint((prodata[i].pd_da = devp), ebintr))
                goto fail;
        }
        return 1;
    } else
        } else
        printf("Can't find port for etherbox %d\n", i);
        return 0;
    }

#ifdef doneinppintr
/* applereset -- reset apple interrupt hware */
applereset(pport)
{
    register struct device_d *dp = pro_da[pport];
    dp->d_ifr = dp->d_ifr; /* reset interrupt trap */
}
#endif

/* sigh */

short polleb = 1;
/* ...as opposed to pollcat... */
ebpoll(pport)
{
    extern time_t lbolt;

    if (polleb) {

```



```
char ebuf[6];
int i;

/*
ebintr(pport);
*/
ebreset(pport);
/* reset EDLC chip by toggeling reset bit */
ebwr_reg(pport, EB_AUXCSR, EB_EDLCRES);
ebwr_reg(pport, EB_AUXCSR, 0);

/* Initialize the address RAM */
ebwr_reg(pport, EB_BBPCLEAR, 0); /* reset bus-buffer pointer */
ebrd_setup(pport, EB_PROM);
ebrd_data(pport, ebuf, 6);
for (i = 0; i < 6; i++)
    ebwr_reg(pport, EB_ACTADDR0+i, ebuf[i]);

/* Hang receive buffers and start any pending writes. */
ebwr_reg(pport, EB_RCVCMD, EB_RCVNORM); /* normal bits */
ebwr_reg(pport, EB_AUXCSR, (EB_RBASW|EB_RBBSW|EB_SYSEI));
timeout(ebpoll, pport, v.v_hz*10);
}

ebsetaddr(ifp, sin)
    register struct ifnet *ifp;
    register struct sockaddr_in *sin;
{
    ifp->if_addr = *(struct sockaddr *)sin;
    ifp->if_net = in_netof(sin->sin_addr);
    ifp->if_host[0] = in_lnaof(sin->sin_addr);
    sin = (struct sockaddr_in *)&ifp->if_broadaddr;
    sin->sin_family = AF_INET;
    sin->sin_addr = if_makeaddr(ifp->if_net, INADDR_ANY);
    ifp->if_flags |= IFF_BROADCAST;
}
```

```

/* if_ec.c 4.22 82/07/21 */

#include "ec.h"

/*
 * 3Com Ethernet Controller interface
 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/systm.h"
#include "net/mbuf.h"
#include "sys/buf.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "net/ubavar.h"
#include "net/ereg.h"
#include "net/in.h"
#include "net/in_systm.h"
#include "net/if.h"
#include "net/if_ec.h"
#include "net/if_uba.h"
#include "net/ip.h"
#include "net/ip_var.h"
#include "net/pup.h"
#include "net/route.h"
#include "errno.h"

#define ECMTU 1500
#define ECMEM 0000000

int ecprobe(), ecattach(), ecrint(), ecxint(), eccollide();
struct uba_device *ecinfo[NEC];
u_short ecstd[] = { 0 };
struct uba_driver ecdriver =
    { ecprobe, 0, ecattach, 0, ecstd, "ec", ecinfo };
u_char ec_iltop[3] = { 0x02, 0x07, 0x01 };
#define ECUNIT(x) minor(x)

int ecinit(), ecoutput(), ecreset();
struct mbuf *ecget();

extern struct ifnet loif;

/*
 * Ethernet software status per interface.
 *
 * Each interface is referenced by a network interface structure,
 * es_if, which the routing code uses to locate the interface.
 * This structure contains the output queue for the interface, its address, ...
 * We also have, for each interface, a UBA interface structure, which
 * contains information about the UNIBUS resources held by the interface:
 * map registers, buffered data paths, etc. Information is cached in this
 * structure for use by the if_uba.c routines in running the interface
 * efficiently.
 */
struct ec_softc {
    struct ifnet es_if;          /* network-visible interface */
    struct ifuba es_ifuba;      /* UNIBUS resources */
    short es_mask;              /* mask for current output delay */
    short es_oactive;           /* is output active? */
    caddr_t es_buf[16];         /* virtual addresses of buffers */
    u_char es_enaddr[6];        /* board's ethernet address */
} ec_softc[NEC];

/*
 * Do output DMA to determine interface presence and
 * interrupt vector. DMA is too short to disturb other hosts.
 */
ecprobe(reg)
    caddr_t reg;
{
    register int br, cvec;      /* r11, r10 value-result */
    register struct ecdevice *addr = (struct ecdevice *)reg;

    register caddr_t ecbuf = (caddr_t) &umem[numuba][ECMEM];

#ifdef lint
    br = 0; cvec = br; br = cvec;
    ecrint(0); ecxint(0); eccollide(0);
#endif
    /*
     * Make sure memory is turned on
     */
    addr->ec_rcr = EC_AROM;
    /*
     * Disable map registers for ec unibus space,
     * but don't allocate yet.
     */
    ubamem(numuba, ECMEM, 32*2, 0);
    /*
     * Check for existence of buffers on Unibus.
     */
    if (badaddr((caddr_t) ecbuf, 2)) {
    bad1:
        printf("ec: buffer mem not found\n");
    bad2:
        ubamem(numuba, 0, 0, 0);          /* reenable map (780 only) */
        addr->ec_rcr = EC_MDISAB;        /* disable memory */
        return (0);
    }
}
#endif VAX780
if (cpu == VAX_780 && uba_hd[numuba].uh_uba->uba_sr) {
    uba_hd[numuba].uh_uba->uba_sr = uba_hd[numuba].uh_uba->uba_sr;
    goto bad1;
}
#endif

/*
 * Tell the system that the board has memory here, so it won't
 * attempt to allocate the addresses later.
 */
if (ubamem(numuba, ECMEM, 32*2, 1) == 0) {
    printf("ecprobe: cannot reserve uba addresses\n");
    goto bad2;
}

/*
 * Make a one byte packet in what should be buffer #0.
 * Submit it for sending. This would cause an xmit interrupt.
 * The xmit interrupt vector is 8 bytes after the receive vector,
 * so adjust for this before returning.
 */
*(u_short *)ecbuf = (u_short) 03777;
ecbuf[03777] = '\0';
addr->ec_xcr = EC_XINTEN|EC_XWBN;
DELAY(100000);
addr->ec_xcr = EC_XCLR;
if (cvec > 0 && cvec != 0x200) {
    if (cvec & 04) { /* collision interrupt */
        cvec -= 04;
        br += 1; /* rcv is xmit + 1 */
    } else { /* xmit interrupt */
        cvec -= 010;
        br += 2; /* rcv is xmit + 2 */
    }
}
return (1);
}

/*
 * Interface exists: make available by filling in network interface
 * record. System will initialize the interface when it is ready
 * to accept packets.
 */
ecattach(ui)
    struct uba_device *ui;
{
    struct ec_softc *es = &ec_softc[ui->ui_unit];
    register struct ifnet *ifp = &es->es_if;
    register struct ecdevice *addr = (struct ecdevice *)ui->ui_addr;

```

```

struct sockaddr_in *sin;
int i, j;
u_char *cp;

ifp->if_unit = ui->ui_unit;
ifp->if_name = "ec";
ifp->if_mtu = ECMTU;
ifp->if_net = ui->ui_flags;

/*
 * Read the ethernet address off the board, one nibble at a time.
 */
addr->ec_xcr = EC_VECLR;
addr->ec_rcr = EC_AROM;
cp = es->es_enaddr;
#define NEXTBIT addr->ec_rcr = EC_AROM|EC_ASTEP; addr->ec_rcr = EC_AROM
for (i=0; i<6; i++) {
    *cp = 0;
    for (j=0; j<=4; j+=4) {
        *cp |= ((addr->ec_rcr >> 8) & 0xf) << j;
        NEXTBIT; NEXTBIT; NEXTBIT; NEXTBIT;
    }
    cp++;
}
#endif notdef
printf("ec%d: addr=%x:%x:%x:%x:%x:%x\n", ui->ui_unit,
    es->es_enaddr[0]&0xff, es->es_enaddr[1]&0xff,
    es->es_enaddr[2]&0xff, es->es_enaddr[3]&0xff,
    es->es_enaddr[4]&0xff, es->es_enaddr[5]&0xff);
#endif
ifp->if_host[0] = ((es->es_enaddr[3]&0xff)<<16) |
    ((es->es_enaddr[4]&0xff)<<8) | (es->es_enaddr[5]&0xff);
sin = (struct sockaddr_in *)es->es_if.if_addr;
sin->sin_family = AF_INET;
sin->sin_addr = if_makeaddr(ifp->if_net, ifp->if_host[0]);

sin = (struct sockaddr_in *)&ifp->if_broadcast;
sin->sin_family = AF_INET;
sin->sin_addr = if_makeaddr(ifp->if_net, INADDR_ANY);
ifp->if_flags = IFF_BROADCAST;

ifp->if_init = ecinit;
ifp->if_output = ecoutput;
ifp->if_ubareset = ecreset;
for (i=0; i<16; i++)
    es->es_buf[i] = &umem[ui->ui_ubanum][ECMEM+2048*i];
if_attach(ifp);

/*
 * Reset of interface after UNIBUS reset.
 * If interface is on specified uba, reset its state.
 */
ecreset(unit, uban)
    int unit, uban;
{
    register struct uba_device *ui;

    if (unit >= NEC || (ui = ecinfo[unit]) == 0 || ui->ui_alive == 0 ||
        ui->ui_ubanum != uban)
        return;
    printf(" ec%d", unit);
    ubanem(uban, ECMEM, 32*2, 0); /* map register disable (no alloc) */
    ecinit(unit);
}

/*
 * Initialization of interface; clear recorded pending
 * operations, and reinitialize UNIBUS usage.
 */
ecinit(unit)
    int unit;
{
    struct ec_softc *es = &ec_softc[unit];
    struct ecdevice *addr;
    int i, s;

    /*
     * Hang receive buffers and start any pending writes.
     * Writing into the rcr also makes sure the memory
     * is turned on.
     */
    addr = (struct ecdevice *)ecinfo[unit]->ui_addr;
    s = splmp();
    for (i=ECRHB; i>=ECRLB; i--)
        addr->ec_rcr = EC_READ|i;
    es->es_oactive = 0;
    es->es_mask = ~0;
    es->es_if.if_flags |= IFF_UP;
    if (es->es_if.if_snd.ifq_head)
        ecstart(unit);
    splx(s);
    if_rtinit(&es->es_if, RTF_UP);

    /*
     * Start or restart output on interface.
     * If interface is already active, then this is a retransmit
     * after a collision, and just restuff registers.
     * If interface is not already active, get another datagram
     * to send off of the interface queue, and map it to the interface
     * before starting the output.
     */
    ecstart(dev)
        dev_t dev;
    {
        int unit = ECUNIT(dev), dest;
        struct ec_softc *es = &ec_softc[unit];
        struct ecdevice *addr;
        struct mbuf *m;
        caddr_t ecbuf;

        if (es->es_oactive)
            goto restart;

        IF_DEQUEUE(&es->es_if_snd, m);
        if (m == 0) {
            es->es_oactive = 0;
            return;
        }
        ecput(es->es_buf[ECTBF], m);

    restart:
        addr = (struct ecdevice *)ecinfo[unit]->ui_addr;
        addr->ec_xcr = EC_WRITE|ECTBF;
        es->es_oactive = 1;
    }

    /*
     * Ethernet interface transmitter interrupt.
     * Start another output if more data to send.
     */
    ecxint(unit)
        int unit;
    {
        register struct ec_softc *es = &ec_softc[unit];
        register struct ecdevice *addr =
            (struct ecdevice *)ecinfo[unit]->ui_addr;

        if (es->es_oactive == 0)
            return;
        if ((addr->ec_xcr&EC_XDONE) == 0 || (addr->ec_xcr&EC_XBN) != ECTBF) {
            printf("ec%d: stray xmit interrupt, xcr=%b\n", unit,
                addr->ec_xcr, EC_XBITS);
            es->es_oactive = 0;
            addr->ec_xcr = EC_XCLR;
            return;
        }
        es->es_if.if_opackets++;
        es->es_oactive = 0;
        es->es_mask = ~0;
        addr->ec_xcr = EC_XCLR;
    }
}

```

```

    if (es->es_if.if_snd.ifq_head)
        ecstart(unit);
}

/*
 * Collision on ethernet interface. Do exponential
 * backoff, and retransmit. If have backed off all
 * the way print warning diagnostic, and drop packet.
 */
eccollide(unit)
    int unit;
{
    struct ec_softc *es = &ec_softc[unit];

    es->es_if.if_collisions++;
    if (es->es_oactive)
        ecdocoll(unit);
}

ecdocoll(unit)
    int unit;
{
    register struct ec_softc *es = &ec_softc[unit];
    register struct ecdevice *addr =
        (struct ecdevice *)ecinfo[unit]->ui_addr;
    register i;
    int delay;

    /*
     * Es_mask is a 16 bit number with n low zero bits, with
     * n the number of backoffs. When es_mask is 0 we have
     * backed off 16 times, and give up.
     */
    if (es->es_mask == 0) {
        es->es_if.if_oerrors++;
        printf("ec%d: send error\n", unit);
        /*
         * Reset interface, then requeue rcv buffers.
         * Some incoming packets may be lost, but that
         * can't be helped.
         */
        addr->ec_xcr = EC_UECLR;
        for (i=ECRHF; i>=ECRLBF; i--)
            addr->ec_rcr = EC_READ|i;

        /*
         * Reset and transmit next packet (if any).
         */
        es->es_oactive = 0;
        es->es_mask = ~0;
        if (es->es_if.if_snd.ifq_head)
            ecstart(unit);

        return;
    }

    /*
     * Do exponential backoff. Compute delay based on low bits
     * of the interval timer. Then delay for that number of
     * slot times. A slot time is 51.2 microseconds (rounded to 51).
     * This does not take into account the time already used to
     * process the interrupt.
     */
    es->es_mask <<= 1;
    delay = mfpr(ICR) &- es->es_mask;
    DELAY(delay * 51);

    /*
     * Clear the controller's collision flag, thus enabling retransmit.
     */
    addr->ec_xcr = EC_CLEAR;
}

/*
 * Ethernet interface receiver interrupt.
 * If input error just drop packet.
 * Otherwise purge input buffered data path and examine
 * packet to determine type. If can't determine length
 * from type, then have to drop packet. Otherwise decapsulate
 * packet based on type and pass to type specific higher-level

```

```

 * input routine.
 */
ecrint(unit)
    int unit;
{
    struct ecdevice *addr = (struct ecdevice *)ecinfo[unit]->ui_addr;

    while (addr->ec_rcr & EC_RDONE)
        ecread(unit);
}

ecread(unit)
    int unit;
{
    register struct ec_softc *es = &ec_softc[unit];
    struct ecdevice *addr = (struct ecdevice *)ecinfo[unit]->ui_addr;
    register struct ec_header *ec;
    struct mbuf *m;
    int len, off, resid, ecoff, buf;
    register struct ifqueue *inq;
    caddr_t ecbuf;

    es->es_if.if_ipackets++;
    buf = addr->ec_rcr & EC_RBN;
    if (buf < ECRLEF || buf > ECRHBF)
        panic("ecrint");
    ecbuf = es->es_buf[buf];
    ecoff = *(short *)ecbuf;
    if (ecoff <= ECRDOFF || ecoff > 2046) {
        es->es_if.if_ierrors++;
#ifdef notdef
        if (es->es_if.if_ierrors % 100 == 0)
            printf("ec%d: += 100 input errors\n", unit);
#endif
        goto setup;
    }

    /*
     * Get input data length.
     * Get pointer to ethernet header (in input buffer).
     * Deal with trailer protocol: if type is PUP trailer
     * get true type from first 16-bit word past data.
     * Remember that type was trailer by setting off.
     */
    len = ecoff - ECRDOFF - sizeof(struct ec_header);
    ec = (struct ec_header *) (ecbuf + ECRDOFF);
#define ecdataaddr(ec, off, type) ((type) (((caddr_t) ((ec)+1)+(off))))
    if (ec->ec_type >= ECPUP_TRAIL &&
        ec->ec_type < ECPUP_TRAIL+ECPUP_NTRAILER) {
        off = (ec->ec_type - ECPUP_TRAIL) * 512;
        if (off >= ECMTU)
            goto setup; /* sanity */
        ec->ec_type = *ecdataaddr(ec, off, u_short *);
        resid = *(ecdataaddr(ec, off+2, u_short *));
        if (off + resid > len)
            goto setup; /* sanity */
        len = off + resid;
    } else
        off = 0;
    if (len == 0)
        goto setup;

    /*
     * Pull packet off interface. Off is nonzero if packet
     * has trailing header; ecget will then force this header
     * information to be at the front, but we still have to drop
     * the type and length which are at the front of any trailer data.
     */
    m = ecget(ecbuf, len, off);
    if (m == 0)
        goto setup;
    if (off) {
        m->m_off += 2 * sizeof(u_short);
        m->m_len -= 2 * sizeof(u_short);
    }
    switch (ec->ec_type) {

```

```

#ifdef INET
    case ECPUP_IPTYPE:
        schednetisr(NETISR_IP);
        inq = &ipintrq;
        break;
#endif

default:
    m_freem(m);
    goto setup;
}

if (IF_QFULL(inq)) {
    IF_DROP(inq);
    m_freem(m);
    goto setup;
}
IF_ENQUEUE(inq, m);

setup:
/*
 * Reset for next packet.
 */
addr->ec_rcr = EC_READ|EC_RCLR|buf;
}

/*
 * Ethernet output routine.
 * Encapsulate a packet of type family for the local net.
 * Use trailer local net encapsulation if enough data in first
 * packet leaves a multiple of 512 bytes of data in remainder.
 * If destination is this address or broadcast, send packet to
 * loop device to kludge around the fact that 3com interfaces can't
 * talk to themselves.
 */
ecoutput(ifp, m0, dst)
    struct ifnet *ifp;
    struct mbuf *m0;
    struct sockaddr *dst;
{
    int type, dest, s, error;
    register struct ec_softc *es = &ec_softc[ifp->if_unit];
    register struct mbuf *m = m0;
    register struct ec_header *ec;
    register int off, i;
    struct mbuf *mcopy = (struct mbuf *) 0;    /* Null */

    switch (dst->sa_family) {

#ifdef INET
    case AF_INET:
        dest = ((struct sockaddr_in *)dst)->sin_addr.s_addr;
        if ((dest &~ 0xff) == 0)
            mcopy = m_copy(m, 0, M_COPYALL);
        else if (dest == ((struct sockaddr_in *)es->es_if.if_addr)->
            sin_addr.s_addr) {
            mcopy = m;
            goto gotlocal;
        }
        off = ntohs((u_short)mtod(m, struct ip *)->ip_len) - m->m_len;
        if (off > 0 && (off & 0x1fff) == 0 &&
            m->m_off >= MMINOFF + 2 * sizeof(u_short)) {
            type = ECPUP_TRAIL + (off>>9);
            m->m_off -= 2 * sizeof(u_short);
            m->m_len += 2 * sizeof(u_short);
            *mtod(m, u_short *) = ECPUP_IPTYPE;
            *(mtod(m, u_short *) + 1) = m->m_len;
            goto gottrailerstype;
        }
        type = ECPUP_IPTYPE;
        off = 0;
        goto gottype;
#endif

default:
    printf("ec%d: can't handle af%d\n", ifp->if_unit,

```

```

        dst->sa_family);
        error = EAFNOSUPPORT;
        goto bad;
    }
}

gottrailerstype:
/*
 * Packet to be sent as trailer: move first packet
 * (control information) to end of chain.
 */
while (m->m_next)
    m = m->m_next;
m->m_next = m0;
m = m0->m_next;
m0->m_next = 0;
m0 = m;

gottype:
/*
 * Add local net header.  If no space in first mbuf,
 * allocate another.
 */
if (m->m_off > MMAXOFF ||
    MMINOFF + sizeof(struct ec_header) > m->m_off) {
    m = m_get(M_DONTWAIT);
    if (m == 0) {
        error = ENOBUFS;
        goto bad;
    }
    m->m_next = m0;
    m->m_off = MMINOFF;
    m->m_len = sizeof(struct ec_header);
} else {
    m->m_off -= sizeof(struct ec_header);
    m->m_len += sizeof(struct ec_header);
}
ec = mtod(m, struct ec_header *);
for (i=0; i<6; i++)
    ec->ec_ghost[i] = es->es_enaddr[i];
if ((dest &~ 0xff) == 0)
    /* broadcast address */
    for (i=0; i<6; i++)
        ec->ec_dhost[i] = 0xff;
else {
    if (dest & 0x8000) {
        ec->ec_dhost[0] = ec_iltop[0];
        ec->ec_dhost[1] = ec_iltop[1];
        ec->ec_dhost[2] = ec_iltop[2];
    } else {
        ec->ec_dhost[0] = es->es_enaddr[0];
        ec->ec_dhost[1] = es->es_enaddr[1];
        ec->ec_dhost[2] = es->es_enaddr[2];
    }
    ec->ec_dhost[3] = (dest>>8) & 0x7f;
    ec->ec_dhost[4] = (dest>>16) & 0xff;
    ec->ec_dhost[5] = (dest>>24) & 0xff;
}
ec->ec_type = type;

/*
 * Queue message on interface, and start output if interface
 * not yet active.
 */
s = splimp();
if (IF_QFULL(&ifp->if_snd)) {
    IF_DROP(&ifp->if_snd);
    error = ENOBUFS;
    goto qfull;
}
IF_ENQUEUE(&ifp->if_snd, m);
if (es->es_oactive == 0)
    ecstart(ifp->if_unit);
splx(s);

gotlocal:
return(mcopy ? looutput(&loif, mcopy, dst) : 0);

```

```

qfull:
    m0 = m;
    splx(s);
bad:
    m_freem(m0);
    return(error);
}

/*
 * Routine to copy from mbuf chain to transmitter
 * buffer in UNIBUS memory.
 */
ecput(ecbuf, m)
    u_char *ecbuf;
    struct mbuf *m;
{
    register struct mbuf *mp;
    register int off;
    u_char *bp;

    for (off = 2048, mp = m; mp; mp = mp->m_next)
        off -= mp->m_len;
    *(u_short *)ecbuf = off;
    bp = (u_char *) (ecbuf + off);
    for (mp = m; mp; mp = mp->m_next) {
        register unsigned len = mp->m_len;
        u_char *mcp;

        if (len == 0)
            continue;
        mcp = mtod(mp, u_char *);
        if ((unsigned)bp & 01) {
            *bp++ = *mcp++;
            len--;
        }
        if (off = (len >> 1)) {
            register u_short *to, *from;

            to = (u_short *)bp;
            from = (u_short *)mcp;
            do
                *to++ = *from++;
            while (--off > 0);
            bp = (u_char *)to;
            mcp = (u_char *)from;
        }
        if (len & 01)
            *bp++ = *mcp++;
    }
}
#endif notdef
    if (bp - ecbuf != 2048)
        printf("ec: bad ecput, diff=%d\n", bp-ecbuf);
#endif
    m_freem(m);
}

/*
 * Routine to copy from UNIBUS memory into mbufs.
 * Similar in spirit to if_rubaget.
 *
 * Warning: This makes the fairly safe assumption that
 * mbufs have even lengths.
 */
ecget(ecbuf, totlen, off0)
    u_char *ecbuf;
    int totlen, off0;
{
    register struct mbuf *m;
    struct mbuf *top = 0, **mp = &top;
    register int off = off0, len;
    u_char *cp;

    cp = ecbuf + ECRDOFF + sizeof (struct ec_header);
    while (totlen > 0) {

```

```

        register int words;
        u_char *mcp;

        MGET(m, 0);
        if (m == 0)
            goto bad;
        if (off) {
            len = totlen - off;
            cp = ecbuf + ECRDOFF + sizeof (struct ec_header) + off;
        } else
            len = totlen;
        if (len >= CLBYTES) {
            struct mbuf *p;

            MCLGET(p, 1);
            if (p != 0) {
                m->m_len = len = CLBYTES;
                m->m_off = (int)p - (int)m;
            } else {
                m->m_len = len = MIN(MLEN, len);
                m->m_off = MMINOFF;
            }
        } else {
            m->m_len = len = MIN(MLEN, len);
            m->m_off = MMINOFF;
        }
        mcp = mtod(m, u_char *);
        if (words = (len >> 1)) {
            register u_short *to, *from;

            to = (u_short *)mcp;
            from = (u_short *)cp;
            do
                *to++ = *from++;
            while (--words > 0);
            mcp = (u_char *)to;
            cp = (u_char *)from;
        }
        if (len & 01)
            *mcp++ = *cp++;
        *mp = m;
        mp = &m->m_next;
        if (off == 0) {
            totlen -= len;
            continue;
        }
        off += len;
        if (off == totlen) {
            cp = ecbuf + ECRDOFF + sizeof (struct ec_header);
            off = 0;
            totlen = off0;
        }
    }
    return (top);
bad:
    m_freem(top);
    return (0);
}

```

```

/*   if_ether.c   6.2   83/08/28   */

/*
 * Ethernet address resolution protocol.
 */

#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysm.h"
#include "sys/var.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/socket.h"
#include "sys/errno.h"

#include "net/if.h"
#include "net/in.h"
#include "net/if_ether.h"

/*
 * Internet to ethernet address resolution table.
 */
struct arptab {
    struct in_addr at_iaddr;      /* internet address */
    u_char at_enaddr[6];         /* ethernet address */
    struct mbuf *at_hold;         /* last packet until resolved/timeout */
    u_char at_timer;             /* minutes since last reference */
    u_char at_flags;             /* flags */
};

/* at_flags field values */
#define ATF_INUSE 1             /* entry in use */
#define ATF_COM 2              /* completed entry (enaddr valid) */

#define ARPTAB_BSIZ 5           /* bucket size */
#define ARPTAB_NB 19           /* number of buckets */
#define ARPTAB_SIZE (ARPTAB_BSIZ * ARPTAB_NB)
struct arptab arptab[ARPTAB_SIZE];

#define ARPTAB_HASH(a) \
    ((short) (((a) >> 16) ^ (a)) & 0x7fff) % ARPTAB_NB

#define ARPTAB_LOOK(at,addr) { \
    register n; \
    at = sarptab[ARPTAB_HASH(addr) * ARPTAB_BSIZ]; \
    for (n = 0; n < ARPTAB_BSIZ; n++,at++) \
        if (at->at_iaddr.s_addr == addr) \
            break; \
    if (n >= ARPTAB_BSIZ) \
        at = 0; \
}

struct arpcom *arpcom;          /* chain of active ether interfaces */
int arpt_age;                  /* aging timer */

/* timer values */
#define ARPT_AGE (60*1)        /* aging timer, 1 min. */
#define ARPT_KILLC 20          /* kill completed entry in 20 mins. */
#define ARPT_KILLI 3          /* kill incomplete entry in 3 minutes */

u_char etherbroadcastaddr[6] = { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff };
extern struct ifnet loif;

/*
 * Local addresses in the range oldmap to infinity are
 * mapped according to the old mapping scheme. That is,
 * mapping of Internet to Ethernet addresses is performed
 * by taking the high three bytes of the network interface's
 * address and the low three bytes of the local address part.
 * This only allows boards from the same manufacturer to
 * communicate unless the on-board address is overridden
 * (not possible in many manufacture's hardware).
 *
 * NB: setting oldmap to zero completely disables ARP
 * (i.e. identical to setting IFF_NOARP with an ioctl).
 */
int oldmap = 0xffffffff;

```

```

/*
 * Attach an ethernet interface to the list "arpcom" where
 * arptimer() can find it. If first time
 * initialization, start arptimer().
 */
arpattach(ac)
{
    register struct arpcom *ac;

    register struct arpcom *acp;

    for (acp = arpcom; acp != (struct arpcom *)0; acp = acp->ac_ac)
        if (acp == ac) /* if already on list */
            return;
    ac->ac_ac = arpcom;
    arpcom = ac;
    if (arpcom->ac_ac == 0) /* very first time */
        arptimer();
}

/*
 * Timeout routine. Age arp_tab entries once a minute.
 */
arptimer()
{
    register struct arptab *at;
    register i;

    timeout(arptimer, (caddr_t)0, v.v_hz);
#ifdef notdef
    if (++arpt_sanity > ARPT_SANITY) {
        register struct arpcom *ac;

        /*
         * Randomize sanity timer based on my host address.
         * Ask who has my own address; if someone else replies,
         * then they are impersonating me.
         */
        arpt_sanity = arpcom->ac_enaddr[5] & 0x3f;
        for (ac = arpcom; ac != (struct arpcom *)-1; ac = ac->ac_ac)
            arpwhoas(ac, &((struct sockaddr_in *)
                &ac->ac_if.if_addr)->sin_addr);
    }
#endif
    if (++arpt_age > ARPT_AGE) {
        arpt_age = 0;
        at = sarptab[0];
        for (i = 0; i < ARPTAB_SIZE; i++, at++) {
            if (at->at_flags == 0)
                continue;
            if (++at->at_timer < ((at->at_flags & ATF_COM) ?
                ARPT_KILLC : ARPT_KILLI))
                continue;
            /* timer has expired, clear entry */
            arptfree(at);
        }
    }
}

/*
 * Broadcast an ARP packet, asking who has addr on interface ac.
 */
arpwhoas(ac, addr)
{
    register struct arpcom *ac;
    struct in_addr *addr;

    {
        register struct mbuf *m;
        register struct ether_header *eh;
        register struct ether_arp *ea;
        struct sockaddr sa;

        if ((m = m_get(M_DONTWAIT)) == NULL)
            return;
        m->m_len = sizeof *ea + sizeof *eh;
        m->m_off = MMAXOFF - m->m_len;
        ea = mtod(m, struct ether_arp *);
    }
}

```

```

eh = (struct ether_header *)sa.sa_data;
bzero((caddr_t)ea, sizeof (*ea));
bcopy((caddr_t)etherbroadcastaddr, (caddr_t)eh->ether_dhost,
      sizeof (etherbroadcastaddr));
eh->ether_type = ETHERPUP_ARPTYPE; /* if_output will swap */
ea->arp_hrd = htons(ARPHRD_ETHER);
ea->arp_pro = htons(ETHERPUP_IPTYPE);
ea->arp_hln = sizeof ea->arp_sha; /* hardware address length */
ea->arp_pln = sizeof ea->arp_spa; /* protocol address length */
ea->arp_op = htons(ARPOP_REQUEST);
bcopy((caddr_t)ac->ac_enaddr, (caddr_t)ea->arp_sha,
      sizeof (ea->arp_sha));
bcopy((caddr_t)((struct sockaddr_in *)&ac->ac_if.if_addr)->sin_addr,
      (caddr_t)ea->arp_spa, sizeof (ea->arp_spa));
bcopy((caddr_t)addr, (caddr_t)ea->arp_tpa, sizeof (ea->arp_tpa));
sa.sa_family = AF_UNSPEC;
(void) (*ac->ac_if.if_output)(&ac->ac_if, m, &sa);
}

/*
 * Resolve an IP address into an ethernet address. If success,
 * desten is filled in and 1 is returned. If there is no entry
 * in arptab, set one up and broadcast a request
 * for the IP address; return 0. Hold onto this mbuf and
 * resend it once the address is finally resolved.
 *
 * We do some (conservative) locking here at splimp, since
 * arptab is also altered from input interrupt service (ecintr/ilintr
 * calls arpinut when ETHERPUP_ARPTYPE packets come in).
 */
arpresolve(ac, m, destip, desten)
register struct arpcom *ac;
struct mbuf *m;
register struct in_addr *destip;
register u_char *desten;
{
    register struct arptab *at;
    register struct ifnet *ifp;
    struct sockaddr_in sin;
    int s, lna;

    lna = in_lnaof(*destip);
    if (lna == INADDR_ANY) { /* broadcast address */
        bcopy((caddr_t)etherbroadcastaddr, (caddr_t)desten,
              sizeof (etherbroadcastaddr));
        return (1);
    }
    ifp = &ac->ac_if;
    /* if for us, then use software loopback driver */
    if (destip->s_addr ==
        ((struct sockaddr_in *)&ifp->if_addr)->sin_addr.s_addr) {
        sin.sin_family = AF_INET;
        sin.sin_addr = *destip;
        return (looutput(&lloif, m, (struct sockaddr *)&sin));
    }
    /* billn: temp...
    if ((ifp->if_flags & IFF_NOARP) || lna >= oldmap) {
        */
    {
        extern iff_noarp;
        if (iff_noarp) {
            bcopy((caddr_t)ac->ac_enaddr, (caddr_t)desten, 3);
            desten[3] = (lna >> 16) & 0x7f;
            desten[4] = (lna >> 8) & 0xff;
            desten[5] = lna & 0xff;
            return (1);
        }
    }
    s = splimp();
    ARPTAB_LOOK(at, destip->s_addr);
    if (at == 0) { /* not found */
        at = arptnew(destip);
        at->at_hold = m;
        arpwhoas(ac, destip);
        splx(s);
        return (0);
    }
}

}

/* restart the timer */
at->at_timer = 0;
if (at->at_flags & ATF_COM) { /* entry IS complete */
    bcopy((caddr_t)at->at_enaddr, (caddr_t)desten, 6);
    splx(s);
    return (1);
}
/*
 * There is an arptab entry, but no ethernet address
 * response yet. Replace the held mbuf with this
 * latest one.
 */
if (at->at_hold)
    m_freem(at->at_hold);
at->at_hold = m;
arpwhoas(ac, destip); /* ask again */
splx(s);
return (0);
}

/*
 * Find my own IP address. It will either be waiting for us in
 * monitor RAM, or can be obtained via broadcast to the file/boot
 * server (not necessarily using the ARP packet format).
 *
 * Unimplemented at present, return 0 and assume that the host
 * will set his own IP address via the SIOCSIFADDR ioctl.
 */
/*ARGSUSED*/
struct in_addr
arpmyaddr(ac)
register struct arpcom *ac;
{
    static struct in_addr addr;

#ifdef lint
    ac = ac;
#endif
    addr.s_addr = 0;
    return (addr);
}

/*
 * Called from ecintr/ilintr when ether packet type ETHERPUP_ARP
 * is received. Algorithm is exactly that given in RFC 826.
 * In addition, a sanity check is performed on the sender
 * protocol address, to catch impersonators.
 */
arpinput(ac, m)
register struct arpcom *ac;
struct mbuf *m;
{
    register struct ether_arp *ea;
    struct ether_header *eh;
    register struct arptab *at = 0; /* same as "merge" flag */
    struct sockaddr_in sin;
    struct sockaddr sa;
    struct mbuf *mhold;
    struct in_addr isaddr, itaddr, myaddr;

    if (m->m_len < sizeof *ea) {
        goto out;
    }
    myaddr = ((struct sockaddr_in *)&ac->ac_if.if_addr)->sin_addr;
    ea = mtod(m, struct ether_arp *);
    if (ntohs(ea->arp_pro) != ETHERPUP_IPTYPE) {
        goto out;
    }
    isaddr.s_addr = ((struct in_addr *)ea->arp_spa)->s_addr;
    itaddr.s_addr = ((struct in_addr *)ea->arp_tpa)->s_addr;
    if (!bcmp((caddr_t)ea->arp_sha, (caddr_t)ac->ac_enaddr,
             sizeof (ac->ac_enaddr))) {
        goto out; /* it's from me, ignore it. */
    }
    if (isaddr.s_addr == myaddr.s_addr) {
        printf("duplicate IP address!! sent from ethernet address: ");

```



```

    printf("%x %x %x %x %x\n", ea->arp_sha[0], ea->arp_sha[1],
           ea->arp_sha[2], ea->arp_sha[3],
           ea->arp_sha[4], ea->arp_sha[5]);
    if (ntohs(ea->arp_op) == ARPOP_REQUEST)
        goto reply;
    goto out;
}
ARPTAB_LOOK(at, isaddr.s_addr);
if (at) {
    bcopy((caddr_t)ea->arp_sha, (caddr_t)at->at_enaddr,
          sizeof (ea->arp_sha));
    at->at_flags |= ATF_COM;
    if (at->at_hold) {
        mhold = at->at_hold;
        at->at_hold = 0;
        sin.sin_family = AF_INET;
        sin.sin_addr = isaddr;
        (*ac->ac_if.if_output)(&ac->ac_if,
                              mhold, (struct sockaddr *)&sin);
    }
}
if (itaddr.s_addr != myaddr.s_addr) {
    goto out; /* if I am not the target */
}
if (at == 0) { /* ensure we have a table entry */
    at = arptnew(&isaddr);
    bcopy((caddr_t)ea->arp_sha, (caddr_t)at->at_enaddr,
          sizeof (ea->arp_sha));
    at->at_flags |= ATF_COM;
}
if (ntohs(ea->arp_op) != ARPOP_REQUEST) {
    goto out;
}
reply:
bcopy((caddr_t)ea->arp_sha, (caddr_t)ea->arp_tha,
      sizeof (ea->arp_sha));
bcopy((caddr_t)ea->arp_spa, (caddr_t)ea->arp_tpa,
      sizeof (ea->arp_spa));
bcopy((caddr_t)ac->ac_enaddr, (caddr_t)ea->arp_sha,
      sizeof (ea->arp_sha));
bcopy((caddr_t)&myaddr, (caddr_t)ea->arp_spa,
      sizeof (ea->arp_spa));
ea->arp_op = htons(ARPOP_REPLY);
eh = (struct ether_header *)sa.sa_data;
bcopy((caddr_t)ea->arp_tha, (caddr_t)eh->ether_dhost,
      sizeof (eh->ether_dhost));
eh->ether_type = ETHERPUP_ARPTYPE;
sa.sa_family = AF_UNSPEC;
(*ac->ac_if.if_output)(&ac->ac_if, m, &sa);
return;
out:
    m_freem(m);
    return;
}
/*
 * Free an arptab entry.
 */
arptfree(at)
    register struct arptab *at;
{
    int s = splimp();

    if (at->at_hold)
        m_freem(at->at_hold);
    at->at_hold = 0;
    at->at_timer = at->at_flags = 0;
    at->at_iaddr.s_addr = 0;
    splx(s);
}
/*
 * Enter a new address in arptab, pushing out the oldest entry
 * from the bucket if there is no room.
 */
struct arptab *

```

```

arptnew(addr)
    struct in_addr *addr;
{
    register n;
    int oldest = 0;
    register struct arptab *at, *ato;

    ato = at = &arptab[ARPTAB_HASH(addr->s_addr) * ARPTAB_BSIZ];
    for (n = 0 ; n < ARPTAB_BSIZ ; n++, at++) {
        if (at->at_flags == 0)
            goto out; /* found an empty entry */
        if (at->at_timer > oldest) {
            oldest = at->at_timer;
            ato = at;
        }
    }
    at = ato;
    arptfree(at);
out:
    at->at_iaddr = *addr;
    at->at_flags = ATF_INUSE;
    return (at);
}

```

```

/* if_il.c 4.11 82/08/25 */

#include "il.h"

/*
 * Interlan Ethernet Communications Controller interface
 */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/system.h"
#include "net/mbuf.h"
#include "sys/buf.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "net/ubavar.h"
#include "net/ilreg.h"
#include "net/in.h"
#include "net/in_sys.h"
#include "net/if.h"
#include "net/if_il.h"
#include "net/if_uba.h"
#include "net/ip.h"
#include "net/ip_var.h"
#include "net/pup.h"
#include "net/route.h"
#include "errno.h"

#define ILMTU 1500 /* Maximum data size for Ethernet packet */
#define ILMIN (60-14) /* Minimum data size for Ethernet packet */

int ilprobe(), ilattach(), ilrint(), ilcint();
struct uba_device *ilinfo[NIL];
u_short ilstd[] = { 0 };
struct uba_driver ildriver =
{ ilprobe, 0, ilattach, 0, ilstd, "il", ilinfo };
#define ILUNIT(x) minor(x)
int ilinit(), iloutput(), ilreset(), ilwatch();

u_char il_ectop[3] = { 0x02, 0x60, 0x8c };
u_char ilbroadcastaddr[6] = { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff };

/*
 * Ethernet software status per interface.
 */
/* Each interface is referenced by a network interface structure,
 * is if, which the routing code uses to locate the interface.
 * This structure contains the output queue for the interface, its address, ...
 * We also have, for each interface, a UBA interface structure, which
 * contains information about the UNIBUS resources held by the interface:
 * map registers, buffered data paths, etc. Information is cached in this
 * structure for use by the if_uba.c routines in running the interface
 * efficiently.
 */
struct il_softc {
    struct ifnet is_if; /* network-visible interface */
    struct ifuba is_ifuba; /* UNIBUS resources */
    int is_flags;
#define ILF_OACTIVE 0x1 /* output is active */
#define ILF_RCVPENDING 0x2 /* start rcv in ilcint */
#define ILF_STATPENDING 0x4 /* stat cmd pending */
    short is_lastcmd; /* can't read csr, so must save it */
    short is_scaninterval; /* interval of stat collection */
#define ILWATCHINTERVAL 60 /* once every 60 seconds */
    struct il_stats is_stats; /* holds on-board statistics */
    struct il_stats is_sum; /* summation over time */
    long is_ubaddr; /* mapping registers of is_stats */
} il_softc[NIL];

ilprobe(reg)
    caddr_t reg;
{
    register int br, cvec; /* r11, r10 value-result */
    register struct ildevice *addr = (struct ildevice *)reg;
    register i;

#ifdef lint
    br = 0; cvec = br; br = cvec;
    ilrint(0); ilcint(0); ilwatch(0);
#endif

    addr->il_csr = ILC_OFFLINE|IL_CIE;
    DELAY(100000);
    i = addr->il_csr; /* clear CDONE */
    if (cvec > 0 && cvec != 0x200)
        cvec -= 4;
    return (1);
}

/*
 * Interface exists: make available by filling in network interface
 * record. System will initialize the interface when it is ready
 * to accept packets. A STATUS command is done to get the ethernet
 * address and other interesting data.
 */
ilattach(ui)
    struct uba_device *ui;
{
    register struct il_softc *is = &il_softc[ui->ui_unit];
    register struct ifnet *ifp = &is->is_if;
    register struct ildevice *addr = (struct ildevice *)ui->ui_addr;
    struct sockaddr_in *sin;

    ifp->if_unit = ui->ui_unit;
    ifp->if_name = "il";
    ifp->if_mtu = ILMTU;
    ifp->if_net = htonl(ui->ui_flags);

    /*
     * Reset the board and map the statistics
     * buffer onto the Unibus.
     */
    addr->il_csr = ILC_RESET;
    while ((addr->il_csr&IL_CDONE) == 0)
        ;
    if (addr->il_csr&IL_STATUS)
        printf("il%d: reset failed, csr=%b\n", ui->ui_unit,
            addr->il_csr, IL_BITS);

    is->is_ubaddr = uballoc(ui->ui_ubanum, &is->is_stats,
        sizeof (struct il_stats), 0);
    addr->il_bar = is->is_ubaddr & 0xffff;
    addr->il_bcr = sizeof (struct il_stats);
    addr->il_csr = ((is->is_ubaddr >> 2) & IL_EVA)|ILC_STAT;
    while ((addr->il_csr&IL_CDONE) == 0)
        ;
    if (addr->il_csr&IL_STATUS)
        printf("il%d: status failed, csr=%b\n", ui->ui_unit,
            addr->il_csr, IL BITS);
    ubarelease(ui->ui_ubanum, &is->is_ubaddr);
    printf("il%d: addr=%x:%x:%x:%x:%x:%x module=%s firmware=%s\n",
        ui->ui_unit,
        is->is_stats.ils_addr[0]&0xff, is->is_stats.ils_addr[1]&0xff,
        is->is_stats.ils_addr[2]&0xff, is->is_stats.ils_addr[3]&0xff,
        is->is_stats.ils_addr[4]&0xff, is->is_stats.ils_addr[5]&0xff,
        is->is_stats.ils_module, is->is_stats.ils_firmware);
    ifp->if_host[0] =
        ((long)(is->is_stats.ils_addr[3]&0xff)<<16) | 0x800000 |
        ((is->is_stats.ils_addr[4]&0xff)<<8) |
        (is->is_stats.ils_addr[5]&0xff);
    sin = (struct sockaddr_in *)&ifp->if_addr;
    sin->sin_family = AF_INET;
    sin->sin_addr = if_makeaddr(ifp->if_net, ifp->if_host[0]);

    sin = (struct sockaddr_in *)&ifp->if_broadaddr;
    sin->sin_family = AF_INET;
    sin->sin_addr = if_makeaddr(ifp->if_net, INADDR_ANY);
    ifp->if_flags = IFF_BROADCAST;

    ifp->if_init = ilinit;
    ifp->if_output = iloutput;
}

```

```

ifp->if_ubareset = ilreset;
ifp->if_watchdog = ilwatch;
is->is_scaninterval = ILWATCHINTERVAL;
ifp->if_timer = is->is_scaninterval;
is->is_ifuba.ifu_flags = UBA_CANTWAIT;
#ifdef notdef
is->is_ifuba.ifu_flags |= UBA_NEEDBDP;
#endif
if_attach(ifp);
}

/*
 * Reset of interface after UNIBUS reset.
 * If interface is on specified uba, reset its state.
 */
ilreset(unit, uban)
int unit, uban;
{
    register struct uba_device *ui;

    if (unit >= NIL || (ui = ilinfo(unit)) == 0 || ui->ui_alive == 0 ||
        ui->ui_ubanum != uban)
        return;
    printf("il%d: ", unit);
    ilinit(unit);
}

/*
 * Initialization of interface; clear recorded pending
 * operations, and reinitialize UNIBUS usage.
 */
ilinit(unit)
int unit;
{
    register struct il_softc *is = &il_softc[unit];
    register struct uba_device *ui = ilinfo(unit);
    register struct ildevice *addr;
    int s;

    if (if_ubainit(&is->is_ifuba, ui->ui_ubanum,
        sizeof(struct il_rheader), (int)btoc(ILMTU)) == 0) {
        printf("il%d: can't initialize\n", unit);
        is->is_if.if_flags &= ~IFF_UP;
        return;
    }
    is->is_ubaddr = uballoc(ui->ui_ubanum, &is->is_stats,
        sizeof(struct il_stats), 0);
    addr = (struct ildevice *)ui->ui_addr;

    /*
     * Turn off source address insertion (it's faster this way),
     * and set board online.
     */
    s = splimp();
    addr->il_csr = ILC_CISA;
    while ((addr->il_csr & IL_CDONE) == 0)
        ;
    addr->il_csr = ILC_ONLINE;
    while ((addr->il_csr & IL_CDONE) == 0)
        ;

    /*
     * Hang receive buffer and start any pending
     * writes by faking a transmit complete.
     * Receive bcr is not a multiple of 4 so buffer
     * chaining can't happen.
     */
    addr->il_bar = is->is_ifuba.ifu_r.ifrw_info & 0xffff;
    addr->il_bcr = sizeof(struct il_rheader) + ILMTU + 6;
    addr->il_csr =
        ((is->is_ifuba.ifu_r.ifrw_info >> 2) & IL_EUA) | ILC_RCV | IL_RIE;
    while ((addr->il_csr & IL_CDONE) == 0)
        ;
    is->is_flags = ILF_OACTIVE;
    is->is_if.if_flags |= IFF_UP;
    is->is_lastcmd = 0;
    ilcint(unit);
}

```

```

splx(s);
if_rtnit(&is->is_if, RTF_UP);
}

/*
 * Start output on interface.
 * Get another datagram to send off of the interface queue,
 * and map it to the interface before starting the output.
 */
ilstart(dev)
dev_t dev;
{
    int unit = ILUNIT(dev), dest, len;
    struct uba_device *ui = ilinfo(unit);
    register struct il_softc *is = &il_softc[unit];
    register struct ildevice *addr;
    struct mbuf *m;
    short csr;

    IF_DEQUEUE(&is->is_if.if_snd, m);
    addr = (struct ildevice *)ui->ui_addr;
    if (m == 0) {
        if ((is->is_flags & ILF_STATPENDING) == 0)
            return;
        addr->il_bar = is->is_ubaddr & 0xffff;
        addr->il_bcr = sizeof(struct il_stats);
        csr = ((is->is_ubaddr >> 2) & IL_EUA) | ILC_STAT | IL_RIE | IL_CIE;
        is->is_flags &= ~ILF_STATPENDING;
        goto startcmd;
    }
    len = if_wubaput(&is->is_ifuba, m);
    /*
     * Ensure minimum packet length.
     * This makes the safe assumption that there are no virtual holes
     * after the data.
     * For security, it might be wise to zero out the added bytes,
     * but we're mainly interested in speed at the moment.
     */
    if (len - sizeof(struct il_xheader) < ILMIN)
        len = ILMIN + sizeof(struct il_xheader);
    if (is->is_ifuba.ifu_flags & UBA_NEEDBDP)
        UBAPURGE(is->is_ifuba.ifu_uba, is->is_ifuba.ifu_w.ifrw_bdp);
    addr->il_bar = is->is_ifuba.ifu_w.ifrw_info & 0xffff;
    addr->il_bcr = len;
    csr =
        ((is->is_ifuba.ifu_w.ifrw_info >> 2) & IL_EUA) | ILC_XMIT | IL_CIE | IL_RIE;

startcmd:
    is->is_lastcmd = csr & IL_CMD;
    addr->il_csr = csr;
    is->is_flags |= ILF_OACTIVE;
}

/*
 * Command done interrupt.
 */
ilcint(unit)
int unit;
{
    register struct il_softc *is = &il_softc[unit];
    struct uba_device *ui = ilinfo(unit);
    register struct ildevice *addr = (struct ildevice *)ui->ui_addr;
    short csr;

    MAPSAVE();
    if ((is->is_flags & ILF_OACTIVE) == 0) {
        printf("il%d: stray xmit interrupt, csr=%b\n", unit,
            addr->il_csr, IL_BITS);
        goto out;
    }

    csr = addr->il_csr;
    /*
     * Hang receive buffer if it couldn't
     * be done earlier (in ilrint).
     */
}

```

```

if (is->is_flags & ILF_RCVPENDING) {
    addr->il_bar = is->is_ifuba.ifu_r.ifrw_info & 0xffff;
    addr->il_bcr = sizeof(struct il_rheader) + ILMTU + 6;
    addr->il_csr =
        ((is->is_ifuba.ifu_r.ifrw_info >> 2) & IL_EUA) | ILC_RCV | IL_RIE;
    while ((addr->il_csr & IL_CDONE) == 0)
        ;
    is->is_flags &= ~ILF_RCVPENDING;
}
is->is_flags &= ~ILF_OACTIVE;
csr &= IL_STATUS;
switch (is->is_lastcmd) {

case ILC_XMIT:
    is->is_if.if_opackets++;
    if (csr > ILERR_RETRIES)
        is->is_if.if_oerrors++;
    break;

case ILC_STAT:
    if (csr == ILERR_SUCCESS)
        iltotal(is);
    break;
}
if (is->is_ifuba.ifu_xtofree) {
    m_freem(is->is_ifuba.ifu_xtofree);
    is->is_ifuba.ifu_xtofree = 0;
}
ilstart(unit);
out:
    MAPREST();
}

/*
 * Ethernet interface receiver interrupt.
 * If input error just drop packet.
 * Otherwise purge input buffered data path and examine
 * packet to determine type. If can't determine length
 * from type, then have to drop packet. Otherwise decapsulate
 * packet based on type and pass to type specific higher-level
 * input routine.
 */
ilrint(unit)
    int unit;
{
    register struct il_softc *is = &il_softc[unit];
    struct ildevice *addr = (struct ildevice *)ilinfo[unit]->ui_addr;
    register struct il_rheader *il;
    struct mbuf *m;
    int len, off, resid;
    register struct ifqueue *inq;

    MAPSAVE();
    is->is_if.if_ipackets++;
    if (is->is_ifuba.ifu_flags & UBA_NEEDBDP)
        UBAPURGE(is->is_ifuba.ifu_uba, is->is_ifuba.ifu_r.ifrw_bdp);
#ifdef lpdpl1
    il = (struct il_rheader *) (is->is_ifuba.ifu_r.ifrw_addr);
#else
    *aka5 = is->is_ifuba.ifu_r.ifrw_click;
    il = (struct il_rheader *) MBX;
#endif
    len = il->ilr_length - sizeof(struct il_rheader);
    if ((il->ilr_status & (ILFSTAT_A | ILFSTAT_C)) || len < 46 || len > ILMTU) {
        is->is_if.if_ierrors++;
#ifdef notdef
        if (is->is_if.if_ierrors % 100 == 0)
            printf("il%d: += 100 input errors\n", unit);
#endif
        goto setup;
    }

/*
 * Deal with trailer protocol: if type is PUP trailer
 * get true type from first 16-bit word past data.
 * Remember that type was trailer by setting off.
 */

```

```

*/
#define ildataaddr(il, off, type) ((type)((caddr_t)((il)+1)+(off)))
if (il->ilr_type >= ILPUP_TRAIL &&
    il->ilr_type < ILPUP_TRAIL+ILPUP_NTRAILER) {
    off = (il->ilr_type - ILPUP_TRAIL) * 512;
    if (off >= ILMTU)
        goto setup; /* sanity */
    il->ilr_type = *ildataaddr(il, off, u_short *);
    resid = *ildataaddr(il, off+2, u_short *);
    if (off + resid > len)
        goto setup; /* sanity */
    len = off + resid;
} else
    off = 0;
if (len == 0)
    goto setup;

/*
 * Pull packet off interface. Off is nonzero if packet
 * has trailing header; ilget will then force this header
 * information to be at the front, but we still have to drop
 * the type and length which are at the front of any trailer data.
 */
m = if_rubaget(&is->is_ifuba, len, off);
if (m == 0)
    goto setup;
if (off) {
    m->m_off += 2 * sizeof(u_short);
    m->m_len -= 2 * sizeof(u_short);
}
switch (il->ilr_type) {

#ifdef INET
case ILPUP_IPTYPE:
    schednetisr(NETISR_IP);
    inq = &pintrq;
    break;
#endif
default:
    m_freem(m);
    goto setup;
}
if (IF_QFULL(inq)) {
    IF_DROP(inq);
    m_freem(m);
    goto setup;
}
IF_ENQUEUE(inq, m);

setup:
/*
 * Reset for next packet if possible.
 * If waiting for transmit command completion, set flag
 * and wait until command completes.
 */
if (is->is_flags & ILF_OACTIVE) {
    is->is_flags |= ILF_RCVPENDING;
    goto out;
}
addr->il_bar = is->is_ifuba.ifu_r.ifrw_info & 0xffff;
addr->il_bcr = sizeof(struct il_rheader) + ILMTU + 6;
addr->il_csr =
    ((is->is_ifuba.ifu_r.ifrw_info >> 2) & IL_EUA) | ILC_RCV | IL_RIE;
while ((addr->il_csr & IL_CDONE) == 0)
    ;
out:
    MAPREST();
}

/*
 * Ethernet output routine.
 * Encapsulate a packet of type family for the local net.
 * Use trailer local net encapsulation if enough data in first
 * packet leaves a multiple of 512 bytes of data in remainder.
 */

```

```

iloutput(ifp, m0, dst)
    struct ifnet *ifp;
    struct mbuf *m0;
    struct sockaddr *dst;
{
    int type, s, error;
    long dest;
    register struct il_softc *is = &il_softc[ifp->if_unit];
    register struct mbuf *m = m0;
    register struct il_xheader *il;
    register int off;

    switch (dst->sa_family) {
#ifdef INET
    case AF_INET:
        dest = ((struct sockaddr_in *)dst)->sin_addr.s_addr;
        off = ntohs((u_short)mtod(m, struct ip *)->ip_len) - m->m_len;
        if (off > 0 && (off & 0x1fff) == 0 &&
            m->m_off >= MMINOFF + 2 * sizeof (u_short)) {
            type = ILPUP_TRAIL + (off>>9);
            m->m_off -= 2 * sizeof (u_short);
            m->m_len += 2 * sizeof (u_short);
            *mtod(m, u_short *) = ILPUP_IPTYPE;
            *(mtod(m, u_short *) + 1) = m->m_len;
            goto gottrailertype;
        }
        type = ILPUP_IPTYPE;
        off = 0;
        goto gotttype;
#endif

    default:
        printf("il%d: can't handle af%d\n", ifp->if_unit,
            dst->sa_family);
        error = EAFNOSUPPORT;
        goto bad;
    }

gottrailertype:
    /*
     * Packet to be sent as trailer: move first packet
     * (control information) to end of chain.
     */
    while (m->m_next)
        m = m->m_next;
    m->m_next = m0;
    m = m0->m_next;
    m0->m_next = 0;
    m0 = m;

gotttype:
    /*
     * Add local net header.  If no space in first mbuf,
     * allocate another.
     */
    if (m->m_off > MMAXOFF ||
        MMINOFF + sizeof (struct il_xheader) > m->m_off) {
        m = m_get(M_DONTWAIT);
        if (m == 0) {
            error = ENOBUFS;
            goto bad;
        }
        m->m_next = m0;
        m->m_off = MMINOFF;
        m->m_len = sizeof (struct il_xheader);
    } else {
        m->m_off -= sizeof (struct il_xheader);
        m->m_len += sizeof (struct il_xheader);
    }
    il = mtod(m, struct il_xheader *);
    if (in_lnaof(dest) == 0)
        bcopy(ilbroadcastaddr, il->ilx_dhost, 6);
    else {
        u_char *to = ntohl(dest) & 0x800000 ?
            is->is_stats.ils_addr : il_ectop;

```

```

    /*
     * this is a kludge to talk with other company's boards;
     * instead 1 byte multicast addresses should be used, and the
     * physical board address will be unused.
     */
    bcopy(to, il->ilx_dhost, 3);
    bcopy(((caddr_t)&dest)+1, &il->ilx_dhost[3], 3);
    il->ilx_dhost[3] &= 0x7f;
}
bcopy(is->is_stats.ils_addr, il->ilx_shost, 6);
il->ilx_type = type;

/*
 * Queue message on interface, and start output if interface
 * not yet active.
 */
s = splimp();
if (IF_QFULL(&ifp->if_snd)) {
    IF_DROP(&ifp->if_snd);
    splx(s);
    m_freem(m);
    return (ENOBUFS);
}
IF_ENQUEUE(&ifp->if_snd, m);
if ((is->is_flags & ILF_OACTIVE) == 0)
    ilstart(ifp->if_unit);
splx(s);
return (0);

bad:
    m_freem(m0);
    return (error);
}

/*
 * Watchdog routine, request statistics from board.
 */
ilwatch(unit)
    int unit;
{
    register struct il_softc *is = &il_softc[unit];
    register struct ifnet *ifp = &is->is_if;
    int s;

    if (is->is_flags & ILF_STATPENDING) {
        ifp->if_timer = is->is_scaninterval;
        return;
    }
    s = splimp();
    is->is_flags |= ILF_STATPENDING;
    if ((is->is_flags & ILF_OACTIVE) == 0)
        ilstart(ifp->if_unit);
    splx(s);
    ifp->if_timer = is->is_scaninterval;
}

/*
 * Total up the on-board statistics.
 */
iltotal(is)
    register struct il_softc *is;
{
    register u_short *interval, *sum, *end;

    interval = &is->is_stats.ils_frames;
    sum = &is->is_sum.ils_frames;
    end = is->is_sum.ils_fill2;
    while (sum < end)
        *sum++ += *interval++;
    is->is_if.if_collisions = is->is_sum.ils_collis;
}

```

```

/*   if_loop.c       4.13   82/06/20   */

/*
 * Loopback interface driver for protocol testing and timing.
 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/system.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/socket.h"
#include "net/in.h"
#include "net/in_system.h"
#include "net/if.h"
#include "net/ip.h"
#include "net/ip_var.h"
#include "net/route.h"
#include "errno.h"

#define LONET  0x7f000000
#define LOMTU  (1024+512)

struct ifnet loif;
int looutput();

loattach()
{
    register struct ifnet *ifp = &loif;
    register struct sockaddr_in *sin;

    ifp->if_name = "lo";
    ifp->if_mtu = LOMTU;
    ifp->if_net = htonl((u_long)LONET);
    sin = (struct sockaddr_in *)&ifp->if_addr;
    sin->sin_family = AF_INET;
    /*
     * sin->sin_addr = if_makeaddr((u_long)ifp->if_net, (u_long)0);
     */
    sin->sin_addr = if_makeaddr((u_long)ifp->if_net, (u_long)1);
    ifp->if_flags = IFF_UP;
    ifp->if_output = looutput;
    if_attach(ifp);
    if_rtinit(ifp, RTF_UP);
}

looutput(ifp, m0, dst)
    register struct ifnet *ifp;
    register struct mbuf *m0;
    register struct sockaddr *dst;
{
    register int s = splimp();
    register struct ifqueue *ifq;

    ifp->if_opackets++;
    switch (dst->sa_family) {
#ifdef INET
    case AF_INET:
        ifq = &ipintrq;
        if (IF_QFULL(ifq)) {
            IF_DROP(ifq);
            m_freem(m0);
            splx(s);
            return (ENOBUFS);
        }
        IF_ENQUEUE(ifq, m0);
        schednetisr(NETISR_IP);
        break;
#endif
    default:
        splx(s);
        printf("lo%d: can't handle af%d\n", ifp->if_unit,
            dst->sa_family);

```

```

        m_freem(m0);
        return (EAFNOSUPPORT);
    }
    ifp->if_ipackets++;
    splx(s);
    return (0);
}

```

```

/*
 * Structure of an Ethernet header.
 */

#include "net/misc.h"
#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/systm.h"
#include "net/mbuf.h"
#include "sys/buf.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/sysmacros.h"
#include "net/ubavar.h"
#include "errno.h"

#include "net/if.h"
#include "net/route.h"
#include "net/in.h"
#include "net/in_systm.h"
#include "net/ip.h"
#include "net/ip_var.h"
#include "net/pup.h"
#include "net/if_ether.h"

/*
 * 3Com Ethernet controller registers.
 */
struct medevice {
    u_short me_csr;          /* control and status */
    u_short me_back;        /* backoff value */
    u_char me_pad1[0x400-2*2];
    u_char me_aram[6];      /* address ROM */
    u_char me_pad2[0x200-6];
    u_char me_aram[6];      /* address RAM */
    u_char me_pad3[0x200-6];
    u_char me_tbuf[2048];   /* transmit buffer */
    u_char me_abuf[2048];   /* receive buffer A */
    u_char me_bbuf[2048];   /* receive buffer B */
};

/*
 * Control and status bits
 */
#define ME_BBSW 0x8000 /* buffer B belongs to ether */
#define ME_ABSW 0x4000 /* buffer A belongs to ether */
#define ME_TBSW 0x2000 /* transmit buffer belongs to ether */
#define ME_JAM 0x1000 /* Ethernet jammed (collision) */
#define ME_AMSW 0x0800 /* address RAM belongs to ether */
#define ME_RBBA 0x0400 /* buffer B older than A */
#define ME_RESET 0x0100 /* reset controller */
#define ME_BINT 0x0080 /* buffer B interrupt enable */
#define ME_AINT 0x0040 /* buffer A interrupt enable */
#define ME_TINT 0x0020 /* transmitter interrupt enable */
#define ME_JINT 0x0010 /* jam interrupt enable */
#define ME_PAMASK 0x000f /* PA field */

/* with old VAX 4.1a, keep range errors, since vax sends too-small packets. */
#define ME_PA_OLDVAX 0x0008 /* receive mine+broadcast-(fcs+frame) */
#define ME_PA 0x0002 /* all- fcs and frame errors */

/*
 * Receive status bits
 */
#define ME_FCSERR 0x8000 /* FCS error */
#define ME_BROADCAST 0x4000 /* packet was broadcast packet */
#define ME_RGERR 0x2000 /* range error */
#define ME_ADDRMATCH 0x1000 /* address match */
#define ME_FRERR 0x0800 /* framing error */

```

```

#define ME_DOFF 0x07ff /* first free byte */
#define MERDOFF 2 /* packet offset in read buffer */
#ifndef notdef
#define MEMAXTDOFF (2048-60) /* max packet offset (min size) */
#endif
#define MEMAXTDOFF (2048-512) /* max packet offset (min size) */

#define NME 1

/*
 * 3Com Ethernet Controller interface
 */

#define MEMTU 1500

int nulldev(), meattach(), meintr();
struct uba_device *meinfo[NME];

struct uba_driver medriver = {
    nulldev, meattach, (u_short *)0, meinfo
};

#define MEUNIT(x) minor(x)

int meinit(), meoutput(), mewatch();
struct mbuf *meget();

extern struct ifnet loif;

#ifdef HAS8259
#define ME_EOI 2
int meeoi = ME_EOI;
#endif

/*
 * Ethernet software status per interface.
 *
 * Each interface is referenced by a network interface structure,
 * es_if, which the routing code uses to locate the interface.
 * This structure contains the output queue for the interface, its address, ...
 */
struct me_softc {
    struct arpcm es_ac; /* common Ethernet structures */
#define es_if es_ac.ac_if /* network-visible interface */
#define es_enaddr es_ac.ac_enaddr /* hardware Ethernet address */
    short es_mask; /* mask for current output delay */
    short es_oactive; /* is output active? */
} me_softc[NME];

/*
 * Interface exists: make available by filling in network interface
 * record. System will initialize the interface when it is ready
 * to accept packets.
 */
meattach(md)
{
    struct uba_device *md;

    struct me_softc *es = &me_softc[md->ui_unit];
    register struct ifnet *ifp = &es->es_if;
    register struct medevice *addr;
    struct sockaddr_in *sin;
    int i;
    u_char *cp, *ap;
    char *memap();

    /* map controller into kernel space for Unisoft. memap is in config.c */
    addr = (struct medevice *) memap((int)md->ui_addr, btoc(8192));
    /* replace the value of ui_addr for everyone else. */
    md->ui_addr = (caddr_t)addr;

    ifp->if_unit = md->ui_unit;
    ifp->if_name = "me";
    ifp->if_mtu = MEMTU;

```

```

ifp->if_net = md->ui_flags & 0xff000000;

if (!iocheck((caddr_t)addr)) {
printf("***\nCould not find me%d; am not initializing network device...*\n",
    ifp->if_unit);
    return;
}

addr->me_csr |= ME_RESET; /* reset the board */
medelay(1); /* wait for it to reset (1 sec) */
/*
 * Read the ethernet address off the board, one byte at a time.
 */
cp = es->es_enaddr;
ap = addr->me_aram;

/* check for mem. board mistakenly mapped to same addr. as me */
if ((*ap == 'g') == 'g') {
    printf (
"\n**Able to write the rom on me%d; probable memory addressing conflict.**\nNot initializing interface.\n\n", ifp->if_unit);
    return;
}

for (i = 0; i < 6; i++)
    *cpt++ = *ap++;
printf("Ethernet address = ");
{
    char *p = &es->es_enaddr[0];
    int i, j = 0;
    char buf[14];

    for (i = 0; i < 6; i++) {
        buf[j++] = "0123456789ABCDEF"[(i*p >> 4)&0xf];
        buf[j++] = "0123456789ABCDEF"[(i*p++)&0xf];
    }
    buf[j++] = '\n';
    buf[j] = 0;
    printf(buf);
}

sin = (struct sockaddr_in *)&es->es_if.if_addr;
sin->sin_family = AF_INET;

/* this is a way to set addresses for now (without an ioctl()) */
sin->sin_addr.s_addr = (u_long)md->ui_flags;
mesetaddr(ifp, sin);

ifp->if_init = meinit;
ifp->if_output = meoutput;
ifp->if_watchdog = mewatch;
if_attach(ifp);
}

mewatch()
{}

/*
 * Initialization of interface; clear recorded pending
 * operations.
 */
meinit(unit)
int unit;
{
    struct me_softc *es = &me_softc[unit];
    register struct ifnet *ifp = &es->es_if;
    register struct sockaddr_in *sin;
    struct medevice *addr;
    int i, s;
    u_char *cp, *ap;

    sin = (struct sockaddr_in *)&ifp->if_addr;
    if (sin->sin_addr.s_addr == 0) /* address still unknown */
        return;
    if ((es->es_if.if_flags & IFF_RUNNING) == 0) {
        addr = (struct medevice *)meinfo[unit]->ui_addr;
        s = splimp();
        /*
 * Initialize the address RAM
 */
        cp = es->es_enaddr;
        ap = addr->me_aram;
        for (i = 0; i < 6; i++)
            *ap++ = *cp++;
        addr->me_csr |= ME_AMSW;

        /*
 * Hang receive buffers and start any pending writes.
 */
        addr->me_csr |= ME_ABSW|ME_AINT|ME_BBSW|ME_BINT
            | ME_PA_OLDVAX;
        es->es_oactive = 0;
        es->es_mask = ~0;
        es->es_if.if_flags |= IFF_UP|IFF_RUNNING;
        if (es->es_if.if_snd.ifq_head)
            mestart(unit);
        splx(s);
        if_rtnit(&es->es_if, RTF_UP);
        arpattach(&es->es_ac);
        arpwhoas(&es->es_ac, &sin->sin_addr);
    }

    /*
 * Start or restart output on interface.
 * If interface is already active, then this is a retransmit
 * after a collision, and just restuff registers.
 * If interface is not already active, get another datagram
 * to send off of the interface queue, and map it to the interface
 * before starting the output.
 */
    mestart(dev)
    register dev_t dev;
    {
        register int unit = MEUNIT(dev);
        register struct me_softc *es = &me_softc[unit];
        register struct medevice *addr;
        register struct mbuf *m;

        addr = (struct medevice *)meinfo[unit]->ui_addr;
        if (es->es_oactive)
            goto restart;

        IF_DEQUEUE(&es->es_if.if_snd, m);
        if (m == 0) {
            es->es_oactive = 0;
            return;
        }
        meput(addr->me_tbuf, m);
        addr->me_csr |= ME_TBSW|ME_TINT|ME_JINT;

restart:
        es->es_oactive = 1;
    }

    /*
 * Ethernet interface interrupt.
 * If received packet examine
 * packet to determine type. If can't determine length
 * from type, then have to drop packet. Otherwise decapsulate
 * packet based on type and pass to type specific higher-level
 * input routine.
 */
    meintr()
    {
        register struct me_softc *es;
        register struct medevice *addr;
        register int unit;
        extern short netoff;
        register unsigned short meenables = 0;

        if (netoff)
            return;
        (void) splnet();
    }
}

```



```

for (unit = 0; unit < NME; unit++) {
es = &me_softc[unit];
addr = (struct medevice *)meinfo[unit]->ui_addr;
switch (addr->me_csr & (ME_ABSW|ME_BBSW|ME_RBBA)) {
case ME_ABSW:
case ME_ABSW|ME_RBBA:
/* BBSW == 0, receive B packet */
addr->me_csr &= ~ME_BINT;
mread(es, addr->me_bbuf);
addr->me_csr |= ME_BBSW|ME_BINT;
break;

case ME_BBSW:
case ME_BBSW|ME_RBBA:
/* ABSW == 0, receive A packet */
addr->me_csr &= ~ME_AINT;
mread(es, addr->me_abuf);
addr->me_csr |= ME_ABSW|ME_AINT;
break;

case ME_RBBA:
/* ABSW == 0, BBSW == 0, RBBA, receive B, then A */
addr->me_csr &= ~(ME_AINT|ME_BINT);
mread(es, addr->me_bbuf);
addr->me_csr |= ME_BBSW|ME_BINT;
mread(es, addr->me_abuf);
addr->me_csr |= ME_ABSW|ME_AINT;
break;

case 0:
/* ABSW == 0, BBSW == 0, RBBA == 0, receive A, then B */
addr->me_csr &= ~(ME_AINT|ME_BINT);
mread(es, addr->me_abuf);
addr->me_csr |= ME_ABSW|ME_AINT;
mread(es, addr->me_bbuf);
addr->me_csr |= ME_BBSW|ME_BINT;
break;

case ME_ABSW|ME_BBSW:
case ME_ABSW|ME_BBSW|ME_RBBA:
/* no input packets */
addr->me_csr &= ~(ME_AINT|ME_BINT);
addr->me_csr |= ME_AINT|ME_BINT;
break;

default:
panic("meintr: impossible value");
/* NOT REACHED */
}
} /* end of "for unit" */

for (unit = 0; unit < NME; unit++) {
es = &me_softc[unit];
addr = (struct medevice *)meinfo[unit]->ui_addr;

if (es->es_oactive == 0)
continue;
if (addr->me_csr & ME_JAM) {
/*
* Collision on ethernet interface. Do exponential
* backoff, and retransmit. If have backed off all
* the way print warning diagnostic, and drop packet.
*/
es->es_if.if_collisions++;
medocoll(unit);
continue;
}
if ((addr->me_csr & ME_TBSW) == 0) {
addr->me_csr &= ~(ME_TINT|ME_JINT);
es->es_if.if_opackets++;
es->es_oactive = 0;
es->es_mask = ~0;
if (es->es_if.if_snd.ifq_head)
mstart(unit);
}
} /* end of "for unit" */

```

```

#ifdef HAS8259
/* try to force level change by exciting current ints on bd */
meenables = addr->me_csr & (ME_TINT|ME_AINT|ME_BINT|ME_JINT);
addr->me_csr &= ~meenables;
sendeoi(meeoi);
addr->me_csr |= meenables;
#endif

return ;

mread(es, mebuf)
register struct me_softc *es;
register caddr_t mebuf;

{
register struct ether_header *me;
register struct mbuf *m;
register int len, off, meoff;
short resid;
register struct ifqueue *inq;

/*
int i;

printf("mer.");
me = (struct ether_header *) (mebuf + MERDOFF);
printf("s:");
for (i = 0; i < 6; i++)
printf("%x.", me->me_shost[i]&0xff);
printf(" ");
printf("d:");
for (i = 0; i < 6; i++)
printf("%x.", me->me_dhost[i]&0xff);
*/

es->es_if.if_ipackets++;
meoff = *(short *)mebuf;

if (meoff & (ME_FRERR|ME_RGERR|ME_FCSERR))
goto err;

meoff &= ME_DOFF;

if (meoff <= MERDOFF || meoff > 2046) {
err:
es->es_if.if_ierrors++;
return;
}

/*
* Get input data length.
* Get pointer to ethernet header (in input buffer).
* Deal with trailer protocol: if type is PUP trailer
* get true type from first 16-bit word past data.
* Remember that type was trailer by setting off.
*/
len = meoff - MERDOFF - sizeof (struct ether_header) - 4; /* 4 == FCS */
me = (struct ether_header *) (mebuf + MERDOFF);
#define medataaddr(me, off, type) ((type)((caddr_t)((me)+1)+(off)))

if (me->ether_type >= ETHERPUP_TRAIL &&
me->ether_type < ETHERPUP_TRAIL+ETHERPUP_NTRAILER) {
off = (me->ether_type - ETHERPUP_TRAIL) * 512;
if (off >= MEMTU)
return; /* sanity */
me->ether_type = *medataaddr(me, off, u_short *);
resid = *(medataaddr(me, off+2, u_short *));
if (off + resid > len)
return; /* sanity */
len = off + resid;
} else
off = 0;
if (len == 0)
return;

/*

```

```

* Pull packet off interface.  Off is nonzero if packet
* has trailing header; meget will then force this header
* information to be at the front, but we still have to drop
* the type and length which are at the front of any trailer data.
*/
m = meget(mebuf, len, off);
if (m == 0)
    return;
if (off) {
    m->m_off += 2 * sizeof(u_short);
    m->m_len -= 2 * sizeof(u_short);
}
switch (me->ether_type) {

#ifdef INET
    case ETHERPUP_IPTYPE:
        schednetisr(NETISR_IP);
        inq = &ipintrq;
        break;

    case ETHERPUP_ARPTYPE:
        arpinput(&es->es_ac, m);
        return;
#endif

    default:
        m_freem(m);
        return;
}

if (IF_QFULL(inq)) {
    IF_DROP(inq);
    m_freem(m);
    return;
}
IF_ENQUEUE(inq, m);
}

medocoll(unit)
int unit;
{
    register struct me_softc *es = &me_softc[unit];
    register struct medevice *addr =
        (struct medevice *)meinfo[unit]->ui_addr;
    int delay;

    /*
     * Es_mask is a 16 bit number with n low zero bits, with
     * n the number of backoffs.  When es_mask is 0 we have
     * backed off 16 times, and give up.
     */
    if (es->es_mask == 0) {
        es->es_if.if_oerrors++;
        printf(
            "%\nme%d: 16 collisions detected on ethernet.  Dropping current packet...\n\n",
                unit);

        /*
         * Reset and transmit next packet (if any).
         */
        es->es_oactive = 0;
        es->es_mask = ~0;
        if (es->es_if.if_snd.ifq_head)
            mestart(unit);

        return;
    }

    /*
     * Do exponential backoff.  Compute delay based on low bits
     * of the time.  A slot time is 51.2 microseconds (rounded to 51).
     * This does not take into account the time already used to
     * process the interrupt.
     */
    es->es_mask <<= 1;
    delay = (time%0xffff) &- es->es_mask;
    addr->me_back = delay * 51;
    addr->me_csr |= ME_JAM|ME_JINT;
}

```

```

/*
 * Ethernet output routine.
 * Encapsulate a packet of type family for the local net.
 * Use trailer local net encapsulation if enough data in first
 * packet leaves a multiple of 512 bytes of data in remainder.
 * If destination is this address or broadcast, send packet to
 * loop device to kludge around the fact that 3com interfaces can't
 * talk to themselves.
 */
meoutput(ifp, m0, dst)
    register struct ifnet *ifp;
    register struct mbuf *m0;
    register struct sockaddr *dst;
{
    int type, s, error;
    u_char edst[6];
    struct in_addr idst;
    register struct me_softc *es = &me_softc[ifp->if_unit];
    register struct mbuf *m = m0;
    register struct ether_header *me;
    register int i;
    struct mbuf *mcopy = (struct mbuf *) 0; /* Null */

    s = splnet();
    switch (dst->sa_family) {

#ifdef INET
    case AF_INET:
        idst = ((struct sockaddr_in *)dst)->sin_addr;
        if (!arpresolve(&es->es_ac, m, &idst, edst))
            return (0); /* if not yet resolved */
        if (in_lnaof(idst) == INADDR_ANY)
            mcopy = m_copy(m, 0, (int)M_COPYALL);
        type = ETHERPUP_IPTYPE;
        goto gotttype;
#endif

    case AF_UNSPEC:
        me = (struct ether_header *)dst->sa_data;
        bcopy((caddr_t)me->ether_dhost, (caddr_t)edst, sizeof(edst));
        type = me->ether_type;
        goto gotttype;

    default:
        printf("%\nme%d: can't handle af%d\n", ifp->if_unit,
            dst->sa_family);
        error = EAFNOSUPPORT;
        goto bad;
    }

gotttype:
    /*
     * Add local net header.  If no space in first mbuf,
     * allocate another.
     */
    if (m->m_off > MMAXOFF ||
        MMINOFF + sizeof(struct ether_header) > m->m_off) {
        m = m_get(M_DONTWAIT);
        if (m == 0) {
            error = ENOBUFS;
            goto bad;
        }
        m->m_next = m0;
        m->m_off = MMINOFF;
        m->m_len = sizeof(struct ether_header);
    } else {
        m->m_off -= sizeof(struct ether_header);
        m->m_len += sizeof(struct ether_header);
    }
    me = mtod(m, struct ether_header *);
    bcopy((caddr_t)edst, (caddr_t)me->ether_dhost, sizeof(edst));
    me->ether_type = htons((u_short)type);
    bcopy((caddr_t)es->es_enaddr, (caddr_t)me->ether_shost, 6);
    /*
     * Queue message on interface, and start output if interface
     * not yet active.
     */
}

```

```

    */
    s = splimp();
    if (IF_QFULL(&ifp->if_snd) {
        IF_DROP(&ifp->if_snd);
        error = ENOBUFS;
        goto qfull;
    }
    IF_ENQUEUE(&ifp->if_snd, m);
    if (es->es_oactive == 0)
        mestart(ifp->if_unit);
    splx(s);

gotlocal:
    return(mcopy ? looutput(&lloif, mcopy, dst) : 0);

qfull:
    m0 = m;
bad:
    m_freem(m0);
    splx(s);
    return(error);
}

/*
 * Routine to copy from mbuf chain to transmitter
 * buffer in Multibus memory.
 */
meput(mebuf, m)
    register u_char *mebuf;
    register struct mbuf *m;
{
    register struct mbuf *mp;
    register short off;
    register u_char *bp;
    register flag = 0;

    for (off = 2048, mp = m; mp; mp = mp->m_next)
        off -= mp->m_len;
    if (off > MEMAXTDOFF) /* enforce minimum packet size */
        off = MEMAXTDOFF;

    if (off & 01) {
        off--;
        flag++;
    }

    *(u_short *)mebuf = off;
    bp = (u_char *) (mebuf + off);
    for (mp = m; mp; mp = mp->m_next) {
        register unsigned len = mp->m_len;
        u_char *mcp;

        if (len == 0)
            continue;
        mcp = mtod(mp, u_char *);
        bcopy(mcp, bp, (int)len);
        bp += len;
    }

    if ((off & 01) && (off > (2048-70)))
        (int i; int j; printf("meput(%x):\n", off); for(j=0;j<4;j++){for(i=0;i<16;i++)printf("%x ", ((char *) (off+mcp))[i+16*j]); printf("\n");});
    /*
    if (flag)
        *bp = 0;

    m_freem(m);
}

/*
 * Routine to copy from Multibus memory into mbufs.
 *
 * Warning: This makes the fairly safe assumption that
 * mbufs have even lengths.
 */

```

```

struct mbuf *
meget(mebuf, totlen, off0)
    register u_char *mebuf;
    register int totlen, off0;
{
    register struct mbuf *m;
    struct mbuf *top = 0, **mp = &top;
    register int off = off0, len;
    register u_char *cp;

    cp = mebuf + MERDOFF + sizeof (struct ether_header);
    /*
    (int i; int j; printf("meget:\n"); for(j=0;j<6;j++){for(i=0;i<16;i++)printf("%x ", ((char *)cp)
    */
    while (totlen > 0) {
        u_char *mcp;

        MGET(m, 0);
        if (m == 0)
            goto bad;
        if (off) {
            len = totlen - off;
            cp = mebuf + MERDOFF + sizeof (struct ether_header) + off;
        } else
            len = totlen;
        m->m_len = len = MIN(MLEN, len);
        m->m_off = MMIOFF;
        mcp = mtod(m, u_char *);
        bcopy(cp, mcp, len);
        cp += len;
        *mp = m;
        mp = &m->m_next;
        if (off == 0) {
            totlen -= len;
            continue;
        }
        off += len;
        if (off == totlen) {
            cp = mebuf + MERDOFF + sizeof (struct ether_header);
            off = 0;
            totlen = off0;
        }
    }
    return (top);
bad:
    m_freem(top);
    return (0);
}

/* medelay -- wait about tim secs */
medelay(tim)
    register tim;
{
    register i;

    while (tim--){
        i = 100000;
        while (i--);
    }
}

mesetaddr(ifp, sin)
    register struct ifnet *ifp;
    register struct sockaddr_in *sin;
{
    ifp->if_addr = *(struct sockaddr *)sin;
    ifp->if_net = in_netof(sin->sin_addr);
    ifp->if_host[0] = in_lnaof(sin->sin_addr);
    sin = (struct sockaddr_in *) &ifp->if_broadaddr;
    sin->sin_family = AF_INET;
    sin->sin_addr = if_makeaddr(ifp->if_net, INADDR_ANY);
    ifp->if_flags |= IFF_BROADCAST;
}

```



```

/* B(8)iget.c 1.4 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/sysinfo.h"
#include "sys/mount.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/inode.h"
#include "sys/file.h"
#include "sys/ino.h"
#include "sys/filsys.h"
#include "sys/buf.h"
#include "sys/var.h"

/*
 * Look up an inode by device, inumber.
 * If it is in core (in the inode structure), honor the locking protocol.
 * If it is not in core, read it in from the specified device.
 * If the inode is mounted on, perform the indicated indirection.
 * In all cases, a pointer to a locked inode structure is returned.
 *
 * printf warning: no inodes -- if the inode structure is full
 * panic: no imt -- if the mounted filesystem is not in the mount table.
 * "cannot happen"
 */

#define NHINO 128 /* must be power of 2 */
#define ihash(X) (&hinode[(int)(X) % (NHINO-1)])
struct hinode {
    struct inode *i_forw;
} hinode[NHINO];
struct inode *ifreelist;

struct inode *
iget(dev, ino)
dev_t dev;
ino_t ino;
{
    register struct inode *ip;
    register struct hinode *hip;
    register struct mount *mp;
    struct inode *iread();

    sysinfo.iget++;
loop:
    hip = ihash(ino);
    for (ip = hip->i_forw; ip; ip = ip->i_forw)
        if (ino == ip->i_number && dev == ip->i_dev)
            goto found;
    if ((ip = ifreelist) == NULL) {
        printf("Inode table overflow\n");
        syserr.inodeovf++;
        u.u_error = ENFILE;
        return(NULL);
    }
    ifreelist = ip->i_forw;
    if (ip->i_forw == hip->i_forw)
        ip->i_forw->i_back = ip;
    ip->i_back = (struct inode *)hip;
    hip->i_forw = ip;
    ip->i_dev = dev;
    ip->i_number = ino;
    ip->i_flag = ILOCK;
    ip->i_count++;
    ip->i_lastr = 0;
    return(iread(ip));
found:
    if((ip->i_flag&ILOCK) != 0) {
        ip->i_flag |= IWANT;
        (void) sleep((caddr_t)ip, PINOD);
        goto loop;
    }
}

if((ip->i_flag&IMOUNT) != 0) {
    for(mp = &mount[0]; mp < (struct mount *)v.v_e_mount; mp++)
        if(mp->m_inodp == ip) {
            dev = mp->m_dev;
            ino = ROOTINO;
            if (ip == mp->m_mount)
                goto found;
            else
                goto loop;
        }
    panic("no imt");
}
ip->i_count++;
ip->i_flag |= ILOCK;
return(ip);
}

inoinit()
{
    register struct inode *ip;
    register short i;

    ifreelist = ip = &inode[0];
    i = v.v_inode - 1 - 1;
    do {
        ip->i_forw = ip+1;
        ip++;
    } while (--i != -1);
#ifdef notdef
    register i = v.v_inode;

    while (--i)
        inode[i-1].i_forw = &inode[i];
    ifreelist = &inode[0];
#endif
}

struct inode *
iread(ip)
register struct inode *ip;
{
    register char *p1, *p2;
    register struct dinode *dp;
    struct buf *bp;
    register short i;

    bp = bread(ip->i_dev, FsITOD(ip->i_dev, ip->i_number));
    if (u.u_error) {
        brelse(bp);
        iput(ip);
        return(NULL);
    }
    dp = bp->b_un.b_dino;
    dp += FsITOO(ip->i_dev, ip->i_number);
    ip->i_mode = dp->di_mode;
    ip->i_nlink = dp->di_nlink;
    ip->i_uid = dp->di_uid;
    ip->i_gid = dp->di_gid;
    ip->i_size = dp->di_size;
    p1 = (char *)ip->i_addr;
    p2 = (char *)dp->di_addr;
    i = NADDR - 1;
    do {
        *p1++ = 0;
        *p1++ = *p2++;
        *p1++ = *p2++;
        *p1++ = *p2++;
    } while (--i != -1);
    brelse(bp);
    return(ip);
}

/*
 * Decrement reference count of an inode structure.
 * On the last reference, write the inode out and if necessary,
 * truncate and deallocate the file.

```

```

*/
input(ip)
register struct inode *ip;
{
    if(ip->i_count == 1) {
        ip->i_flag |= ILOCK;
        if(ip->i_nlink <= 0) {
            itrunc(ip);
            ip->i_mode = 0;
            ip->i_flag |= IUPD|ICHG;
            ifree(ip->i_dev, ip->i_number);
        }
        if(ip->i_flag & (IACC|IUPD|ICHG))
            iupdat(ip, &time, &time);
        prele(ip);
        if(ip->i_back->i_forw == ip->i_forw)
            ip->i_forw->i_back = ip->i_back;
        ip->i_forw = ifreelist;
        ifreelist = ip;
        ip->i_flag = 0;
        ip->i_number = 0;
        ip->i_count = 0;
        return;
    }
    ip->i_count--;
    prele(ip);
}

/*
 * Update the inode with the current time.
 */
iupdat(ip, ta, tm)
register struct inode *ip;
time_t *ta, *tm;
{
    register struct buf *bp;
    struct dinode *dp;
    register char *p1;
    char *p2;
    register short i;

    if(getfs(ip->i_dev)->s_ronly) {
        if(ip->i_flag & (IUPD|ICHG))
            u.u_error = EROFS;
        ip->i_flag &= ~(IACC|IUPD|ICHG|ISYN);
        return;
    }
    bp = bread(ip->i_dev, FsITOD(ip->i_dev, ip->i_number));
    if(bp->b_flags & B_ERROR) {
        brelse(bp);
        return;
    }
    dp = bp->b_un.b_dino;
    dp += FsITGO(ip->i_dev, ip->i_number);
    dp->di_mode = ip->i_mode;
    dp->di_nlink = ip->i_nlink;
    dp->di_uid = ip->i_uid;
    dp->di_gid = ip->i_gid;
    dp->di_size = ip->i_size;
    p1 = (char *)dp->di_addr;
    p2 = (char *)ip->i_addr;
    if ((ip->i_mode & IFMT) == IFIFO) {
        i = NFADDR - 1;
        do {
            if (*p2++ != 0)
                printf("iaddress > 2^24\n");
            *p1++ = *p2++;
            *p1++ = *p2++;
            *p1++ = *p2++;
        } while (--i != -1);
        i = NADDR - NFADDR - 1;
        if (i >= 0) {
            do {
                *p1++ = 0;
                *p1++ = 0;
            }
        }
    }
}

```

```

        *p1++ = 0;
    } while (--i != -1);
} else {
    i = NADDR - 1;
    do {
        if(*p2++ != 0)
            printf("iaddress > 2^24\n");
        *p1++ = *p2++;
        *p1++ = *p2++;
        *p1++ = *p2++;
    } while (--i != -1);
}
if(ip->i_flag & IACC)
    dp->di_atime = *ta;
if(ip->i_flag & IUPD)
    dp->di_mtime = *tm;
if(ip->i_flag & ICHG)
    dp->di_ctime = time;
if(ip->i_flag & ISYN)
    bwrite(bp);
else
    bdwrite(bp);
ip->i_flag &= ~(IACC|IUPD|ICHG|ISYN);
}

/*
 * Free all the disk blocks associated with the specified inode structure.
 * The blocks of the file are removed in reverse order. This FILO
 * algorithm will tend to maintain
 * a contiguous free list much longer than FIFO.
 */
itrunc(ip)
register struct inode *ip;
{
    register i;
    dev_t dev;
    daddr_t bn;

    i = ip->i_mode & IFMT;
    if (i != IFREG && i != IFDIR)
        return;
    dev = ip->i_dev;
    for(i=NADDR-1; i>=0; i--) {
        bn = ip->i_addr[i];
        if(bn == (daddr_t)0)
            continue;
        ip->i_addr[i] = (daddr_t)0;
        switch(i) {
            default:
                free(dev, bn);
                break;

            case NADDR-3:
                tloop(dev, bn, 0, 0);
                break;

            case NADDR-2:
                tloop(dev, bn, 1, 0);
                break;

            case NADDR-1:
                tloop(dev, bn, 1, 1);
        }
    }
    ip->i_size = 0;
    ip->i_flag |= IUPD|ICHG;
}

tloop(dev, bn, f1, f2)
dev_t dev;
daddr_t bn;
{
    register i;
    register struct buf *bp;
}

```

```

register daddr_t *bap;
daddr_t nb;

bp = NULL;
for(i=FsNINDIR(dev)-1; i>=0; i--) {
    if(bp == NULL) {
        bp = bread(dev, bn);
        if (bp->b_flags & B_ERROR) {
            brelse(bp);
            return;
        }
        bap = bp->b_un.b_daddr;
    }
    nb = bap[i];
    if(nb == (daddr_t)0)
        continue;
    if(f1) {
        brelse(bp);
        bp = NULL;
        tloop(dev, nb, f2, 0);
    } else
        free(dev, nb);
}
if(bp != NULL)
    brelse(bp);
free(dev, bn);
}

/*
 * Make a new file.
 */
struct inode *
maknode(mode)
register mode;
{
    register struct inode *ip;

    if ((mode&IFMT) == 0)
        mode |= IFREG;
    mode &= ~u.u_cmask;
    ip = ialloc(u.u_pdir->i_dev, mode, 1);
    if (ip == NULL) {
        iput(u.u_pdir);
        return(NULL);
    }
    wdir(ip);
    return(ip);
}

/*
 * Write a directory entry with parameters left as side effects
 * to a call to namei.
 */
wdir(ip)
struct inode *ip;
{
    register struct user *up;

    up = &u;
    up->u_dent.d_ino = ip->i_number;
    up->u_count = sizeof(struct direct);
    up->u_segflg = 1;
    up->u_base = (caddr_t)&up->u_dent;
    up->u_fmode = FWRITE;
    writel(up->u_pdir);
    iput(up->u_pdir);
}

```

```

/* in.c 4.3 82/06/20 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/system.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "net/in.h"
#include "net/in_system.h"
#include "net/if.h"
#include "net/route.h"
#include "net/af.h"

#ifdef INET
inet_hash(sin, hp)
    register struct sockaddr_in *sin;
    struct afhash *hp;
{
#ifdef ELEVEN
    long l;
    l = in_netof(sin->sin_addr);
    hp->afh_nethash = ((int)l ^ (int)(l>>16)) & 077777;
    l = sin->sin_addr.s_addr;
    hp->afh_hosthash = ((int)l ^ (int)(l>>16)) & 077777;
    if (hp->afh_hosthash < 0)
        hp->afh_hosthash = -hp->afh_hosthash;
#endif
    hp->afh_nethash = in_netof(sin->sin_addr);
    hp->afh_hosthash = ntohl(sin->sin_addr.s_addr);
}

inet_netmatch(sin1, sin2)
    struct sockaddr_in *sin1, *sin2;
{
    return (in_netof(sin1->sin_addr) == in_netof(sin2->sin_addr));
}

/*
 * Formulate an Internet address from network + host. Used in
 * building addresses stored in the ifnet structure.
 */
struct in_addr
if_mkaddr(net, host)
    u_long net, host;
{
    u_long addr;

    addr = htonl(host) | net;
    return (*(struct in_addr *)&addr);
}

/* billn -- for transition */
#ifdef if_makeaddr(x,y)
struct in_addr
if_makeaddr(n, h)
    u_long n, h;
{
    return (if_mkaddr(n, h));
}
#endif

/*
 * Return the network number from an internet
 * address; handles class a/b/c network #'s.
 */
u_long
in_netof(in)
    struct in_addr in;
{
    return (IN_NETOF(in));
}

```

```

/*
 * Return the local network address portion of an
 * internet address; handles class a/b/c network
 * number formats.
 */
u_long
in_lnaof(in)
    struct in_addr in;
{
    return (IN_LNAOF(in));
}

/*
 * Initialize an interface's routing
 * table entry according to the network.
 * INTERNET SPECIFIC.
 */
if_rtinit(ifp, flags)
    register struct ifnet *ifp;
    int flags;
{
    struct sockaddr_in sin;

    if (ifp->if_flags & IFF_ROUTE)
        return;
    bzero((caddr_t)&sin, sizeof (sin));
    sin.sin_family = AF_INET;
    sin.sin_addr = if_makeaddr((u_long)ifp->if_net, (u_long)0);
    rtinit((struct sockaddr *)&sin, &ifp->if_addr, flags);
}
#endif

```



```

#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/system.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/in.h"
#include "net/in_system.h"

#define M68000
#define bswap(x)      (((int)(x))>>8&0xff)|(((int)(x))&0xff)<<8)

/*
 * Checksum routine for Internet Protocol family headers.
 * This routine is very heavily used in the network
 * code and should be rewritten for each CPU to be as fast as possible.
 *
 * billn: This shows the main outline for a prospective algorithm on
 * a prospective machine. I suppose one could try to outline
 * general guidelines for writing this routine, ie, if the machine
 * is byte-swapped, etc, etc. In practice, what one does is find
 * a machine which is known to conform to the "standard" and
 * beat on the code of the new machine till it works with the
 * known ok machine.
 */

/* int    in_ckodd;      /* number of calls on odd start add */
/* int    in_ckprint = 0; /* print sums */
/* 68k */
union w {
    unsigned short wword;
    unsigned char wchar[2];
};

in_cksum(m, len)
register struct mbuf *m;
register short len;
{
    register unsigned short *ptr; /* "unsigned" is important... */
    register short mlen = 0;
    register long result = 0;
    register unsigned short r;
    register char *cptr;
    register unsigned short wasodd;
    register unsigned short thisodd;
    union w w;
    extern short tcpcksum;
    extern short ipcksum;

    /*
     * if (in_ckprint) printf("ck m%o l%o",m,len);
     */
    if (!tcpcksum) /* not checksumming? */
        if (!ipcksum)
            return 0;
    wasodd = 0;
    for (;;) {
        /*
         * Each trip around loop adds in
         * words from one mbuf segment.
         */
        thisodd = 0;
        ptr = mtod(m, unsigned short *);
        mlen = m->m_len;
        if (len < mlen)
            mlen = len;
        len -= mlen;
        if (mlen > 0) {
            if (wasodd) { /* "last mbuf odd" code... */
                cptr = (char *)ptr;
                w.wchar[1] = *cptr++;
                result += w.wword;
                while (--mlen) {
                    w.wchar[0] = *cptr++;
                    mlen--;

```

```

                if (mlen) {
                    w.wchar[1] = *cptr++;
                    result += w.wword;
                }
                else
                    /* note wasodd still set */
                    goto nextbuf; /* next mbuf */
            }
            wasodd = 0;
            goto nextbuf; /* next mbuf */
        }

        /* main line... check odd byte count */
        if (mlen & 01) {
            thisodd++;
            mlen--;
            if (mlen == 0)
                goto lastbyte;
        }
        /* make wc a word count */
        mlen >>= 1;
        /*
         * this is the main loop of the algorithm.
         */
        mlen -- 1;
        do {
            result += *ptr++;
        } while(--mlen != -1);
        if (thisodd) {
            wasodd++;
            cptr = (char *) ptr;
            w.wchar[0] = *cptr++;
        }
        else
            wasodd = 0;
    }

nextbuf:
    if (len <= 0)
        break;
    m = m->m_next;
    /*
     * Locate the next block with some data.
     */
    for (;;) {
        if (m == 0) {
            printf("cksum: out of data\n");
            goto done;
        }
        if (m->m_len)
            break;
        m = m->m_next;
    }
}

done:
if (r = (result >> 16)) {
    result &= 0xffff;
    result += (unsigned)r;
    goto done;
}

/*
 * if (in_ckprint) printf(" s%o\n",~result);
 */
return (((unsigned short)result) & 0xffff);
}

```

```

/*      in_pcb.c      4.28      82/06/20      */
/*
 * 8/27/84 wrn In in_pcbattach, protection check excludes case where port == 20
 * thus allowing ftp to create data sockets. This is a hack, but
 * the decision was made not to put in the setreuid system call for
 * now.
 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "net/in.h"
#include "net/in_system.h"
#include "net/if.h"
#include "net/route.h"
#include "net/in_pcb.h"
#include "net/protosw.h"

/*
 * Routines to manage internet protocol control blocks.
 *
 * At PRU_ATTACH time a protocol control block is allocated in
 * in_pcballoc() and inserted on a doubly-linked list of such blocks
 * for the protocol. A port address is either requested (and verified
 * to not be in use) or assigned at this time. We also allocate
 * space in the socket sockbuf structures here, although this is
 * not a clearly correct place to put this function.
 *
 * A connectionless protocol will have its protocol control block
 * removed at PRU_DETACH time, when the socket will be freed (freeing
 * the space reserved) and the block will be removed from the list of
 * blocks for its protocol.
 *
 * A connection-based protocol may be connected to a remote peer at
 * PRU_CONNECT time through the routine in_pcbconnect(). In the normal
 * case a PRU_DISCONNECT occurs causing a in_pcbdisconnect().
 * It is also possible that higher-level routines will opt out of the
 * relationship with the connection before the connection shut down
 * is complete. This often occurs in protocols like TCP where we must
 * hold on to the protocol control block for a unreasonably long time
 * after the connection is used up to avoid races in later connection
 * establishment. To handle this we allow higher-level routines to
 * disassociate themselves from the socket, marking it SS_USERGONE while
 * the disconnect is in progress. We notice that this has happened
 * when the disconnect is complete, and perform the PRU_DETACH operation,
 * freeing the socket.
 *
 * TODO:
 * use hashing
 */
struct in_addr zeroin_addr;

/*
 * Allocate a protocol control block, space
 * for send and receive data, and local host information.
 * Return error. If no error make socket point at pcb.
 */
in_pcbattach(so, head, sndcc, rcvcc, sin)
    struct socket *so;
    struct inpcb *head;
    int sndcc, rcvcc;
    struct sockaddr_in *sin;
{
    register struct inpcb *inp;
    u_short lport = 0;

    if (ifnet == 0)
        return (EADDRNOTAVAIL);
    if (sin) {
        if (sin->sin_family != AF_INET)
            return (EAFNOSUPPORT);
        if (sin->sin_addr.s_addr) {
            int tport = sin->sin_port;

            sin->sin_port = 0; /* yech... */
            if (if_ifwithaddr((struct sockaddr *)sin) == 0)
                return (EADDRNOTAVAIL);
            sin->sin_port = tport;
        }
        lport = sin->sin_port;
        if (lport) {
            u_short aport = lport;
            int wild = 0;

            ifnet WATCHOUT
            aport = htons(aport);
            #endif

            /* GROSS */
            if (aport < IPPORT_RESERVED && u.u_uid != 0) {
                if (aport != 20)
                    return (EACCES);
            }
            if ((so->so_proto->pr_flags & PR_CONNREQUIRED) == 0 ||
                (so->so_options & SO_ACCEPTCONN) == 0)
                wild = INPLOOKUP_WILDCARD;
            if (in_pcblookup(head,
                zeroin_addr, 0, sin->sin_addr, lport, wild)) {
                return (EADDRINUSE);
            }
        }
    }
    MSGSET(inp, struct inpcb, 1);
    if (inp == 0)
        return (ENOBUFS);
    if (sbreserve(&so->so_snd, sndcc) == 0)
        goto bad;
    if (sbreserve(&so->so_rcv, rcvcc) == 0)
        goto bad2;
    inp->inp_head = head;
    if (sin)
        inp->inp_laddr = sin->sin_addr;
    if (lport == 0)
        do {
            if (head->inp_lport++ < IPPORT_RESERVED)
                head->inp_lport = IPPORT_RESERVED;
            lport = htons(head->inp_lport);
        } while (in_pcblookup(head,
            zeroin_addr, 0, inp->inp_laddr, lport, 0));
    inp->inp_lport = lport;
    inp->inp_socket = so;
    insque(inp, head);
    so->so_pcb = (caddr_t)inp;
    return (0);
bad2:
    sbrelease(&so->so_snd);
bad:
    MSFREE(inp);
    return (ENOBUFS);
}

/*
 * Connect from a socket to a specified address.
 * Both address and port must be specified in argument sin.
 * If don't have a local address for this socket yet,
 * then pick one.
 */
in_pcbconnect(inp, sin)
    - struct inpcb *inp;
    struct sockaddr_in *sin;
{
    struct ifnet *ifp;
    struct sockaddr_in *ifaddr;

```

```

if (sin->sin_family != AF_INET)
    return (EAFNOSUPPORT);
if (sin->sin_addr.s_addr == 0 || sin->sin_port == 0)
    return (EADDRNOTAVAIL);
if (inp->inp_laddr.s_addr == 0) {
    ifp = if_ifonnetof((int) (in_netof(sin->sin_addr)));
    if (ifp == 0) {
        /*
         * We should select the interface based on
         * the route to be used, but for udp this would
         * result in two calls to rmalloc for each packet
         * sent; hardly worthwhile...
         */
        ifp = if_ifwithaf(AF_INET);
        if (ifp == 0)
            return (EADDRNOTAVAIL);
    }
    ifaddr = (struct sockaddr_in *) &ifp->if_addr;
}
if (in_pcblookup(inp->inp_head,
    sin->sin_addr,
    sin->sin_port,
    /* c will pass a structure.
     * inp->inp_laddr.s_addr ? inp->inp_laddr.s_addr : ifaddr->sin_addr.s_addr,
     */
    inp->inp_laddr.s_addr ? inp->inp_laddr : ifaddr->sin_addr,
    inp->inp_lport,
    0) {
    return (EADDRINUSE);
}
if (inp->inp_laddr.s_addr == 0)
    inp->inp_laddr = ifaddr->sin_addr;
inp->inp_faddr = sin->sin_addr;
inp->inp_fport = sin->sin_port;
return (0);
}

in_pcbdisconnect(inp)
    struct inpcb *inp;
{
    inp->inp_faddr.s_addr = 0;
    inp->inp_fport = 0;
    if (inp->inp_socket->so_state & SS_USERGONE)
        in_pcbdetach(inp);
}

in_pcbdetach(inp)
    struct inpcb *inp;
{
    struct socket *so = inp->inp_socket;

    so->so_pcb = 0;
    sofree(so);
    if (inp->inp_route.ro_rt)
        rtfree(inp->inp_route.ro_rt);
    remque(inp);
    MSFREE(inp);
}

in_setsockaddr(sin, inp)
    register struct sockaddr_in *sin;
    register struct inpcb *inp;
{
    if (sin == 0 || inp == 0)
        panic("setsockaddr in");
    bzero((caddr_t) sin, sizeof(*sin));
    sin->sin_family = AF_INET;
    sin->sin_port = inp->inp_lport;
    sin->sin_addr = inp->inp_laddr;
}

/*
 * Pass an error to all internet connections
 * associated with address sin. Call the

```

```

* protocol specific routine to clean up the
* mess afterwards.
*/
in_pcbnotify(head, dst, errno, abort)
    struct inpcb *head;
    register struct in_addr *dst;
    int errno, (*abort)();
{
    register struct inpcb *inp, *oinp;
    int s = splimp();

    for (inp = head->inp_next; inp != head;) {
        if (inp->inp_faddr.s_addr != dst->s_addr) {
            next:
                inp = inp->inp_next;
                continue;
        }
        if (inp->inp_socket == 0)
            goto next;
        inp->inp_socket->so_error = errno;
        oinp = inp;
        inp = inp->inp_next;
        (*abort)(oinp);
    }
    splx(s);
}

/*
 * SHOULD ALLOW MATCH ON MULTI-HOMING ONLY
 */
struct inpcb *
in_pcblookup(head, faddr, fport, laddr, lport, flags)
    struct inpcb *head;
    struct in_addr faddr, laddr;
    u_short fport, lport;
    int flags;
{
    register struct inpcb *inp, *match = 0;
    int matchwild = 3, wildcard;

    for (inp = head->inp_next; inp != head; inp = inp->inp_next) {
        if (inp->inp_lport != lport)
            continue;
        wildcard = 0;
        if (inp->inp_laddr.s_addr != 0) {
            if (laddr.s_addr == 0)
                wildcard++;
            else if (inp->inp_laddr.s_addr != laddr.s_addr)
                continue;
        } else {
            if (laddr.s_addr != 0)
                wildcard++;
        }
        if (inp->inp_faddr.s_addr != 0) {
            if (faddr.s_addr == 0)
                wildcard++;
            else if (inp->inp_faddr.s_addr != faddr.s_addr ||
                inp->inp_fport != fport)
                continue;
        } else {
            if (faddr.s_addr != 0)
                wildcard++;
        }
        if (wildcard && (flags & INPLOOKUP_WILDCARD) == 0)
            continue;
        if (wildcard < matchwild) {
            match = inp;
            matchwild = wildcard;
            if (matchwild == 0)
                break;
        }
    }
    return (match);
}

```

```

/* ip_icmp.c 4.28 83/02/22 */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/system.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "net/in.h"
#include "net/in_system.h"
#include "net/ip.h"
#include "net/ip_icmp.h"
#include "net/route.h"

/*
 * ICMP routines: error generation, receive packet processing, and
 * routines to turnaround packets back to the originator, and
 * host table maintenance routines.
 */
int icmpprintfs = 0;

/*
 * Generate an error packet of type error
 * in response to bad packet ip.
 */
icmp_error(oip, type, code)
    struct ip *oip;
    int type, code;
{
    register unsigned oiplen = oip->ip_hl << 2;
    register struct icmp *icp;
    struct mbuf *m;
    struct ip *nip;

    if (icmpprintfs)
        printf("icmp_error(%x, %d, %d)\n", oip, type, code);
    /*
     * Make sure that the old IP packet had 8 bytes of data to return;
     * if not, don't bother. Also don't EVER error if the old
     * packet protocol was ICMP.
     */
    if (oip->ip_len < 8 || oip->ip_p == IPPROTO_ICMP)
        goto free;

    /*
     * First, formulate icmp message
     m = m_get(M_DONTWAIT, MT_HEADER);
     */
    m = m_get(M_DONTWAIT);
    if (m == 0)
        goto free;
    m->m_len = oiplen + 8 + ICMP_MINLEN;
    m->m_off = MMAXOFF - m->m_len;
    icp = mtod(m, struct icmp *);
    icp->icmp_type = type;
    icp->icmp_void = 0;
    if (type == ICMP_PARAMPROB) {
        icp->icmp_pptr = code;
        code = 0;
    }
    icp->icmp_code = code;
    bcopy((caddr_t)oip, (caddr_t)&icp->icmp_ip, (int)(oiplen + 8));
    nip = &icp->icmp_ip;
    nip->ip_len += oiplen;
    nip->ip_len = htons((u_short)nip->ip_len);

    /*
     * Now, copy old ip header in front of icmp
     * message. This allows us to reuse any source
     * routing info present.
     */
    m->m_off -= oiplen;
    nip = mtod(m, struct ip *);
    bcopy((caddr_t)oip, (caddr_t)nip, (int)oiplen);

    nip->ip_len = m->m_len + oiplen;
    nip->ip_p = IPPROTO_ICMP;
    /* icmp_send adds ip header to m_off and m_len, so we deduct here */
    m->m_off += oiplen;
    icmp_reflect(nip);

free:
    m_freem(mtod(oip));
}

static char icmpmap[] = {
    -1, -1, -1,
    PRC_UNREACH_NET, PRC_QUENCH, PRC_REDIRECT_NET,
    -1, -1, -1,
    -1, -1, PRC_TIMXCEED_INTRANS,
    PRC_PARAMPROB, -1, -1,
    -1, -1
};

static struct sockproto icmpproto = { AF_INET, IPPROTO_ICMP };
static struct sockaddr_in icmpsrc = { AF_INET };
static struct sockaddr_in icmpdst = { AF_INET };

/*
 * Process a received ICMP message.
 */
icmp_input(m)
    struct mbuf *m;
{
    register struct icmp *icp;
    register struct ip *ip = mtod(m, struct ip *);
    int icmplen = ip->ip_len, hlen = ip->ip_hl << 2;
    int i, (*ctfunc)(), type;
    extern u_char ip_protox[];

    /*
     * Locate icmp structure in mbuf, and check
     * that not corrupted and of at least minimum length.
     */
    if (icmpprintfs)
        printf("icmp_input from %x, len %d\n", ip->ip_src, icmplen);
    if (icmplen < ICMP_MINLEN)
        goto free;
    m->m_len -= hlen;
    m->m_off += hlen;
    /* need routine to make sure header is in this mbuf here */
    icp = mtod(m, struct icmp *);
    i = icp->icmp_cksum;
    icp->icmp_cksum = 0;
    if (i != in_cksum(m, icmplen)) {
        printf("icmp: cksum %x\n", i);
        goto free;
    }

    /*
     * Message type specific processing.
     */
    if (icmpprintfs)
        printf("icmp_input, type %d code %d\n", icp->icmp_type,
            icp->icmp_code);
    switch (i = icp->icmp_type) {
        case ICMP_UNREACH:
        case ICMP_TIMXCEED:
        case ICMP_PARAMPROB:
        case ICMP_REDIRECT:
        case ICMP_SOURCEQUENCH:
            /*
             * Problem with previous datagram; advise
             * higher level routines.
             */
            icp->icmp_ip.ip_len = ntohs((u_short)icp->icmp_ip.ip_len);
            if (icmplen < ICMP_ADVLENMIN || icmplen < ICMP_ADVLEN(icp))
                goto free;
            if (icmpprintfs)
                printf("deliver to protocol %d\n", icp->icmp_ip.ip_p);

```

```

type = i == ICMP_PARAMPROB ? 0 : icp->icmp_code;
if (ctlfunc = inetsw[ip_protox[icp->icmp_ip.p]]>pr_ctlinput)
    (*ctlfunc)(icmptype[i] + type, (caddr_t)icp);
goto free;

case ICMP_ECHO:
    icp->icmp_type = ICMP_ECHOREPLY;
    goto reflect;

case ICMP_TSTAMP:
    if (icmplen < ICMP_TSLEN)
        goto free;
    icp->icmp_type = ICMP_TSTAMPREPLY;
    icp->icmp_rtime = iptime();
    icp->icmp_ttime = icp->icmp_rtime;    /* bogus, do later! */
    goto reflect;

case ICMP_IREQ:
    /* fill in source address zero fields! */
    goto reflect;

case ICMP_ECHOREPLY:
case ICMP_TSTAMPREPLY:
case ICMP_IREQREPLY:
    if (icmplen < ICMP_ADVLENMIN || icmplen < ICMP_ADVLEN(icp))
        goto free;
    icmpsrc.sin_addr = ip->ip_src;
    icmpdst.sin_addr = ip->ip_dst;
    raw_input(dtom(icp), &icmpproto, (struct sockaddr *)&icmpsrc,
              (struct sockaddr *)&icmpdst);
    goto free;

default:
    goto free;
}

reflect:
    ip->ip_len += hlen;    /* since ip_input deducts this */
    icmp_reflect(ip);
    return;

free:
    m_freem(dtom(ip));
}

/*
 * Reflect the ip packet back to the source
 * TODO: rearrange ip source routing options.
 */
icmp_reflect(ip)
    struct ip *ip;
{
    struct in_addr t;

    t = ip->ip_dst;
    ip->ip_dst = ip->ip_src;
    ip->ip_src = t;
    icmp_send(ip);
}

/*
 * Send an icmp packet back to the ip level,
 * after supplying a checksum.
 */
icmp_send(ip)
    struct ip *ip;
{
    register int hlen;
    register struct icmp *icp;
    register struct mbuf *m;

    m = dtom(ip);
    hlen = ip->ip_hl << 2;
    icp = mtod(m, struct icmp *);
    icp->icmp_cksum = 0;
    icp->icmp_cksum = in_cksum(m, ip->ip_len - hlen);
    m->m_off -= hlen;
    m->m_len += hlen;

```

```

if (icmpprintfs)
    printf("icmp_send dst %x src %x\n", ip->ip_dst, ip->ip_src);
(void) ip_output(m, (struct mbuf *)0, (struct route *)0, 0);
}

n_time
iptime()
{
    int s = spl6();
    u_long t;

    /* system 3...
    t = (time.tv_sec % (24*60*60)) * 1000 + time.tv_usec / 1000;
    */
    t = (time % SECDAY) * 1000 + lbolt * hz;
    splx(s);
    return (htonl(t));
}

```

```

/* ip_input.c 1.65 83/02/23 */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/system.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "net/in.h"
#include "net/in_system.h"
#include "net/ip.h"
#include "net/ip_icmp.h"

#include "net/if.h"
#include "net/route.h"

#include "net/in_pcb.h"
#include "net/ip_var.h"
#include "net/tcp.h"

u_char ip_protox[IPPROTO_MAX];
int ipqmaxlen = IFQ_MAXLEN;
struct ifnet *ifinet; /* first inet interface */

/*
 * IP initialization: fill in IP protocol switch table.
 * All protocols not implemented in kernel go to raw IP protocol handler.
 */
ip_init()
{
    register struct protosw *pr;
    register int i;

    pr = pffindproto(PF_INET, IPPROTO_RAW);
    if (pr == 0)
        panic("ip_init");
    for (i = 0; i < IPPROTO_MAX; i++)
        ip_protox[i] = pr - inetsw;
    /* billn - meld with old code
    for (pr = inetdomain.dom_protosw;
        pr <= inetdomain.dom_protoswNPROTOSW; pr++)
        if (pr->pr_family == PF_INET &&
            pr->pr_protocol != IPPROTO_RAW)
            ip_protox[pr->pr_protocol] = pr - inetsw;
    */
    for (pr = protosw; pr <= protoswLAST; pr++)
        if (pr->pr_family == PF_INET &&
            pr->pr_protocol != IPPROTO_RAW)
            ip_protox[pr->pr_protocol] = pr - protosw;
    ipq.next = ipq.prev = &ipq;
    /* billn...
    ip_id = time.tv_sec & 0xffff;
    */
    ip_id = time & 0xffff;
    ipintrq.ifq_maxlen = ipqmaxlen;
    ifinet = if_ifwithaf(AF_INET);
}

short ipcksum = 1;
struct ip *ip_reass();
struct sockaddr_in ipaddr = { AF_INET };

/*
 * Ip input routine. Checksum and byte swap header. If fragmented
 * try to reassemble. If complete and fragment queue exists, discard.
 * Process options. Pass to next level.
 */
ipintr()
{
    register struct ip *ip;
    register struct mbuf *m;
    struct mbuf *m0;
    register int i;
    register struct ipq *fp;

```

```

    int hlen, s;

next:
    /*
     * Get next datagram off input queue and get IP header
     * in first mbuf.
     */
    s = splimp();
    IF_DEQUEUE(&ipintrq, m);
    splx(s);
    if (m == 0)
        return;
    if ((m->m_off > MMAKOFF || m->m_len < sizeof (struct ip)) &&
        (m = m_pullup(m, sizeof (struct ip))) == 0) {
        ipstat.ips_toosmall++;
        goto next;
    }
    ip = mtod(m, struct ip *);
    if ((hlen = ip->ip_hl << 2) > m->m_len) {
        if ((m = m_pullup(m, hlen)) == 0) {
            ipstat.ips_badhlen++;
            goto next;
        }
        ip = mtod(m, struct ip *);
    }
    if (ipcksum)
        if (ip->ip_sum != in_cksum(m, hlen)) {
            ipstat.ips_badsum++;
            goto bad;
        }

    /*
     * Convert fields to host representation.
     */
    ip->ip_len = ntohs((u_short)ip->ip_len);
    if (ip->ip_len < hlen) {
        ipstat.ips_badlen++;
        goto bad;
    }
    ip->ip_id = ntohs(ip->ip_id);
    ip->ip_off = ntohs((u_short)ip->ip_off);

    /*
     * Check that the amount of data in the buffers
     * is as at least much as the IP header would have us expect.
     * Trim mbufs if longer than we expect.
     * Drop packet if shorter than we expect.
     */
    i = -ip->ip_len;
    m0 = m;
    for (;;) {
        i += m->m_len;
        if (m->m_next == 0)
            break;
        m = m->m_next;
    }
    if (i != 0) {
        if (i < 0) {
            ipstat.ips_tooshort++;
            printf("[iptooshort by %d]", i);
            goto bad;
        }
        if (i <= m->m_len)
            m->m_len -= i;
        else
            m_adj(m0, -i);
    }
    m = m0;

    /*
     * Process options and, if not destined for us,
     * ship it on. ip_dooptions returns 1 when an
     * error was detected (causing an icmp message
     * to be sent).
     */
    if (hlen > sizeof (struct ip) && ip_dooptions(ip))

```

```

        goto next;

/*
 * Fast check on the first internet
 * interface in the list.
 */
if (ifinet) {
    struct sockaddr_in *sin;

    sin = (struct sockaddr_in *)&ifinet->if_addr;
    if (sin->sin_addr.s_addr == ip->ip_dst.s_addr)
        goto ours;
    sin = (struct sockaddr_in *)&ifinet->if_broadcast;
    if ((ifinet->if_flags & IFF_BROADCAST) &&
        sin->sin_addr.s_addr == ip->ip_dst.s_addr)
        goto ours;
}

/* BEGIN GROT */
#if NND > 0
#include "nd.h"
/*
 * Diskless machines don't initially know
 * their address, so take packets from them
 * if we're acting as a network disk server.
 */
if (ip->ip_dst.s_addr == INADDR_ANY &&
    (in_netof(ip->ip_src) == INADDR_ANY &&
     in_lnaof(ip->ip_src) != INADDR_ANY))
    goto ours;
#endif
/* END GROT */
ipaddr.sin_addr = ip->ip_dst;
if (if_ifwithaddr((struct sockaddr *)&ipaddr) == 0) {
    ip_forward(ip);
    goto next;
}

ours:
/*
 * Look for queue of fragments
 * of this datagram.
 */
for (fp = ipq.next; fp != &ipq; fp = fp->next)
    if (ip->ip_id == fp->ipq_id &&
        ip->ip_src.s_addr == fp->ipq_src.s_addr &&
        ip->ip_dst.s_addr == fp->ipq_dst.s_addr &&
        ip->ip_p == fp->ipq_p)
        goto found;

fp = 0;

found:
/*
 * Adjust ip_len to not reflect header,
 * set ip_mff if more fragments are expected,
 * convert offset of this to bytes.
 */
ip->ip_len -= hlen;
((struct ipasfrag *)ip)->ipf_mff = 0;
if (ip->ip_off & IP_MF)
    ((struct ipasfrag *)ip)->ipf_mff = 1;
ip->ip_off <<= 3;

/*
 * If datagram marked as having more fragments
 * or if this is not the first fragment,
 * attempt reassembly; if it succeeds, proceed.
 */
if (((struct ipasfrag *)ip)->ipf_mff || ip->ip_off) {
    ip = ip_reass((struct ipasfrag *)ip, fp);
    if (ip == 0)
        goto next;
    hlen = ip->ip_hl << 2;
    m = dtom(ip);
} else
    if (fp)
        ip_freef(fp);

/*
 * Switch out to protocol's input routine.
 */
(*inetsw[ip_protox[ip->ip_p]].pr_input)(m);
goto next;

bad:
    m_free(m);
    goto next;
}

/*
 * Take incoming datagram fragment and try to
 * reassemble it into whole datagram. If a chain for
 * reassembly of this datagram already exists, then it
 * is given as fp; otherwise have to make a chain.
 */
struct ip *
ip_reass(ip, fp)
    register struct ipasfrag *ip;
    register struct ipq *fp;
{
    register struct mbuf *m = dtom(ip);
    register struct ipasfrag *q;
    struct mbuf *t;
    int hlen = ip->ip_hl << 2;
    int i, next;

    /*
     * Presence of header sizes in mbufs
     * would confuse code below.
     */
    m->m_off += hlen;
    m->m_len -= hlen;

    /*
     * If first fragment to arrive, create a reassembly queue.
     */
    if (fp == 0) {
        /* billr -- meld with old
         * if ((t = m_get(M_WAIT, MT_FTABLE)) == NULL)
         */
        if ((t = m_get(M_WAIT)) == NULL)
            goto dropfrag;
        fp = mtod(t, struct ipq *);
        insque(fp, &ipq);
        fp->ipq_ttl = IPFRAGTTL;
        fp->ipq_p = ip->ip_p;
        fp->ipq_id = ip->ip_id;
        fp->ipq_next = fp->ipq_prev = (struct ipasfrag *)fp;
        fp->ipq_src = ((struct ip *)ip)->ip_src;
        fp->ipq_dst = ((struct ip *)ip)->ip_dst;
        q = (struct ipasfrag *)fp;
        goto insert;
    }

    /*
     * Find a segment which begins after this one does.
     */
    for (q = fp->ipq_next; q != (struct ipasfrag *)fp; q = q->ipf_next)
        if (q->ipf_off > ip->ip_off)
            break;

    /*
     * If there is a preceding segment, it may provide some of
     * our data already. If so, drop the data from the incoming
     * segment. If it provides all of our data, drop us.
     */
    if (q->ipf_prev != (struct ipasfrag *)fp) {
        i = q->ipf_prev->ipf_off + q->ipf_prev->ip_len - ip->ip_off;
        if (i > 0) {
            if (i >= ip->ip_len)
                goto dropfrag;
            m_adj(dtom(ip), i);
            ip->ip_off += i;
            ip->ip_len -= i;

```

```

    }
}

/*
 * While we overlap succeeding segments trim them or,
 * if they are completely covered, dequeue them.
 */
while (q != (struct ipasfrag *)fp && ip->ip_off + ip->ip_len > q->ip_off) {
    i = (ip->ip_off + ip->ip_len) - q->ip_off;
    if (i < q->ip_len) {
        q->ip_len -= i;
        q->ip_off += i;
        m_adj(dtom(q), i);
        break;
    }
    q = q->ipf_next;
    m_freem(dtom(q->ipf_prev));
    ip_deq(q->ipf_prev);
}

insert:
/*
 * Stick new segment in its place;
 * check for complete reassembly.
 */
ip_enq(ip, q->ipf_prev);
next = 0;
for (q = fp->ipq_next; q != (struct ipasfrag *)fp; q = q->ipf_next) {
    if (q->ip_off != next)
        return (0);
    next += q->ip_len;
}
if (q->ipf_prev->ipf_mff)
    return (0);

/*
 * Reassembly is complete; concatenate fragments.
 */
q = fp->ipq_next;
m = dtom(q);
t = m->m_next;
m->m_next = 0;
m_cat(m, t);
q = q->ipf_next;
while (q != (struct ipasfrag *)fp) {
    t = dtom(q);
    q = q->ipf_next;
    m_cat(m, t);
}

/*
 * Create header for new ip packet by
 * modifying header of first packet;
 * dequeue and discard fragment reassembly header.
 * Make header visible.
 */
ip = fp->ipq_next;
ip->ip_len = next;
((struct ip *)ip)->ip_src = fp->ipq_src;
((struct ip *)ip)->ip_dst = fp->ipq_dst;
remque(fp);
(void) m_free(dtom(fp));
m = dtom(ip);
m->m_len += sizeof (struct ipasfrag);
m->m_off -= sizeof (struct ipasfrag);
return ((struct ip *)ip);
}

dropfrag:
    m_freem(m);
    return (0);
}

/*
 * Free a fragment reassembly header and all
 * associated datagrams.
 */

```

```

ip_freem(fp)
    struct ipq *fp;
{
    register struct ipasfrag *q, *p;

    for (q = fp->ipq_next; q != (struct ipasfrag *)fp; q = p) {
        p = q->ipf_next;
        ip_deq(q);
        m_freem(dtom(q));
    }
    remque(fp);
    (void) m_free(dtom(fp));
}

/*
 * Put an ip fragment on a reassembly chain.
 * Like insque, but pointers in middle of structure.
 */
ip_enq(p, prev)
    register struct ipasfrag *p, *prev;
{
    p->ipf_prev = prev;
    p->ipf_next = prev->ipf_next;
    prev->ipf_next->ipf_prev = p;
    prev->ipf_next = p;
}

/*
 * To ip_enq as remque is to insque.
 */
ip_deq(p)
    register struct ipasfrag *p;
{
    p->ipf_prev->ipf_next = p->ipf_next;
    p->ipf_next->ipf_prev = p->ipf_prev;
}

/*
 * IP timer processing;
 * if a timer expires on a reassembly
 * queue, discard it.
 */
ip_slowtimo()
{
    register struct ipq *fp;
    int s = splnet();

    fp = ipq.next;
    if (fp == 0) {
        splx(s);
        return;
    }
    while (fp != &ipq) {
        --fp->ipq_ttl;
        fp = fp->next;
        if (fp->prev->ipq_ttl == 0)
            ip_freem(fp->prev);
    }
    splx(s);
}

/*
 * Drain off all datagram fragments.
 */
ip_drain()
{
    while (ipq.next != &ipq)
        ip_freem(ipq.next);
}

/*
 * Do option processing on a datagram,
 * possibly discarding it if bad options

```



```

* are encountered.
*/
ip_dooptions(ip)
    struct ip *ip;
{
    register u_char *cp;
    int opt, optlen, cnt, code, type;
    struct in_addr *sin;
    register struct ip_timestamp *ipt;
    register struct ifnet *ifnet;
    struct in_addr t;

    cp = (u_char *) (ip + 1);
    cnt = (ip->ip_hl << 2) - sizeof (struct ip);
    for (; cnt > 0; cnt -= optlen, cp += optlen) {
        opt = cp[0];
        if (opt == IPOPT_EOL)
            break;
        if (opt == IPOPT_NOP)
            optlen = 1;
        else
            optlen = cp[1];
        switch (opt) {
        default:
            break;

        /*
        * Source routing with record.
        * Find interface with current destination address.
        * If none on this machine then drop if strictly routed,
        * or do nothing if loosely routed.
        * Record interface address and bring up next address
        * component. If strictly routed make sure next
        * address on directly accessible net.
        */
        case IPOPT_LSRR:
        case IPOPT_SSRR:
            if (cp[2] < 4 || cp[2] > optlen - (sizeof (long) - 1))
                break;
            sin = (struct in_addr *) (cp + cp[2]);
            ipaddr.sin_addr = *sin;
            ifp = if_ifwithaddr((struct sockaddr *)&ipaddr);
            type = ICMP_UNREACH, code = ICMP_UNREACH_SRCFAIL;
            if (ifp == 0) {
                if (opt == IPOPT_SSRR)
                    goto bad;
                break;
            }
            t = ip->ip_dst; ip->ip_dst = *sin; *sin = t;
            cp[2] += 4;
            if (cp[2] > optlen - (sizeof (long) - 1))
                break;
            ip->ip_dst = sin[1];
            if (opt == IPOPT_SSRR &&
                if_ifonnetof((int) (in_netof(ip->ip_dst))) == 0)
                goto bad;
            break;

        case IPOPT_TS:
            code = cp - (u_char *) ip;
            type = ICMP_PARAMPROB;
            ipt = (struct ip_timestamp *) cp;
            if (ipt->ipt_len < 5)
                goto bad;
            if (ipt->ipt_ptr > ipt->ipt_len - sizeof (long)) {
                if (++ipt->ipt_oflw == 0)
                    goto bad;
                break;
            }
            sin = (struct in_addr *) (cp+cp[2]);
            switch (ipt->ipt_flg) {

            case IPOPT_TS_TSANDADDR:
                if (ipt->ipt_ptr + 8 > ipt->ipt_len)
                    goto bad;
                if (ifinet == 0)
                    goto bad; /* ??? */
                *sin++ = ((struct sockaddr_in *)&ifinet->if_addr)->sin_addr;
                break;

            case IPOPT_TS_PRESPEC:
                ipaddr.sin_addr = *sin;
                if (if_ifwithaddr((struct sockaddr *)&ipaddr) == 0)
                    continue;
                if (ipt->ipt_ptr + 8 > ipt->ipt_len)
                    goto bad;
                ipt->ipt_ptr += 4;
                break;

            default:
                goto bad;
            }
            *(n_time *) sin = iptime();
            ipt->ipt_ptr += 4;
        }
        return (0);
    bad:
        icmp_error(ip, type, code);
        return (1);
    }
}

/*
* Strip out IP options, at higher
* level protocol in the kernel.
* Second argument is buffer to which options
* will be moved, and return value is their length.
*/
ip_stripoptions(ip, mopt)
    struct ip *ip;
    struct mbuf *mopt;
{
    register int i;
    register struct mbuf *m;
    int olen;

    olen = (ip->ip_hl << 2) - sizeof (struct ip);
    m = dtom(ip);
    ip++;
    if (mopt) {
        mopt->m_len = olen;
        mopt->m_off = MMINOFF;
        bcopy((caddr_t) ip, mtod(m, caddr_t), (int) olen);
    }
    i = m->m_len - (sizeof (struct ip) + olen);
    bcopy((caddr_t) ip+olen, (caddr_t) ip, (int) i);
    m->m_len -= olen;
}

u_char inetctlerrmap[] = {
    ECONNABORTED, ECONNABORTED, 0, 0,
    0, 0,
    EHOSTDOWN, EHOSTUNREACH, ENETUNREACH, EHOSTUNREACH,
    ECONNREFUSED, ECONNREFUSED, EMSGSIZE, 0,
    0, 0, 0, 0
};

#ifdef notdef
ip_ctlinput(cmd, arg)
    int cmd;
    caddr_t arg;
{
    struct in_addr *in;
    int tcp_abort(), udp_abort();
    extern struct inpcb tcb, udb;

    if (cmd < 0 || cmd > PRC_NCMTS)
        return;
}

```

```

    if (inetctlerrmap[cmd] == 0)
        return; /* XXX */
    if (cmd == PRC_IFDOWN)
        in = &((struct sockaddr_in *)arg)->sin_addr;
    else if (cmd == PRC_HOSTDEAD || cmd == PRC_HOSTUNREACH)
        in = (struct in_addr *)arg;
    else
        in = &((struct icmp *)arg)->icmp_ip.dst;
/* THIS IS VERY QUESTIONABLE, SHOULD HIT ALL PROTOCOLS */
    in_pcbnotify(&stcb, in, (int)inetctlerrmap[cmd], tcp_abort);
    in_pcbnotify(&udb, in, (int)inetctlerrmap[cmd], udp_abort);
}
#endif

int ipprintfs = 0;
int ipforwarding = 1;
/*
 * Forward a packet. If some error occurs return the sender
 * and icmp packet. Note we can't always generate a meaningful
 * icmp message because icmp doesn't have a large enough repertoire
 * of codes and types.
 */
ip_forward(ip)
    register struct ip *ip;
{
    register int error, type, code;
    struct mbuf *mopt, *mcopy;

    if (ipprintfs)
        printf("forward: src %x dst %x ttl %x\n", ip->ip_src,
            ip->ip_dst, ip->ip_ttl);
    if (ipforwarding == 0) {
        /* can't tell difference between net and host */
        type = ICMP_UNREACH, code = ICMP_UNREACH_NET;
        goto sendicmp;
    }
    if (ip->ip_ttl < IPTTLDEC) {
        type = ICMP_TIMXCEED, code = ICMP_TIMXCEED_INTRANS;
        goto sendicmp;
    }
    ip->ip_ttl -= IPTTLDEC;
    /* billn -- meld with old
    mopt = m_get(M_DONTWAIT, MT_DATA);
    */
    mopt = m_get(M_DONTWAIT);
    if (mopt == NULL) {
        m_freem(dtom(ip));
        return;
    }

    /*
     * Save at most 64 bytes of the packet in case
     * we need to generate an ICMP message to the src.
     */
    /* billn -- imin -> MIN */
    mcopy = m_copy(dtom(ip), 0, MIN(ip->ip_len, 64));
    ip_stripoptions(ip, mopt);

    /* last 0 here means no directed broadcast */
    if ((error = ip_output(dtom(ip), mopt, (struct route *)0, 0)) == 0) {
        if (mcopy)
            m_freem(mcopy);
        return;
    }
    ip = mtod(mcopy, struct ip *);
    type = ICMP_UNREACH, code = 0; /* need ``undefined'' */
    switch (error) {

    case ENETUNREACH:
    case ENETDOWN:
        code = ICMP_UNREACH_NET;
        break;

    case EMSGSIZE:
        code = ICMP_UNREACH_NEEDFRAG;
        break;

    case EPERM:
        /* billn - meld with old
        code = ICMP_UNREACH_PORT;
        break;

        */
    case ENOBUFS:
        type = ICMP_SOURCEQUENCH;
        break;

    case EHOSTDOWN:
    case EHOSTUNREACH:
        code = ICMP_UNREACH_HOST;
        break;
    }
    sendicmp:
        icmp_error(ip, type, code);
}

```

```

/* ip_output.c 1.46 83/02/10 */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/system.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "errno.h"
#include "net/in.h"
#include "net/in_system.h"
#include "net/ip.h"
#include "net/ip_icmp.h"

#include "net/if.h"
#include "net/route.h"

#include "net/ip_var.h"

int ipnorouteprint = 0;

ip_output(m, opt, ro, allowbroadcast)
struct mbuf *m;
struct mbuf *opt;
struct route *ro;
int allowbroadcast;
{
    register struct ip *ip = mtod(m, struct ip *);
    register struct ifnet *ifp;
    int len, hlen = sizeof (struct ip), off, error = 0;
    struct route iproute;
    struct sockaddr *dst;

    if (opt) /* XXX */
        (void) m_free(opt); /* XXX */
    /*
     * Fill in IP header.
     */
    ip->ip_v = IPVERSION;
    ip->ip_hl = hlen >> 2;
    ip->ip_off &= IP_DF;
    ip->ip_id = htons(ip_id++);

    /*
     * Route packet.
     */
    if (ro == 0) {
        ro = &iproute;
        bzero((caddr_t)ro, sizeof (*ro));
    }
    dst = &ro->ro_dst;
    if (ro->ro_rt == 0) {
        ro->ro_dst.sa_family = AF_INET;
        ((struct sockaddr_in *)&ro->ro_dst)->sin_addr = ip->ip_dst;
        /*
         * If routing to interface only, short circuit routing lookup.
         */
        if (ro == &routetoif) {
            /* check ifp is AF_INET??? */
            ifp = if_ifonnetof((int)(in_netof(ip->ip_dst)));
            if (ifp == 0)
                goto unreachable;
            goto gotif;
        }
        rtalloc(ro);
    }
    if (ro->ro_rt == 0 || (ifp = ro->ro_rt->rt_ifp) == 0)
        goto unreachable;
    ro->ro_rt->rt_use++;
    if (ro->ro_rt->rt_flags & RTF_GATEWAY)
        dst = &ro->ro_rt->rt_gateway;
gotif:
#ifdef notdef

```

```

/*
 * If source address not specified yet, use address
 * of outgoing interface.
 */
if (in_lnaof(ip->ip_src) == INADDR_ANY)
    ip->ip_src.sin_addr =
        ((struct sockaddr_in *)&ifp->if_addr)->sin_addr.sin_addr;
#endif

/*
 * Look for broadcast address and
 * and verify user is allowed to send
 * such a packet.
 */
if (in_lnaof(((struct sockaddr_in *)dst)->sin_addr) == INADDR_ANY) {
    if ((ifp->if_flags & IFF_BROADCAST) == 0) {
        error = EADDRNOTAVAIL;
        goto bad;
    }
    if (!allowbroadcast) {
        error = EACCES;
        goto bad;
    }
    /* don't allow broadcast messages to be fragmented */
    if (ip->ip_len > ifp->if_mtu) {
        error = EMSGSIZE;
        goto bad;
    }
}

/*
 * If small enough for interface, can just send directly.
 */
if (ip->ip_len <= ifp->if_mtu) {
    ip->ip_len = htons((u_short)ip->ip_len);
    ip->ip_off = htons((u_short)ip->ip_off);
    ip->ip_sum = 0;
    ip->ip_sum = in_cksum(m, hlen);
    error = (*ifp->if_output)(ifp, m, dst);
    goto done;
}

/*
 * Too large for interface; fragment if possible.
 * Must be able to put at least 8 bytes per fragment.
 */
if (ip->ip_off & IP_DF) {
    error = EMSGSIZE;
    goto bad;
}
len = (ifp->if_mtu - hlen) &~ 7;
if (len < 8) {
    error = EMSGSIZE;
    goto bad;
}

/*
 * Discard IP header from logical mbuf for m_copy's sake.
 * Loop through length of segment, make a copy of each
 * part and output.
 */
m->m_len -= sizeof (struct ip);
m->m_off += sizeof (struct ip);
for (off = 0; off < ip->ip_len-hlen; off += len) {
    /* billn - meld with old
     * struct mbuf *mh = m_get(M_DONTWAIT, MT_HEADER);
     */
    struct mbuf *mh = m_get(M_DONTWAIT);
    struct ip *mhip;

    if (mh == 0) {
        error = ENOBUFS;
        goto bad;
    }
    mh->m_off = MMAXOFF - hlen;
    mhip = mtod(mh, struct ip *);

```

```

    *mhip = *ip;
    if (hlen > sizeof (struct ip)) {
        int olen = ip_optcopy(ip, mhip, off);
        mh->m_len = sizeof (struct ip) + olen;
    } else
        mh->m_len = sizeof (struct ip);
    mhip->ip_off = off >> 3;
    if (off + len >= ip->ip_len-hlen)
        len = mhip->ip_len - ip->ip_len - hlen - off;
    else {
        mhip->ip_len = len;
        mhip->ip_off |= IP_MF;
    }
    mhip->ip_len += sizeof (struct ip);
    mhip->ip_len = htons((u_short)mhip->ip_len);
    mh->m_next = m_copy(m, off, len);
    if (mh->m_next == 0) {
        (void) m_free(mh);
        error = ENOBUFS;          /* ??? */
        goto bad;
    }
    mhip->ip_off = htons((u_short)mhip->ip_off);
    mhip->ip_sum = 0;
    mhip->ip_sum = in_cksum(mh, hlen);
    if (error = (*ifp->if_output)(ifp, mh, dst))
        break;
}
m_freem(m);
goto done;

unreachable:
if (ipnorouteprint)
    printf("no route to %x (from %x, len %d)\n",
        ip->ip_dst.s_addr, ip->ip_src.s_addr, ip->ip_len);
error = ENETUNREACH;

bad:
m_freem(m);

done:
if (ro == siproute && ro->ro_rt)
    RTFREE(ro->ro_rt);
return (error);
}

/*
 * Copy options from ip to jp.
 * If off is 0 all options are copied
 * otherwise copy selectively.
 */
ip_optcopy(ip, jp, off)
    struct ip *ip, *jp;
    int off;
{
    register u_char *cp, *dp;
    int opt, optlen, cnt;

    cp = (u_char *) (ip + 1);
    dp = (u_char *) (jp + 1);
    cnt = (ip->ip_hl << 2) - sizeof (struct ip);
    for (; cnt > 0; cnt -= optlen, cp += optlen) {
        opt = cp[0];
        if (opt == IPOPT_EOL)
            break;
        if (opt == IPOPT_NOP)
            optlen = 1;
        else
            optlen = cp[1];
        if (optlen > cnt)
            optlen = cnt;          /* XXX */
        if (off == 0 || IPOPT_COPIED(opt)) {
            bcopy((caddr_t)cp, (caddr_t)dp, (int)optlen);
            dp += optlen;
        }
    }
    for (optlen = dp - (u_char *) (jp+1); optlen & 0x3; optlen++)
        *dp++ = IPOPT_EOL;
    return (optlen);
}

```

```

/* @(#)ipc.c 1.1 */
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/param.h"
#include "sys/seg.h"
#include "sys/signal.h"
#include "sys/proc.h"
#include "sys/dir.h"
#ifdef u3b
#include "sys/istk.h"
#endif
#include "sys/user.h"
#include "sys/ipc.h"

/*
** Common IPC routines.
*/

/*
** Check message, semaphore, or shared memory access permissions.
**
** This routine verifies the requested access permission for the current
** process. Super-user is always given permission. Otherwise, the
** appropriate bits are checked corresponding to owner, group, or
** everyone. Zero is returned on success. On failure, u.u_error is
** set to EACCES and one is returned.
** The arguments must be set up as follows:
** p - Pointer to permission structure to verify
** mode - Desired access permissions
*/

ipcaccess(p, mode)
register struct ipc_perm *p;
register ushort mode;
{
    if(u.u_uid == 0)
        return(0);
    if(u.u_uid != p->uid && u.u_uid != p->cuid) {
        mode >>= 3;
        if(u.u_gid != p->gid && u.u_gid != p->cgid)
            mode >>= 3;
    }
    if(mode & p->mode)
        return(0);
    u.u_error = EACCES;
    return(1);
}

/*
** Get message, semaphore, or shared memory structure.
**
** This routine searches for a matching key based on the given flags
** and returns a pointer to the appropriate entry. A structure is
** allocated if the key doesn't exist and the flags call for it.
** The arguments must be set up as follows:
** key - Key to be used
** flag - Creation flags and access modes
** base - Base address of appropriate facility structure array
** cnt - # of entries in facility structure array
** size - sizeof(facility structure)
** status - Pointer to status word: set on successful completion
** only: 0 => existing entry found
** 1 => new entry created
** Ipcget returns NULL with u.u_error set to an appropriate value if
** it fails, or a pointer to the initialized entry if it succeeds.
*/

struct ipc_perm *
ipcget(key, flag, base, cnt, size, status)
register struct ipc_perm *base;
int cnt,
flag,
size,
*status;
key_t
{

```

```

register struct ipc_perm *a; /* ptr to available entry */
register int i; /* loop control */

if(key == IPC_PRIVATE) {
    for(i = 0; i++ < cnt;
        base = (struct ipc_perm *)(((char *)base) + size)) {
        if(base->mode & IPC_ALLOC)
            continue;
        goto init;
    }
    u.u_error = ENOSPC;
    return(NULL);
} else {
    for(i = 0, a = NULL; i++ < cnt;
        base = (struct ipc_perm *)(((char *)base) + size)) {
        if(base->mode & IPC_ALLOC) {
            if(base->key == key) {
                if((flag & (IPC_CREAT | IPC_EXCL)) ==
                    (IPC_CREAT | IPC_EXCL)) {
                    u.u_error = EEXIST;
                    return(NULL);
                }
                if((flag & 0777) & ~base->mode) {
                    u.u_error = EACCES;
                    return(NULL);
                }
                *status = 0;
                return(base);
            }
            continue;
        }
        if(a == NULL)
            a = base;
    }
    if(!(flag & IPC_CREAT)) {
        u.u_error = ENOENT;
        return(NULL);
    }
    if(a == NULL) {
        u.u_error = ENOSPC;
        return(NULL);
    }
    base = a;
}

init:
*status = 1;
base->mode = IPC_ALLOC | (flag & 0777);
base->key = key;
base->cuid = base->uid = u.u_uid;
base->cgid = base->gid = u.u_gid;
return(base);
}

```

```

/*
 * (C) 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with Unisoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by Unisoft.
 *
 * Interrupt handler for level 2 interrupts.
 * keyboard, mouse, real time clock, on/off switch
 */

#include "sys/param.h"
#include "sys/types.h"
#include "sys/mmu.h"
#include "sys/reg.h"
#include "sys/local.h"
#include "sys/cops.h"
#include "sys/keyboard.h"
#include "sys/mouse.h"
#include "sys/ms.h"
#include "sys/l2.h"
#include "sys/kb.h"

extern struct rtime rtime;

char *kb_keytab = ToLA;
char kb_altkp; /* are we in alternate keypad mode? (set in vt100.c) */

char pportplug;
char ms_plg, ms_btn;
short ms_row, ms_col;

kbintr()
{
    register char i;
    register struct device_e *p = COPSADDR;
    register char a, ud;
    register int tmp;
    extern time_t lbolt;
#ifdef SUNIX
    extern char kb_getchr; /* flag for polling keyboard */
#endif
    a = p->e_ifr; /* Read and save reason for interrupt */
    if (a & FCA1) { /* keyboard input */
        i = p->e_ira; /* get keyboard/mouse input */
        if (a & FTIMER1)
            a = COPSADDR->e_t1cl; /* prime timer */
        } else { /* no character input */
            a = COPSADDR->e_t1cl; /* prime timer */
            if (--kb_reptrap == 0)
                kbrepeat(); /* possible char repeat */
            return;
        }
    switch (kb_state) {
        case NORMALWAIT: /* IDLE LOOP */
            ud = i & 0x80; /* whether up or down keycode */
            l2_dtrap = lbolt + l2_dtime; /* reset dim delay */
            if (l2_dimmed) /* restore screen intensity */
                l2undim();
            a = kb_keytab[i & 0x7F]; /* convert to ascii */
            if (ud) { /* "key went down" bit */
                /* click(); /* key click */
                if (ARROW(i,a) { /* check arrow keys */
                    kb_chrbuf = Esc; /* send 3-char sequence */
                    coiintr(0);
                    kb_chrbuf = '{';
                    coiintr(0);
                    kb_chrbuf = a;
                    coiintr(0);
                    goto out;
                }
                if (kb_altkp) { /* send special sequence for keypad chars */
                    if (a = altkpad[i & 0x7F]) { /* is it in the keypad? */

```

```

kb_chrbuf = Esc; /* send 3-char sequence */
coiintr(0);
kb_chrbuf = 'O';
coiintr(0);
kb_chrbuf = a;
coiintr(0);
goto out;
}
a = kb_keytab[i & 0x7F]; /* reset to ascii */
}
if (a >= 0) { /* ascii ? */
    kb_keycount++;
    if (kb_ctrl) a &= 0x1F;
    else if (kb_keycount == 1) {
        kb_reptrap = kb_repwait;
        kb_lastc = a;
    } else kb_reptrap = 0;
    kb_chrbuf = a;
    coiintr(0);
    goto out;
}
} else { /* key went up */
    kb_reptrap = 0;
    if (a >= 0) {
        if (kb_keycount-- < 0) {
            kb_keycount = 0;
        }
        goto out;
    }
}
switch (a & 0xF) {
case KB_CTRL: kb_ctrl = ud; kb_reptrap = 0; msintr(M_CTL); goto out;
case KB_SHFT: kb_shft = ud; kbsetcvtab(); msintr(M_SFT); goto out;
case KB_LOCK: kb_lock = ud; kbsetcvtab(); goto out;
case KB_OFF: printf("[SOFT OFF %x]\n",i); goto out;
case KB_MSP: ms_plg = ud; msintr(M_PLUG); goto out;
case KB_MSB: ms_btn = ud; msintr(M_BUT); goto out;
case KB_PPORT: pportplug = ud; goto out;
case KB_D2B: printf("[disk1button %x]\n",i); goto out;
case KB_D2P: printf("[disk1 %x]\n",i); goto out;
case KB_D1B: printf("[disk2button %x]\n",i); goto out;
case KB_D1P: printf("[disk2 %x]\n",i); goto out;
case KB_STATE: if (ud == 0) {
                    kb_state = MOUSERD;
                } else
                    kb_state = RESETCODE;
                goto out;
default: printf("invalid key[0x%x]",i);
}
goto out;
case MOUSERD: /* PICKUP Y axis change in mouse pos */
    kb_state = YMOUSE;
    ms_col = (short)i;
    goto out;
case YMOUSE: /* PICKUP Y axis change in mouse pos */
    kb_state = NORMALWAIT;
    ms_row = (short)i;
    msintr(M_MOVE);
    goto out;
case RESETCODE: /* special condition */
    switch (i & 0xFF) {
        case KB_KBCOPS: /* keyboard cops failure detected */
            printf("KEYBOARD COPS FAILURE\n");
            break;
        case KB_IOCOPS: /* IO board cops failure detected */
            printf("IO BOARD COPS FAILURE\n");
            break;
        case KB_UNPLUG: /* keyboard unplugged */
            kb_chrbuf = 's' & 0x1F; /* cntl S */
            coiintr(0);
            break;
        case KB_CLOCKT: /* clock timer interrupt */
            printf("Real Time Clock interrupt\n");
            break;

```

```

case KB_SFTOFF: /* soft power switch */
    kb_state = SHUTDOWN;
    printf("Shutting down...\n");
    update();
    for (tmp = 0; tmp < 500000; tmp++);
    l2_crate = 2;
    l2_desired = TOTALDIM; /* completely blacken screen */
    l2ramp(0);
    l2ramp(0);
    SPL7(); /* extreme priority */
    l2copscmd(SHUTOFF);
    rom_mon(); /* return to the ROM monitor */
    /*NOTREACHED*/
default: switch (i & 0xF0) {
    case KB_RESERVED:
        printf("[Reserved keycode 0x%x]\n",i);
        break;
    case KB_RDCLK:
        rtime.rt_year = (i & 0xF) + 10;
        kb_state = CLKREAD;
        goto out;
    default:
        kb_idcode = i;
        kb_chrbuf = 'q' & 0x1F; /* cntl Q */
        cointr(0);
        printf("Keyboard type 0x%x\n",i);
    }
}
kb_state = NORMALWAIT;
goto out;
case CLKREAD:
    rtime.rt_day = (((i & 0xF0) >> 4) * 10 + (i & 0xF)) * 10;
    kb_state++;
    goto out;
case CLKREAD+1:
    rtime.rt_day += (i & 0xF0) >> 4;
    rtime.rt_hour = (i & 0xF) * 10;
    kb_state++;
    goto out;
case CLKREAD+2:
    rtime.rt_hour += (i & 0xF0) >> 4;
    rtime.rt_min = (i & 0x0F) * 10;
    kb_state++;
    goto out;
case CLKREAD+3:
    rtime.rt_min += (i & 0xF0) >> 4;
    rtime.rt_sec = (i & 0x0F) * 10;
    kb_state++;
    goto out;
case CLKREAD+4:
    rtime.rt_sec += (i & 0xF0) >> 4;
    rtime.rt_tenth = i & 0x0F;
    kb_state = NORMALWAIT;
    rtCsettod();
    goto out;
case SHUTDOWN:
    goto out;
}
out: ;
#ifdef SUNIX
    if (!ud)
        kb_getchr = 0;
#endif
}

kbrepeat ()
{
    kb_reptrap = kb_repdlay; /* reset repeat timeout */

    if (kb_keycount == 1) {
        kb_chrbuf = kb_lastc;
/* click(); /* key click */
        cointr(0);
    } else
        kb_reptrap = 0; /* reset repeat timeout */
}

```

```

kbsetcvtab()
{
    kb_keytab = ccvtab[(kb_shft?2:0)+(kb_lock?1:0)];
    kb_reptrap = 0;
}

```

```

/*
 * (C) 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with UniSoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by UniSoft.
 *
 * Interrupt handler for level 1 interrupts.
 * parallel port 0, sony, and video circuit controller
 *
 * Since the Parallel Port hard disk interrupt comes in at this level
 * as well as the Sony and vertical retrace (clock) we have included
 * this so we may only call the actual interrupt routine when really
 * necessary.
 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/proc.h"
#include "sys/errno.h"
#include "sys/file.h"
#include "sys/tty.h"
#include "sys/termio.h"
#include "sys/conf.h"
#include "sys/sysinfo.h"
#include "sys/var.h"
#include "sys/reg.h"
#include "setjmp.h"
#include "sys/mmu.h"
#include <sys/pport.h>
#include <sys/cops.h>
#include <sys/mouse.h>

unsigned char msvrcnt;
extern struct msparms mparm;
extern struct proc *msproc;
extern char msblkd;
extern char msvrnsk;

llintr(ap)
struct args *ap;
{
    unsigned char a;
    register struct device_d *devp = PPADDR;
    register struct device_e *f = COPSADDR;

    a = devp->d_ifr;          /* read intr flag register */
    if (a & (FCAL | FTIMER1)) {
        ap->a_dev = 0;      /* set to port number instead of clock value */
        ppintr(ap);        /* built in parallel port */
    } else {
        if ((f->e_irb & FDIR) != 0)
            snintr();      /* sony interrupt */
        else {
            /* verticle retrace interrupt */
            if ((mparm.mp_flags & MF_VRT) != 0) {
                VROFF = 1;
                if (msvrcnt++ >= msvrnsk) {
                    msvrcnt = 0;
                    if (msblkd) {
                        msblkd = 0;
                        wakeup((caddr_t)&msblkd);
                    }
                    psignal(msproc, SIGMOUS);
                }
                VRON = 1;
            } else {
                do {
                    VROFF = 1;
                    VRON = 1;
                } while ((STATUS & S_VR) == 0);
            }
        }
    }
}
    }
}
clock(ap);

```



```

do i--; while(i>0);          /* wait a bit */
dir = 0;                    /* reset dir to cops -> 68K */
*ddra = dir;
splx(pl);
#ifdef lint
    if (crdy) goto send;
#endif lint
}

/* ARGUSED */
l2dim(flag)
{
    extern time_t lbolt;          /* time in ticks */

#ifdef DIMSCREEN
    if (l2_dimmed == 0) {        /* not dimmed already */
        if (l2_dtrap > lbolt) {
            timeout(l2dim, (caddr_t)0, (int)(l2_dtrap - lbolt));
            return;
        }
        l2_desired = l2_dimcont;
        l2_dimmed = 1;
        l2Ramp(0);
    }
#endif

#ifdef HOWFAR
    else printf("\nWHAT!! -- l2dim called while screen was dim\n");
#endif HOWFAR
#ifdef DIMSCREEN
}

l2undim()
{
#ifdef DIMSCREEN
    if (l2_dimmed != 0) {
        l2_dtrap = lbolt + l2_dtime;
        timeout(l2dim, (caddr_t)0, l2_dtime);
        l2_desired = l2_defcont;
        l2_dimmed = 0;
        l2Ramp(0);
    }
#endif

#ifdef HOWFAR
    else printf("\nWHAT!! -- l2undim called while screen was bright\n");
#endif HOWFAR
#ifdef DIMSCREEN
}

l2ramp(tflag)
{
    register struct device_d *pp; /* a5 */
    register int c;              /* d7 */
    int pl;

    switch (tflag) {
        /* who called us */
        /* somebody starting a ramp */
        case 0:
            if (l2_rcflag)
                return;
            break;
        /* timeout (continue ramp) */
        case 1:
            l2_rcflag = 0;
            break;
        /* profile driver (kludge) */
        /* contrast desired */
        /* finished */
        case 2:
            c = l2_desired;
            if (c == l2_contrast)
                return;
            goto skip;
    }

    c = l2_desired;
    if (c == l2_contrast)
        return;

    if (ppinuse) {
        l2_rcflag = 1;
        timeout(l2ramp, (caddr_t)1, 1); /* try again next clock tick */
        return;
    }
}

```

```

if (c < l2_contrast) l2_contrast -= 4; else l2_contrast += 4;
c = l2_contrast;
l2_rcflag = 1;
timeout(l2ramp, (caddr_t)1, l2_crate);

skip:

    pl = spl7();
    pp = PPADDR;
#ifdef lint
    c = (int)pp;
#endif

    asm(" bset      #2,a5@{0x11} ");
    asm(" bset      #2,a5@{1} ");
    asm(" movb      #0xff,a5@{0x19} ");
    asm(" movb      d7,a5@{9} ");
    asm(" bset      #7,a5@{0x11} ");
    asm(" bclr      #7,a5@{1} ");
    asm(" bset      #7,a5@{1} ");
    asm(" bclr      #7,a5@{0x11} ");
    asm(" bclr      #2,a5@{1} ");
    splx(pl);
    return;
}

```

```
/*
 * Line Discipline Switch table
 */
#include "sys/conf.h"

extern nulldev();
extern ttopen(), ttclose(), ttread(), ttwrite(), ttin(), ttout();
struct linesw linesw[] = {
/* 0 */ ttopen, ttclose, ttread, ttwrite, nulldev, ttin, ttout, nulldev,
};
int linecnt = 1;
```

```

/* @(#)lock.c 1.2 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/mmu.h"
#include "sys/proc.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/text.h"
#include "sys/lock.h"

lock()
{
    struct a {
        long oper;
    };

    if (!suser())
        return;
    switch(((struct a *)u.u_ap->oper) {
    case TXTLOCK:
        if ((u.u_lock&(PROCLOCK|TXTLOCK)) || textlock() == 0)
            goto bad;
        break;
    case PROCLOCK:
        if (u.u_lock&(PROCLOCK|TXTLOCK))
            goto bad;
        (void) textlock();
        proclock();
        break;
    case DATLOCK:
        if (u.u_lock&(PROCLOCK|DATLOCK))
            goto bad;
        u.u_lock |= DATLOCK; /* NOP for VAX */
        break;
    case UNLOCK:
        if (punlock() == 0)
            goto bad;
        break;

    default:
        u.u_error = EINVAL;
    }
}

textlock()
{
    if (u.u_procp->p_textp == NULL)
        return(0);
    u.u_lock |= TXTLOCK;
    return(1);
}

tunlock()
{
    if (u.u_procp->p_textp == NULL || (u.u_lock&TXTLOCK) == 0)
        return;
    u.u_lock &= ~TXTLOCK;
}

proclock()
{
    u.u_procp->p_flag |= SSYS;
    u.u_lock |= PROCLOCK;
}

punlock()
{
    if ((u.u_lock&(PROCLOCK|TXTLOCK|DATLOCK)) == 0)
        return(0);
    u.u_procp->p_flag &= ~SSYS;
    u.u_lock &= ~PROCLOCK;
    u.u_lock &= ~DATLOCK;
    tunlock();
}
return(1);
}

```

```

#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/sysinfo.h"
#include "sys/mount.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/inode.h"
#include "sys/file.h"
#include "sys/ino.h"
#include "sys/filsys.h"
#include "sys/buf.h"
#include "sys/var.h"
#include "sys/proc.h"
#include "sys/locking.h"

/*
 * file lock routines
 * John Bass, PO Box 1223, San Luis Obispo, CA 93401
 * Original design spring 1976, CalPoly, San Luis Obispo
 * Deadlock idea from Ed Grudzien at Basys April 1980
 * Extensions Fall 1980, Onyx Systems Inc., San Jose
 * Linted and ported to System V by UniSoft Systems, Berkeley, CA.
 */

#define MAXSIZE (long)(1L<<30) /* number larger than any request */

struct locklist *deadlock();
/*
 * locking -- handles syscall requests
 */
locking() {
    struct file *fp;
    struct inode *ip;
    /*
     * define order and type of syscall args
     */
    register struct a {
        int fdes;
        int flag;
        off_t size;
    } *uap = (struct a *)u.u_ap;
    register struct locklist *cl, *nl;
    off_t LB, UB;

    /*
     * check for valid open file
     */
    fp = getf(uap->fdes);
    if(fp == NULL) return;
    ip = fp->f_inode;
    if ((ip->i_mode&IFMT) == IFDIR) {
        u.u_error = EACCES;
        return;
    }

    /*
     * validate ranges
     * kludge for zero length
     */
    LB = fp->f_offset;
    if(uap->size) {
        UB = LB + uap->size;
        if(UB <= 0) UB = MAXSIZE;
    }
    else UB = MAXSIZE;

    /*
     * test for unlock request
     */
    if(uap->flag == 0) {
        /*
         * starting at list head scan

```

```

 * for locks in the range by
 * this process
 */
cl = (struct locklist *)&ip->i_locklist; /* addr is pointer */
while(nl = cl->ll_link) {
    /*
     * if not by this process skip to next lock
     */
    if(nl->ll_proc != u.u_proc) {
        cl = nl;
        continue;
    }
    /*
     * check for locks in proper range
     */
    if(UB <= nl->ll_start)
        break;
    if(nl->ll_end <= LB) {
        cl = nl;
        continue;
    }
    /*
     * for locks fully contained within
     * requested range, just delete the item
     */
    if(LB <= nl->ll_start && nl->ll_end <= UB) {
        cl->ll_link = nl->ll_link;
        lockfree(nl);
        continue;
    }
    /*
     * if some one is sleeping on this lock
     * do a wakeup, we may free the region
     * being slept on
     */
    if(nl->ll_flags & IWANT) {
        nl->ll_flags &= ~IWANT;
        wakeup((caddr_t)nl);
    }
    /*
     * middle section is being removed
     * add new lock for last section
     * modify existing lock for first section.
     * if no locks, return in error
     */
    if(nl->ll_start < LB && UB < nl->ll_end) {
        if(lockadd(nl,UB,nl->ll_end)) return;
        nl->ll_end = LB;
        break;
    }
    /*
     * first section is being deleted
     * just move starting point up
     */
    if(LB <= nl->ll_start && UB < nl->ll_end) {
        nl->ll_start = UB;
        break;
    }
    /*
     * must be deleting last part of this section
     * move ending point down
     * continue looking for locks covered by upper
     * limit of unlock range
     */
    nl->ll_end = LB;
    cl = nl;
}
/*
 * end of scan for unlock request
 */
return;
}
/*
 * request must be a lock of some kind
 * check to see if the region is lockable by this
 * process

```

```

*/
if(locked(uap->flag, ip, LB, UB))return;
cl = (struct locklist *)&ip->i_locklist; /* note addr is pointer */
/*
 * simple case, no existing locks, simply add new lock
 */
if( (nl=cl->ll_link) == NULL ) {
    (void) lockadd(cl, LB, UB);
    return;
}
/*
 * simple case, lock is before existing locks,
 * simply insert at head of list
 */
if( UB < nl->ll_start ) {
    (void) lockadd(cl, LB, UB);
    return;
}
/*
 * ending range of lock is same as start of lock by
 * another process, simply insert at head of list
 */
if( UB <= nl->ll_start && u.u_procp != nl->ll_procp ) {
    (void) lockadd(cl, LB, UB);
    return;
}
/*
 * request range overlaps with beginning of first request
 * modify starting point in lock to include requested region
 */
if( UB >= nl->ll_start && LB < nl->ll_start ) {
    nl->ll_start = LB;
}
/*
 * scan thru remaining locklist
 */
cl = nl;
for (;;) {
    /*
     * actions for requests at end of list
     */
    if( ( nl = cl->ll_link ) == NULL ) {
        /*
         * request overlaps tail of last entry
         * extend end point
         */
        if( LB <= cl->ll_end && u.u_procp == cl->ll_procp ) {
            if( UB > cl->ll_end ) cl->ll_end = UB;
            return;
        }
        /*
         * otherwise add new entry
         */
        (void) lockadd(cl, LB, UB);
        return;
    }
    /*
     * if more locks in range skip to next
     * otherwise stop scan
     */
    if( nl->ll_start < LB ) {
        cl = nl;
    }
    else {
        break;
    }
}
/*
 * if upper bound is fully resolved were done
 * otherwise fix end of last entry or add new entry
 */
if(UB <= cl->ll_end) return;
if(LB <= cl->ll_end && u.u_procp == cl->ll_procp) cl->ll_end = UB;
else {
    if( lockadd(cl, LB, UB) ) return;
    cl = cl->ll_link;
}

```

```

}
/*
 * end point set above may overlap later entries
 * if so delete or modify them to perform the compaction
 */
while( (nl=cl->ll_link) != NULL ) {
    /*
     * if we found lock by another process we must
     * be done since we validated the range above
     */
    if(u.u_procp != nl->ll_procp) return;
    /*
     * if the new endpoint no longer overlaps were done
     */
    if(cl->ll_end < nl->ll_start) return;
    /*
     * if the new range overlaps the first part of the
     * next lock, take its end point
     * and delete the next lock
     * we should be done
     */
    if(cl->ll_end <= nl->ll_end) {
        cl->ll_end = nl->ll_end;
        cl->ll_link = nl->ll_link;
        lockfree(nl);
        return;
    }
    /*
     * the next lock is fully included in the new range
     * so it may be deleted
     */
    cl->ll_link = nl->ll_link;
    lockfree(nl);
}
}
/*
 * locked -- routine to scan locks and check for a locked condition
 */
locked(flag, ip, LB, UB)
register struct inode *ip;
off_t LB, UB;
{
    register struct locklist *nl = ip->i_locklist;

    /*
     * scan list while locks are in requested range
     */
    while(nl != NULL && nl->ll_start < UB) {
        /*
         * skip locks for this process
         * and those out of range
         */
        if( nl->ll_procp == u.u_procp || nl->ll_end <= LB ) {
            nl = nl->ll_link;
            if(nl == NULL) return(NULL);
            continue;
        }
        /*
         * must have found lock by another process
         * if request is to test only, then exit with
         * error code
         */
        if(flag>1) {
            u.u_error = EACCES;
            return(1);
        }
        /*
         * will need to sleep on lock, check for deadlock first
         * abort on error
         */
        if(deadlock(nl) != NULL) return(1);
        /*
         * post want flag to get awoken
         * then sleep till lock is released
         */
    }
}

```

```

    nl->ll_flags |= IWANT;
    (void) sleep( (caddr_t)nl, PSLEEP);
    /*
     * set scan back to beginning to catch
     * any new areas locked
     * or a partial delete
     */
    nl = ip->i_locklist;
    /*
     * abort if any errors
     */
    if(u.u_error) return(1);
}
return(NULL);
}

/*
 * deadlock -- routine to follow chain of locks and proc table entries
 * to find deadlocks on file locks.
 */
struct locklist *
deadlock(lp)
register struct locklist *lp;
{
    register struct locklist *nl;

    /*
     * scan while the process owning the lock is sleeping
     */
    while(lp->ll_proc->p_stat == SSLEEP) {
        /*
         * if the object the process is sleeping on is
         * NOT in the locktable every thing is ok
         * fall out of loop and return NULL
         */
        nl = lp->ll_proc->p_wlock;
        if( nl < &locklist[0] || nl >= &locklist[(short)v.v_flock] )
            break;

        /*
         * the object was a locklist entry
         * if the owner of that entry is this
         * process then a deadlock would occur
         * set error exit and return
         */
        if(nl->ll_proc == u.u_proc) {
            u.u_error = EDEADLOCK;
            return(nl);
        }
        /*
         * the object was a locklist entry
         * owned by some other process
         * continue the scan with that process
         */
        lp = nl;
    }
    return(NULL);
}

/*
 * unlock -- called by close to release all locks for this process
 */
unlock(ip)
struct inode *ip;
{
    register struct locklist *nl;
    register struct locklist *cl;

    cl = (struct locklist *)&ip->i_locklist;
    while( (nl = cl->ll_link) != NULL) {
        if(nl->ll_proc == u.u_proc) {
            cl->ll_link = nl->ll_link;
            lockfree(nl);
        }
        else cl = nl;
    }
}

```

```

/*
 * lockalloc -- allocates free list, returns free lock items
 */
struct locklist *
lockalloc()
{
    register struct locklist *fl = &locklist[0];
    register struct locklist *nl;

    /*
     * if first entry has never been used
     * link the locklist table into the freelist
     */
    if(fl->ll_proc == NULL) {
        fl->ll_proc = &proc[0];
        for(nl = &locklist[1]; nl < &locklist[(short)v.v_flock]; nl++) {
            lockfree(nl);
        }
    }
    /*
     * if all the locks are used error exit
     */
    if( (nl = fl->ll_link) == NULL) {
        u.u_error = EDEADLOCK;
        return(NULL);
    }
    /*
     * return the next lock on the list
     */
    fl->ll_link = nl->ll_link;
    nl->ll_link = NULL;
    return(nl);
}

/*
 * lockfree -- returns a lock item to the free list
 */
lockfree(lp)
register struct locklist *lp;
{
    register struct locklist *fl = &locklist[0];

    /*
     * if some process is sleeping on this lock
     * wake them up
     */
    if(lp->ll_flags & IWANT) {
        lp->ll_flags &= ~IWANT;
        wakeup((caddr_t)lp);
    }
    /*
     * add the lock into the free list
     */
    lp->ll_link = fl->ll_link;
    fl->ll_link = lp;
}

/*
 * lockadd -- routine to add item to list
 */
lockadd(cl, LB, UB)
register struct locklist *cl;
off_t LB, UB;
{
    register struct locklist *nl;

    /*
     * get a lock, return if none available
     */
    nl = lockalloc();
    if(nl == NULL) {
        return(1);
    }
    /*
     * link the new entry into list at current spot
     * fill in the data from the args
     */
}

```

```
    */
    nl->ll_link = cl->ll_link;
    cl->ll_link = nl;
    nl->ll_proc = u.u_procp;
    nl->ll_start = LB;
    nl->ll_end = UB;
    return(0);
}
```



```

/*
 * (C) 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with Unisoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by Unisoft.
 *
 * Driver for Centronix Citech Dot Matrix Printer
 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/file.h"
#include "sys/tty.h"
#include "sys/termio.h"
#include "sys/conf.h"
#include "sys/sysinfo.h"
#include "sys/var.h"
#include "sys/req.h"
#include "setjmp.h"
#include "sys/mmu.h"

#include "sys/cops.h"
#include "sys/pport.h"

extern struct device_d *pro_da[];

#define LPPRI (PZERO+8)
#define LFLOWAT 40
#define LPHIWAT 100
#define LPMAX NPPDEVS

struct lp {
    struct clist l_outq;
    char flag, ind;
    int ccc, mcc, mlc;
    int line, col;
    int dev;
} lp_dt[LPMAX];

/*
 * flag values - PORT, CAP and NOCR bits from minor device
 */
#define PORT 0x0F /* which port - 1,2,4,5,7,8 */
#define CAP 0x10
#define NOCR 0x20
#define ASLP 0x40 /* only set within driver */
#define OPEN 0x80 /* only set within driver */

#define physical(d) ((minor(d)) & PORT)

#define FORM 014
#define NO_OP asm(" nop ")
char lpflg[LPMAX]; /* whether we expect another interrupt */

/* ARGSUSED */
lpopen(dev, mode)
register dev_t dev;
{
    register unit;
    register struct lp *lp;
    register unsigned char zero;
    register struct device_d *p;
    int lpintr();
    extern char slot[];

    unit = physical(dev);
    SPL5();

```

```

/* check expansion slot number and type (must be 2-port card) */
if (!PPOK(unit) || (slot[PPSLOT(unit)] != PR0)) {
    err:
        u.u_error = EIO;
    fini:
        SPL0();
        return;
}
if ((lp = &lp_dt[unit])->flag) {
    goto err;
}
if (setppint(pro_da[unit], lpintr)) { /* port is already busy */
    u.u_error = ENODEV;
    goto fini;
}
p = pro_da[unit];
zero = 0;
p->d_acr = 0; /* no output latching */
NO_OP;
p->d_pcr = 0x6B; /* set controller CA2 pulse mode strobe */
NO_OP;
p->d_ddra = -1; /* set port A bits to output */
NO_OP;
/*if (p == PPADDR) from system III
p->d_ddrb &= 0x5C; set BSY and OCD to input
else */
p->d_ddrb &= 0xDC; /* two or four port cards */
NO_OP;
p->d_ddrb |= 0x9C; /* set port B bits 2,3,4,7 to out */
NO_OP;
p->d_irb &= ~(DEN|DRW); /* disable buffers */
NO_OP;
p->d_irb |= WCNT; /* set direction to output */
NO_OP;
p->d_ier = FIRQ|FCAL; /* enable interrupt on busy */
NO_OP;
zero = p->d_irb;

if (zero & OCD) { /* out of paper ?? */
    printf("lpopen: cable disconnect or out of paper\n");
out:
    freeppin(p);
    goto err;
}

if ((zero & PCHK) == 0) { /* online ?? */
    printf("lpopen: (ddrb = %x) offline\n", zero);
    goto out;
}

lp->flag = (dev & (PORT | CAP | NOCR)) | OPEN;
lp->ind = 4;
lp->col = 80;
lp->line = 66;
lp->dev = dev;
lpoutput(lp, FORM);
SPL0();
}

lpclose(dev)
register dev_t dev;
{
    register unit;
    register struct lp *lp;

    unit = physical(dev);
    lp = &lp_dt[unit];
    lpoutput(lp, FORM);
    SPL5();
    while (lpflg[unit]) {
        lp->flag |= ASLP;
        (void) sleep((caddr_t)lp, LPPRI);
    }
    freeppin(pro_da[unit]);
    lp->flag = 0;
    SPL0();
}

```

```

ipwrite(dev)
register dev_t dev;
{
    register unit;
    register c;
    register struct lp *lp;

    unit = physical(dev);
    lp = &lp_dt[unit];
    while (u.u_count) {
        SPL5();
        while (lp->l_outq.c_cc > LPHIWAT) {
            lpintr(unit);
            lp->flag |= ASLP;
            (void) sleep((caddr_t)lp, LPPRI);
        }
        SPL0();
        c = fubyte(u.u_base++);
        if (c < 0) {
            u.u_error = EFAULT;
            break;
        }
        u.u_count--;
        lpoutput(lp, c);
    }
    SPL5();
    lpintr(unit);
    SPL0();
}

lpoutput(lp, c)
register struct lp *lp;
register c;
{
    if (lp->flag&CAP) {
        if (c >= 'a' && c <= 'z')
            c += 'A' - 'a'; else
            switch (c) {
                case ' ':
                    c = ' ';
                    goto esc;
                case '}':
                    c = '}';
                    goto esc;
                case '\\':
                    c = '\\';
                    goto esc;
                case '|':
                    c = '|';
                    goto esc;
                case '~':
                    c = '~';
                    goto esc;
                case '\t':
                    lp->ccc = ((lp->ccc+8-lp->ind) & ~7) + lp->ind;
                    return;
                case '\n':
                    lp->mlc++;
                    if (lp->mlc >= lp->line)
                        c = FORM;
                case FORM:
                    lp->mcc = 0;
                    if (lp->mlc) {
                        (void) putc(c, &lp->l_outq);
                        if (c == FORM)
                            lp->mlc = 0;
                    }
                case '\r':
                    lp->ccc = lp->ind;
    }
}

```

```

        SPL5();
        lpintr(lp->dev);
        SPL0();
        return;
    case 010:
        if (lp->ccc > lp->ind)
            lp->ccc--;
        return;
    case ' ':
        lp->ccc++;
        return;
    default:
        if (lp->ccc < lp->mcc) {
            if (lp->flag&NOOCR) {
                lp->ccc++;
                return;
            }
            (void) putc('\r', &lp->l_outq);
            lp->mcc = 0;
        }
        if (lp->ccc < lp->col) {
            while (lp->ccc > lp->mcc) {
                (void) putc(' ', &lp->l_outq);
                lp->mcc++;
            }
            (void) putc(c, &lp->l_outq);
            lp->mcc++;
        }
        lp->ccc++;
    }
}

lpintr(dev)
register dev;
{
    register struct lp *lp;
    register struct device_d *p;
    register c;

    dev = physical(dev);
    if (lpflg[dev]) return;
    lp = &lp_dt[dev];
    p = pro_da[dev];
    while ((c = getc(&lp->l_outq)) >= 0) {
        lpflg[dev] = 1;
        p->d_ira = c;
        c = 30;
        while ((p->d_ifr & FCA1) == 0) {
            if (--c <= 0)
                return;
        }
        lpflg[dev] = 0;
    }

    if (lp->l_outq.c_cc <= LPLOWAT && lp->flag&ASLP) {
        lp->flag &= ~ASLP;
        wakeup((caddr_t)lp);
    }
}

/* ARGSUSED */
lpioctl(dev, cmd, arg, mode)
{
}

```

```

/*#define HOWFAR */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/proc.h"
#include "sys/context.h"
#include "sys/seg.h"
#include "sys/map.h"
#include "sys/reg.h"
#include "sys/psl.h"
#include "sys/utsname.h"
#include "sys/ipc.h"
#include "sys/shm.h"
#include "sys/var.h"
#include "sys/scat.h"
#include "setjmp.h"

typedef int mem_t;

#define LOW (USTART&0xFFFF) /* Low user starting address */
#define HIGH ((USTART>>16)&0xFFFF) /* High user starting address */
#define MEMFACT 8/10 /* swap space to free memory minimum ratio */

int keroff; /* kernel memory offset */

extern short segoff; /* mmu segment offset */

/*
 * Icode is the bootstrap program used to exec init.
 */
short icode[] = {
    /*      */ /*      */ /*      */ /*      */ /*      */
    0x2E7C, HIGH, LOW+0x100, /*      */ /*      */ /*      */ /*      */ /*      */
    0x227C, HIGH, LOW+0x26, /*      */ /*      */ /*      */ /*      */ /*      */
    0x223C, HIGH, LOW+0x22, /*      */ /*      */ /*      */ /*      */ /*      */
    0x207C, HIGH, LOW+0x2A, /*      */ /*      */ /*      */ /*      */ /*      */
    0x42A7, /*      */ /*      */ /*      */ /*      */ /*      */
    0x303C, 0x3B, /*      */ /*      */ /*      */ /*      */ /*      */
    0x4E40, /*      */ /*      */ /*      */ /*      */ /*      */
    0x60FE, /*      */ /*      */ /*      */ /*      */ /*      */
    HIGH, LOW+0x2A, /*      */ /*      */ /*      */ /*      */ /*      */
    0, 0, /*      */ /*      */ /*      */ /*      */ /*      */
    0x2F65, 0x7463, 0x2F69, /*      */ /*      */ /*      */ /*      */ /*      */
    0x6E69, 0x7400
};

int szicode = sizeof(icode);

/*
 * startup code
 */
/* ARGSUSED */
startup(cudot)
{
    extern int nsysseg, dispatch[];
    extern caddr_t end, etext;
    extern struct var v;
    register caddr_t cp, limit;
    register long *ip;

    for (ip = &((long *) 0)[0]; ip < &((long *) 0)[NIVEC]; ip++)
        *ip = (long)dispatch + (long)ip;

    /*
     * free swap memory
     */
    mfree(swapmap, nswap, 1);

    keroff = btoc((unsigned)&end) + v.v_usize;

```

```

    cp = (caddr_t)(ctob(btoc((unsigned)&end)+v.v_usize));
    limit = (caddr_t){*((long *)MEMEND) & 0xFFFFFE00}-ctob(1)};

#ifdef HOWFAR
    printf("Segment offset = 0x%x, first click at %d(0x%x)\n\r",
           segoff, keroff, keroff);
    printf("First free memory = 0x%x\n\r", cp);
    printf("Last free memory = 0x%x\n\r", limit);
    printf("%d clicks available\n\r",btoc((int)limit)-keroff);
#endif

    for (; cp < limit; cp += ctob(1)) {
        clearseg((int)btoc((int)cp));
        maxmem++;
    }
    cp = (caddr_t)(btoc((unsigned)&end)+v.v_usize);
    mfree(coremap, maxmem, (int)cp);

    printf("Available user memory is %d bytes\n\r", ctob((long)maxmem));
    if (maxmem > dtoc(nswap*MEMFACT)) {
        maxmem = dtoc(nswap*MEMFACT);
        printf("Insufficient swap space for available memory.\n\r");
        printf("Largest runnable process is %d.\n\r", ctob(maxmem));
    }
    if (MAXMEM < maxmem) maxmem = MAXMEM;
}

/*
 * ioccheck - check for an I/O device at given address
 */
ioccheck(addr)
caddr_t addr;
{
    register int *saved_jb;
    register int i;
    jmp_buf jb;

    saved_jb = nofault;
    if (!setjmp(jb)) {
        nofault = jb;
        i = *addr;
        i = 1;
    } else
        i = 0;
    nofault = saved_jb;
    return(i);
}

/*
 * usraccess - Check a virtual address for user accessibility.
 */
usraccess(virt, access)
long virt;
{
    register prot;

    SEG1_1 = 1; /* user context */
    /* SEG2_0 = 1; /* user context */

    prot = getmmu((short *) (vtoseq(virt) | ACCLIM)) & PROTMASK;
    /* printf("usraccess at 0x%x is 0x%x\n\r", virt, prot); */
    if (prot == ASRW || prot == ASRWS || (prot == ASRO && access == RO))
        return(1);
    return(0);
}

/*
 * vtop - Convert virtual address to physical.
 */
caddr_t
vtop(virt)
register caddr_t virt;
{
    long seqbase, phys;

    SEG1_1 = 1; /* user context */
    /* SEG2_0 = 1; /* user context */

```

```

    segbase = getmmu((short *) (vtoseg(virt) | ACCSEG) & SEGBASE;
    phys = (segbase << VIRTSHIFT) + ((long)virt & OFFMASK);
    return((caddr_t)(phys - ctob(segoff)));
}

/*
 * fuword - Fetch word from user space.
 */
fuword(addr)
register caddr_t addr;
{
    int val;
    jmp_buf jb;
    int *saved_jb;

    if (addr < (caddr_t)v.v_ustart || addr+3 >= (caddr_t)v.v_uend)
        return (-1);
    if ((addr = (caddr_t)vtop(addr)) == (caddr_t)-1)
        return(-1);
    saved_jb = nofault;
    if (!setjmp(jb)) {
        nofault = jb;
        val = *(int *)addr;
    } else
        val = -1;
    nofault = saved_jb;
    return(val);
}

/*
 * fubyte - Fetch byte from user space.
 */
fubyte(addr)
register caddr_t addr;
{
    register int val;
    jmp_buf jb;
    int *saved_jb;

    if (addr < (caddr_t)v.v_ustart || addr >= (caddr_t)v.v_uend)
        return (-1);
    if ((addr = (caddr_t)vtop(addr)) == (caddr_t)-1)
        return(-1);
    saved_jb = nofault;
    if (!setjmp(jb)) {
        nofault = jb;
        val = *addr & 0377;
    } else
        val = -1;
    nofault = saved_jb;
    return(val);
}

/*
 * suword - Store word into user space.
 */
suword(addr, word)
register caddr_t addr;
int word;
{
    jmp_buf jb;
    int val, *saved_jb;

    if (addr < (caddr_t)v.v_ustart || addr+3 >= (caddr_t)v.v_uend)
        return (-1);
    if ((addr = (caddr_t)vtop(addr)) == (caddr_t)-1)
        return(-1);
    saved_jb = nofault;
    if (!setjmp(jb)) {
        nofault = jb;
        *(int *)addr = word;
        val = 0;
    } else
        val = -1;
}

```

```

    nofault = saved_jb;
    return(val);
}

/*
 * subyte - Store byte into user space.
 */
subyte(addr, byte)
register caddr_t addr;
char byte;
{
    jmp_buf jb;
    int val, *saved_jb;

    if (addr < (caddr_t)v.v_ustart || addr >= (caddr_t)v.v_uend)
        return (-1);
    if ((addr = (char *)vtop(addr)) == (char *)-1)
        return(-1);
    saved_jb = nofault;
    if (!setjmp(jb)) {
        nofault = jb;
        *addr = byte;
        val = 0;
    } else
        val = -1;
    nofault = saved_jb;
    return(val);
}

/*
 * copyout - Move bytes out of the system into user's address space.
 */
copyout(from, to, nbytes)
caddr_t from, to;
int nbytes;
{
    int val;
    jmp_buf jb;
    int *saved_jb;

#ifdef DGETPUT
    printf("copyout: copying %d bytes to user space 0x%x (p=0x%x) from 0x%x\n",
        nbytes, to, vtop(to), from);
#endif
    if ((int)to+nbytes > v.v_uend || usraccess((long)to, RW) == 0 ||
        (to = (char *)vtop(to)) == (char *)-1) {
        printf("copyout failed\n");
        return(-1);
    }
    saved_jb = nofault;
    if (!setjmp(jb)) {
        nofault = jb;
        bvt(to, from, nbytes);
        val = 0;
    } else
        val = -1;
    nofault = saved_jb;
    return(val);
}

/*
 * copyin - Move bytes into the system space out of user's address space.
 */
copyin(from, to, nbytes)
caddr_t from, to;
int nbytes;
{
    int val;
    jmp_buf jb;
    int *saved_jb;

#ifdef DGETPUT
    printf("copyin: copying %d bytes from user space 0x%x (p=0x%x) to 0x%x\n",
        nbytes, from, vtop(from), to);
#endif
    if ((int)from+nbytes > v.v_uend || usraccess((long)from, RO) == 0 ||
}

```

```

    (from = (char *)vtop(from)) -- (char *)-1) {
        printf("copyin failed\n");
        return(-1);
    }
    saved_jb = nofault;
    if (!setjmp(jb)) {
        nofault = jb;
        blt(to, from, nbytes);
        val = 0;
    } else
        val = -1;
    nofault = saved_jb;
    return(val);
}

/*
 * copyseg - Copy one click's worth of data.
 */
copyseg(from, to)
int from, to;
{
    if (from == to)
        return;
    blt512(ctob(to), ctob(from), ctob(1)>>9);
}

/*
 * clearseg - Clear one click's worth of data.
 */
clearseg(wher)
int wher;
{
    /*
     * printf("clearseg 0x%x (0x%x)\n", wher, ctob(wher));
     */
    clear((caddr_t)ctob(wher), ctob(1));
}

/*
 * busaddr - Save the info from a bus or address error.
 */
/* VARARGS */
busaddr(frame)
struct {
    long regs[4]; /* d0,d1,a0,a1 */
    int baddr; /* bsr return address */
    short fcode; /* fault code */
    int aaddr; /* fault address */
    short ireg; /* instruction register */
} frame;
{
    u.u_fcode = frame.fcode;
    u.u_aaddr = frame.aaddr;
    u.u_ireg = frame.ireg;
}

/*
 * dophys - machine dependent set up portion of the phys system call
 */
dophys(phnum, laddr, bcount, phaddr)
unsigned phnum, laddr, phaddr;
register unsigned bcount;
{
    register struct phys *ph;
    register struct user *up;
    register unsigned addr;
    register i;

    up = &u;
    if (phnum >= v.v_phys)
        goto bad;
    ph = &up->u_phys[(short)phnum];
    ph->u_phladdr = 0;
    ph->u_phsize = 0;
    ph->u_phpaddr = 0;
    sureg();

    if (bcount == 0)
        return;
    /* valid logical address ? */
    addr = laddr;
    if (addr & (~SEGMASK))
        goto bad;
    i = ctos(btoc(bcount));
    while (i-- > 0) {
        if ((getmu((short *)vtoseg(addr) | ACCLIM) & PROTMASK) !=
            ASINVAL) {
#ifdef HOWFAR
            printf("addr=0x%x prot=0x%x\n",
                addr, *(short *)vtoseg(addr));
#endif
            goto bad;
            addr += (1<<SEGSSHIFT);
        }
        if ((ctos(btoc(v.v_uend))-ctos(btoc(v.v_ustart))-ctos(up->u_ptsize)-
            ctos(up->u_pdsiize)-ctos(up->u_pssiize)-ctos(btoc(bcount))) < 0)
            goto bad;
        ph->u_phladdr = laddr;
        ph->u_phsize = btoc(bcount);
        ph->u_phpaddr = phaddr;
        goto out;
    }
    bad:
    up->u_error = EINVAL;
    out:
    /* cxrelse(up->u_procp->p_context); */
    sureg();
}

/*
 * Scan phys array for physical I/O (raw I/O) transfer verification
 */
chkphys(base, limit)
{
    register struct phys *ph;
    register i;

    for (i = 0, ph = &u.u_phys[0]; i < v.v_phys; i++, ph++)
        if (ph->u_phsize != 0 && base >= ph->u_phladdr &&
            limit < ph->u_phladdr+ctob(ph->u_phsize))
            return(1);

    return(0);
}

/*
 * addupc - Take a profile sample.
 */
addupc(pc, p, incr)
unsigned pc;
register struct {
    short *pr_base;
    unsigned pr_size;
    unsigned pr_off;
    unsigned pr_scale;
} *p;
{
    union {
        int w_form; /* this is 32 bits on 68000 */
        short s_form[2];
    } word;
    register short *slot;

    slot = &p->pr_base[(((pc - p->pr_off) * p->pr_scale) >> 16) + 1]>>1];
    if ((caddr_t)slot >= (caddr_t)(p->pr_base) &&
        (caddr_t)slot < (caddr_t)((unsigned)p->pr_base + p->pr_size)) {
        if ((word.w_form = fuword((caddr_t)slot)) == -1)
            u.u_prof.pr_scale = 0; /* turn off */
        else {
            word.s_form[0] += (short)incr;
            (void) suword((caddr_t)slot, word.w_form);
        }
    }
}

```

```

/*
 * backup - Prepare for restart on a bus error.
 */
backup(ap)
register int *ap;
{
    register caddr_t pc;
    register ins;
    extern int tstb;

    pc = (caddr_t)ap[PC] - 2;
    ins = fuword((caddr_t)pc);
#ifdef HOWFAR
    printf("backup: pc, ins, tstb = %x, %x, %x.  ", pc, ins, tstb);
#endif
    if (ins & 0x8000 && (tstb & 0xFFFF0000) == (ins & 0xFFFF0000)) {
        ap[PC] = (int)pc;
        return(ins | 0xFFFF0000);
    }
    return(0);          /* signify we could not back up */
}

/*
 * sendsig - Simulate an interrupt.
 */
/* ARGSUSED */
sendsig(p, signo)
caddr_t p;
{
    register unsigned long n;
    register int *regp;
    short ps;

    regp = u.u_ar0;
    n = regp[SP] - 6;
    ps = (short)regp[RPS];
    (void) subyte((caddr_t)n, ps >> 8); /* high order byte of ps */
    (void) subyte((caddr_t)(n+1), ps); /* low order byte of ps */
    (void) suword((caddr_t)(n+2), regp[PC]);
    regp[SP] = n;
    regp[RPS] ^= ~PS_T;
    regp[PC] = (int)p;
}

/*ARGSUSED*/
clkset(oldtime)
time_t oldtime;
{
    /* use real time clock value instead of oldtime */
    time = rtcldoread();
}

/*
 * create a duplicate copy of a process
 */
procdup(p)
register struct proc *p;
{
    register a1, a2, n;

    n = p->p_size;
    if ((a2 = malloc(coremap, n)) == NULL)
        return(NULL);

    a1 = p->p_addr;
    p->p_addr = a2;
    while(n--)
        copyseg(a1++, a2++);
    return(1);
}

```

```

/* #define HOWFAR */

/* @(#)main.c 1.7 */
#include "sys/param.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/filsys.h"
#include "sys/mount.h"
#include "sys/proc.h"
#include "sys/inode.h"
#include "sys/seg.h"
#include "sys/conf.h"
#include "sys/buf.h"
#include "sys/iobuf.h"
#include "sys/tty.h"
#include "sys/var.h"
#include <sys/file.h>

/*
 * Initialization code.
 * Called from cold start routine as
 * soon as a stack and segmentation
 * have been established.
 * Functions:
 * clear and free user core
 * turn on clock
 * hand craft 0th process
 * call all initialization routines
 * fork - process 0 to schedule
 * - process 1 execute bootstrap
 *
 * loop at low address in user mode -- /etc/init
 * cannot be executed.
 */

int maxmem;
int minmem;
int physmem;
int cputype;
int *nofault;
struct inode *rootdir;

main(cudot)
{
    register struct user *up;
    register struct proc *p;
    register struct init_tbl *initp;
    extern char oemmsg[], timestamp[];
    extern int cmask, cdlimit;

    printf("(C) Copyright 1983 - UniSoft Corporation\n");
    printf("%d", cputype ? cputype : 68000);
    printf(" Unix System V - August 1983\n\n");
    printf("Created %s\n\n", timestamp);

    printf("%s\n", oemmsg);

    /*
     * set up system process
     */

    up = &u;
    p = &proc[0];
#ifdef NONSCATLOAD
    p->p_scat = 0;
#endif
    p->p_addr = cudot;
    p->p_size = v.v_usize;
    p->p_stat = SRUN;
    p->p_flag |= SLOAD|SSYS;
    p->p_nice = NZERO;

    up->u_proc = p;
    up->u_cmask = cmask;
    up->u_limit = cdlimit;
    up->u_stack[0] = STRMAGIC;

    startup(cudot);

    /*
     * initialize system tables at priority 7
     */
    for (initp = &init7_tbl[0]; initp->init_func; initp++) {
#ifdef HOWFAR
        printf("main calling %s\n", initp->init_msg);
#endif
        (*initp->init_func)();
    }

#ifdef HOWFAR
    printf("about to spl0\n");
#endif
    SPL0();
#ifdef HOWFAR
    printf("now at level 0\n");
#endif

    /*
     * initialize system tables at priority 0
     */
    for (initp = &init0_tbl[0]; initp->init_func; initp++) {
#ifdef HOWFAR
        printf("main calling %s\n", initp->init_msg);
#endif
        (*initp->init_func)();
    }

#ifdef HOWFAR
    printf("main calling iget\n");
#endif

    rootdir = iget(rootdev, ROOTINO);
    rootdir->i_flag &= ~ILOCK;
    up->u_cdir = iget(rootdev, ROOTINO);
    up->u_cdir->i_flag &= ~ILOCK;
    up->u_rdir = NULL;
    up->u_start = time;

    /*
     * create initial processes
     * start scheduling task
     */

#ifdef HOWFAR
    printf("main calling newproc\n");
#endif
    if (newproc(0)) {
#ifdef HOWFAR
        printf("main calling expand\n");
#endif
        expand((int)(v.v_usize + btoc(szicode)));
#ifdef HOWFAR
        printf("main calling estabur\n");
#endif
        (void) estabur((unsigned)0, (unsigned)btoc(szicode),
            (unsigned)0, 0, RO);
#ifdef HOWFAR
        printf("main calling copyout\n");
#endif
        (void) copyout((caddr_t)icode,
            (caddr_t)(v.v_ustart+v.v_doffset), szicode);
    /*
     * Return goes to loc. 0 of user init
     * code just copied out.
     */
#ifdef HOWFAR
    printf("main returning to icode\n");
#endif
    return;
}

```

```

    }
#ifdef HOWFAR
    printf("main calling sched\n");
#endif
    sched();
}

/*
 * iinit is called once (from main) very early in initialization.
 * It reads the root's super block and initializes the current date
 * from the last modified date.
 *
 * panic: iinit -- cannot read the super block.
 * Usually because of an IO error.
 */
iinit()
{
    register struct user *up;
    register struct buf *cp;
    register struct filsys *fp;
    struct inode iinode;

    up = &u;
    (*bdevsw[bmajor(rootdev)].d_open)(minor(rootdev), FREAD | FWRITE);
    (*bdevsw[bmajor(pipedev)].d_open)(minor(pipedev), FREAD | FWRITE);
    (*bdevsw[bmajor.swapdev]].d_open)(minor(swappedev), FREAD | FWRITE);
    cp = getebk();
    fp = cp->b_un.b_filsys;
    iinode.i_mode = IFBLK;
    iinode.i_rdev = rootdev;
    up->u_offset = SUPERBOFF;
    up->u_count = sizeof(struct filsys);
    up->u_base = (caddr_t)fp;
    up->u_segflg = 1;
    readi(&iinode);
    if (up->u_error)
        panic("iinit");
    mount[0].m_bufp = cp;
    mount[0].m_flags = MINUSE;
    mount[0].m_dev = brdev(rootdev);
    if (fp->s_magic != FSMAGIC)
        fp->s_type = Fs1b; /* assume old file system */
    if (fp->s_type == Fs2b)
        mount[0].m_dev |= Fs2BLK;
#ifdef FsTYPE == 4
    if (fp->s_type == Fs4b)
        mount[0].m_dev |= Fs4BLK;
#endif
    rootdev = mount[0].m_dev;
    if (brdev(pipedev) == brdev(rootdev))
        pipedev = rootdev;
    fp->s_flock = 0;
    fp->s_iloc = 0;
    fp->s_ronly = 0;
    fp->s_ninode = 0;
    fp->s_inode[0] = 0;

    clkset(fp->s_time);
}

/*
 * Initialize clist by freeing all character blocks.
 */
struct chead cfreelist;
cinit()
{
    register n;
    register struct cblock *cp;

    for(n = 0, cp = &cfree[0]; n < v.v_clist; n++, cp++) {
        cp->c_next = cfreelist.c_next;
        cfreelist.c_next = cp;
    }
    cfreelist.c_size = CLSIZE;
}

```

```

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

    dp = &bfreelist;
    dp->b_forw = dp; dp->b_back = dp;
    dp->av_forw = dp; dp->av_back = dp;
    b = ((long)buffers + (sizeof(int) - 1)) & (~sizeof(int));
    for (i=0, bp=buf; i<v.v_buf; i++,bp++) {
        bp->b_dev = NODEV;
        bp->b_un.b_addr = (caddr_t)b;
        b += SBUFSIZE;
        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 = 0;
        brelse(bp);
    }
    pfreelist.av_forw = bp = pbuf;
    for (; bp < &pbuf[(short)(v.v_pbuf-1)]; bp++)
        bp->av_forw = bp+1;
    bp->av_forw = NULL;
    for (i=0; i < v.v_hbuf; i++)
        hbuf[(short)i].b_forw = hbuf[(short)i].b_back = (struct buf *)&hbuf[(short)i];
}

```



```

/* @(#)malloc.c 1.1 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/system.h"
#include "sys/map.h"
#include "sys/var.h"
#include "sys/scat.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 swap map unit is 512 bytes.
 * Algorithm is first-fit.
 */
malloc(mp, size)
register struct map *mp;
{
    int n;

    if (mp != coremap)
        return(domall(mp, size));

    do
        if ((n = domall(mp, size)) != 0)
            return(n);
    while (xmrelease() != 0);
    return(0);
}

domall(mp, size)
struct map *mp;
register size;
{
    register struct map *bp;
    register a;

    for (bp = mapstart(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);
                mapsize(mp)++;
            }
            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, a)
struct map *mp;
register a;
{
    register struct map *bp;
    register t;

    bp = mapstart(mp);
    for (; bp->m_addr <= a && bp->m_size != 0; bp++);
    if (bp > mapstart(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;
            }
        }
    }
}

```

```

    }
    mapsize(mp)++;
}
} else {
    if (a+size == bp->m_addr && bp->m_size) {
        bp->m_addr -= size;
        bp->m_size += size;
    } else if (size) {
        if (mapsize(mp) == 0) {
            printf("\nDANGER: mfree map overflow at map 0x%x\n", mp);
            printf(" lost %d items starting at 0x%x\n", size, a);
            return;
        }
        do {
            t = bp->m_addr;
            bp->m_addr = a;
            a = t;
            t = bp->m_size;
            bp->m_size = size;
            bp++;
        } while (size = t);
        mapsize(mp)--;
    }
}

#ifdef NONSCATLOAD
/*
 * Wake scheduler when freeing core
 */
if (mp == coremap) {
    if (runin) {
        runin = 0;
        wakeup((caddr_t)&runin);
    }
} else
    if (mp != coremap)
#endif

/*
 * wakeup anyone waiting for a map
 */
if (mapwant(mp)) {
    mapwant(mp) = 0;
    wakeup((caddr_t)mp);
}

/*
 * return the largest size in the given map structure
 */
mallocl(mp)
struct map *mp;
{
    register struct map *bp;
    register a;

    a = 0;
    for (bp = mapstart(mp); bp->m_size; bp++)
        if (bp->m_size > a)
            a = bp->m_size;

    return(a);
}

#ifdef NONSCATLOAD
/*
 * malloc size clicks starting at address 'start'
 */
mallocl(mp, size, start)
struct map *mp;
register start;
{
    register struct map *bp;
    register a;

    for (bp = mapstart(mp); bp->m_size; bp++) {
        if (bp->m_addr == start && 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);
            mapsize(mp)++;
        }
        return(a);
    }
}
return(0);
}
#else
extern int nscatfree;

/*
 * allocate size units from the memory map
 */
memalloc(size)
{
    int n;

    while ((n = domemalloc(size)) == NULL)
        if (xmrlse() == 0)
            break;

    return(n);
}

domemalloc(size)
register size;
{
    register struct scatter *s;
    register short i;
    register al, a2;

    if (size <= 0) {
        printf("memalloc error: tried to allocate %d units\n", size);
        return(NULL);
    }
    if (size > nscatfree) {
        /* printf("memalloc: less than %d free pages at present\n",
            size); */
        return(NULL);
    }
    s = scatmap;
    al = a2 = scatfreelist.sc_index;
    if (size > 1) {
        i = size - 2;
        do {
            al = s[al].sc_index;
        } while (--i != -1);
    }
    scatfreelist.sc_index = s[al].sc_index;
    s[al].sc_index = SCATEND;
    nscatfree -= size;
    /* printf("memalloc: found %d free pages starting at index %d\n",
        size, a2); */
    return(a2);
}

/*
 * allocate size contiguous units from the memory map
 */
cmemalloc(size)
{
    int n;

    do
        scatsort();
    while ((n = docmemalloc(size)) == NULL && xmrlse());
    /* printf("cmemalloc: size=%d starting address=0x%x\n", size, n); */
    return(n);
}
docmemalloc(size)

```

```

register size;
{
    register struct scatter *s, *ss;
    register short i;
    register al, a2, n;

    if (size <= 0) {
        printf("cmemalloc error: tried to allocate %d units\n", size);
        return(NULL);
    }
    if (size > nscatfree)
        return(NULL);
    s = scatmap;
    ss = &scatfreelist;
    al = ss->sc_index;
    for (;;) {
        n = memcontig(al, size);
        if (n >= size)
            break;
        if (n > 1) {
            i = n - 2;
            do
                al = s[al].sc_index;
            while (--i != -1);
        }
        ss = &s[al];
        if (ss->sc_index == SCATEND)
            return(NULL);
        al = ss->sc_index;
    }
    a2 = al;
    if (size > 1) {
        i = size - 2;
        do
            al = s[al].sc_index;
        while (--i != -1);
    }
    ss->sc_index = s[al].sc_index;
    s[al].sc_index = SCATEND;
    nscatfree -= size;
    /* printf("cmemalloc: found %d free pages starting at index %d\n",
        size, a2); */
    if (countscat(a2) != size)
        printf("cmemalloc:improper allocation countscat=%d size=%d\n",
            countscat(a2), size);

    return(a2);
}

/*
 * free memory map chain
 */
memfree(a)
register a;
{
    register struct scatter *s;
    register i, al, a2;

    if (a <= 0 || a >= v.v_nscatload) {
        printf("memfree:illegal index %d (0x%x)\n", a, a);
        return;
    }
    i = 1;
    s = scatmap;
    al = a;
    while ((a2 = s[al].sc_index) != SCATEND) {
        al = a2;
        i++;
    }
    /* printf("memfree:%d units freed starting at %d\n", i, a); */
    s[al].sc_index = scatfreelist.sc_index;
    scatfreelist.sc_index = a;
    nscatfree += i;
    /*
     * Wake scheduler when freeing memory
     */
    if (runin) {

```

```

        runin = 0;
        wakeup((caddr_t)&runin);
    }
}

/*
 * find number of contiguous memory pages
 */
memcontig(sindex, ct)
register sindex, ct;
{
    register struct scatter *s;
    register al, n;

    if (sindex == SCATEND || ct <= 0) {
        printf("memcontig:sindex=0x%x ct=%d\n", sindex, ct);
        return(0);
    }
    n = 1;
    s = scatmap;
    al = ixtoc(sindex);
    while (--ct > 0) {
        if ((sindex = s[sindex].sc_index) == SCATEND)
            break;
        if (++al != ixtoc(sindex))
            break;
        n++;
    }
    return(n);
}

/*
 * sort the scatter load map
 */
scatsort()
{
    register struct scatter *s, *sf;
    register int j, k, n, *ip, *jp;
    register short i;

    /*
     * clear scatter sort array
     */
    ip = scsortmap;
    i = ((v.v_nscatload+31) >> 5) - 1;
    do
        *ip++ = 0;
    while (--i != -1);
    /*
     * build bit map of free pages
     */
    s = scatmap;
    sf = &scatfreelist;
    ip = scsortmap;
    for (j = sf->sc_index; j != SCATEND; j = s[j].sc_index)
        ip[j>>5] |= 1 << (j&31);
    /*
     * rebuild freelist
     */
    n = 0;
    sf->sc_index = SCATEND;
    j = ((v.v_nscatload+31) >> 5) - 1;
    jp = &scsortmap[j];
    for (; j >= 0; j--, jp--) {
        if (*jp == 0)
            continue;
        for (k=31; k>=0; k--) {
            if (*jp & (1<<k)) {
                n++;
                i = sf->sc_index;
                sf->sc_index = (j << 5) + k;
                s[(j << 5) + k].sc_index = i;
            }
        }
    }
    nscatfree = n;
}
#endif

```

```

/* mbuf.c 1.36 82/06/20 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/mmu.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/errno.h"
#include "sys/user.h"
#include "sys/proc.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "sys/map.h"
#include "net/in_systm.h" /* XXX */

/* some random constants, ints */

/* THIS SHOULD BE AN EVEN CLICK NUMBER OF BYTES !!! */
/*
#define IOSIZE 8192 /* area for DMA buffers */
#define IOSIZE 0 /* no space for DMA buffers for multibus */

/* unsigned int miosize = IOSIZE; */
unsigned int miobase; /* loc of DMA area. */

/*
 * Initialize the buffer pool. Called from netinit/main.
 */
mbinit()
{
    register i;
    register struct mbuf *m;
    extern struct mbuf * mballoc();

    /* allocate mbuf io area. iomalloc is machine-dependent... */
    /* miobase = mbioalloc(btoc(miosize)); */
    miobase = mbioalloc();
    /* link the mbufs */
    m = mballoc();
    /*
    printf("mbufs at %x\n", m);
    */
    mstat.m_mbufs = NMBUFS;

    for(i=0 ; i<NMBUFS ; i++) {
        m->m_off = 0;
        m->m_free = 0;
        bzero((char *)m, MSIZE);
        (void) m_free(m);
        m++;
    }
}

/* NEED SOME WAY TO RELEASE SPACE */

/*
 * Space allocation routines.
 * These are also available as macros
 * for critical paths.
 */
/* ARGSUSED */
struct mbuf *
m_get(canwait)
    int canwait;
{
    register struct mbuf *m;
#ifdef PRMDEF
    struct call {
        long local;

```

```

        long frmprtr;
        long raddr;
    };
    int i;
    register struct call * cp = &i;
    extern dog_pr;
    register olldogpr = dog_pr;

    if (dog_pr) (printf("<get from %x>",cp->raddr); dog_pr = 0;
#endif

#ifdef MGET(m, canwait);
#ifdef PRMDEF
    if (olldogpr)
        dog_pr = 1;
#endif
#endif
    return (m);
}

struct mbuf *
m_free(m)
    struct mbuf *m;
{
    register struct mbuf *n;

    MFREE(m, n);
    return (n);
}

m_freem(m)
    register struct mbuf *m;
{
    register struct mbuf *n;
    register int s;

    if (m == NULL)
        return;
    s = splimp();
    do {
        MFREE(m, n);
    } while (m = n);
    splx(s);
}

/*
 * Mbuffer utility routines.
 */
struct mbuf *
m_copy(m, off, len)
    register struct mbuf *m;
    int off;
    register int len;
{
    register struct mbuf *n, **np;
    struct mbuf *top;

    if (len == 0)
        return (0);
    if (off < 0 || len < 0)
        panic("m_copy1");
    while (off > 0) {
        if (m == 0)
            panic("m_copy2");
        if (off < m->m_len)
            break;
        off -= m->m_len;
        m = m->m_next;
    }
    MAPSAVE();
    np = &top;
    top = 0;
    while (len > 0) {
        if (m == 0) {
            if (len != M_COPYALL)
                panic("m_copy3");
            break;

```

```

    }
    MGET(n, 1);
    *np = n;
    if (n == 0)
        goto nospace;
    n->m_len = MIN(len, m->m_len - off);
    {
        n->m_off = MMINOFF;
        MBCOPY(m, off, n, 0, (unsigned)n->m_len);
    }
    if (len != M_COPYALL)
        len -= n->m_len;
    off = 0;
    m = m->m_next;
    np = &n->m_next;
}
goto out;
nospace:
    m_freem(top);
    top = 0;
out:
    MAPREST();
    return (top);
}

m_cat(m, n)
register struct mbuf *m, *n;
{
    while (m->m_next)
        m = m->m_next;
    while (n) {
        if (m->m_off >= MMAXOFF ||
            m->m_off + m->m_len + n->m_len > MMAXOFF) {
            /* just join the two chains */
            m->m_next = n;
            return;
        }
        /* splat the data from one into the other */
        MBCOPY(n, 0, m, m->m_len, (u_int)n->m_len);
        m->m_len += n->m_len;
        n = m_free(n);
    }
}

m_adj(mp, len)
struct mbuf *mp;
register int len;
{
    register struct mbuf *m, *n;

    if ((m = mp) == NULL)
        return;
    if (len >= 0) {
        while (m != NULL && len > 0) {
            if (m->m_len <= len) {
                len -= m->m_len;
                m->m_len = 0;
                m = m->m_next;
            } else {
                m->m_len -= len;
                m->m_off += len;
                break;
            }
        }
    } else {
        /* a 2 pass algorithm might be better */
        len = -len;
        while (len > 0 && m->m_len != 0) {
            while (m != NULL && m->m_len != 0) {
                n = m;
                m = m->m_next;
            }
            if (n->m_len <= len) {
                len -= n->m_len;
                n->m_len = 0;
            } else {
                n->m_len -= len;
                n->m_off += len;
                break;
            }
        }
    }
}

} else {
    m = mp;
    } else {
        n->m_len -= len;
        break;
    }
}

struct mbuf *
m_pullup(m0, len)
struct mbuf *m0;
int len;
{
    register struct mbuf *m, *n;
    int count;

    n = m0;
    if (len > MLEN)
        goto bad;
    MGET(m, 0);
    if (m == 0)
        goto bad;
    m->m_off = MMINOFF;
    m->m_len = 0;
    do {
        count = MIN(MLEN - m->m_len, len);
        if (count > n->m_len)
            count = n->m_len;
        MBCOPY(n, 0, m, m->m_len, (u_int)count);
        len -= count;
        m->m_len += count;
        n->m_off += count;
        n->m_len -= count;
        if (n->m_len)
            break;
        n = m_free(n);
    } while (n);
    if (len) {
        (void) m_free(m);
        goto bad;
    }
    m->m_next = n;
    return (m);
}

bad:
    m_freem(m);
    return (0);
}

#endif notdef
/*
 * Allocate a contiguous buffer for DMA IO. Called from if_ubainit().
 * TODO: fix net device drivers to handle scatter/gather to mbufs
 * on their own; thus avoiding the copy to/from this area.
 */
unsigned int
m_ioget(size)
{
    unsigned int base;

    size = ((size + 077) & ~077); /* round up byte size */
    if (size > miosize) return(0);
    miosize -= size;
    base = miobase;
    miobase += size;
    return(base);
}

#endif debug
mbprint(m, s)
register struct mbuf *m;
char *s;
{
    extern enprint;
    register char *ba;
}

```

```

int col,i,bc;

if (enprint == 0) return;
MAPSAVE();
nprintf("MB %s\n",s);
for (;;) {
    if (m == 0) break;
    ba = mtod(m, char *);
    col = 0; bc = m->m_len;
    nprintf("m% next% off% len% act% free%\n",
        m, m->m_next, m->m_off, m->m_len, m->m_click,
        m->m_free);
    for(; bc ; bc--) {
        i = *ba++ & 0377;
        nprintf("%o ",i);
        if(++col > 31) {
            col = 0;
            nprintf("\n ");
        }
    }
    nprintf("\n");
    m = m->m_next;
}
MAPREST();
}

#else
mbprint(m,s)
register struct mbuf *m;
char *s;
{
#ifdef lint
    mbprint(m, s);
#endif
#ifdef debug
}
#endif debug

#ifdef notdef
mcheck(m, msg)
struct mbuf *m;
char * msg;
{
    extern char mbufbufs[];
    extern struct mbuf * mfreep;
    register x, ret = 0;

    x = spl7();
    if ( (m < (struct mbuf *) &mbufbufs[0]) && (m != 0) ||
        (m > (struct mbuf *) &mbufbufs[(NMBUFS+1)*MSIZE]) ||
        ((mfreep < (struct mbuf *) &mbufbufs[0]) && (mfreep != 0)) ||
        (mfreep > (struct mbuf *) &mbufbufs[(NMBUFS+1)*MSIZE]) ) {
        printf ("mcheck fail; m, mfreep = %x, %x, from %s\n",
            m,mfreep,msg);
        ret = 1;
    }
    splx(x);
    return ret;
}
#endif

#ifdef PRMDEF
struct call {
    long    local;
    long    frmprtr;
    long    raddr;
};

int dog_pr;
int dof_pr;

prmget()
{
    int i;
    register struct call * cp = &i;

    if (dog_pr) printf("<MGET from %x>",cp->raddr);
}

```

```

/*
 * mem, kmem and null devices.
 *
 * Memory special file
 * minor device 0 is physical memory
 * minor device 1 is kernel memory
 * minor device 2 is EOF/RATHOLE
 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/buf.h"
#include "sys/system.h"
#include "setjmp.h"

/*
 * mmread - read mem, kmem or null.
 */
mmread(dev)
{
    if (minor(dev) != 2) /* not /dev/null */
        mmmove(dev, B_READ);
}

/*
 * mmwrite - write mem, kmem or null.
 */
mmwrite(dev)
{
    if (minor(dev) == 2) { /* /dev/null, just gobble chars */
        u.u_count = 0;
        return;
    }
    mmmove(dev, B_WRITE);
}

/*
 * mmmove - common routine for mmread and mmwrite
 */
mmmove(dev, flag)
dev_t dev;
{
    register int pageoffs, count, prot;
    jmp_buf jb;
    int *saved_jb;
    int s;

    if (minor(dev) == 1 || minor(dev) == 0) { /* kmem, mem */
        do {
            s = spl7();
            SEG1_0 = 1; /* system context */
            /* SEG2_0 = 1; /* system context */
            prot = getmmu((short *) (vtoseg(u.u_offset) | ACCLIM)) & PROTMASK;
            SEG1_1 = 1; /* user context */
            /* SEG2_0 = 1; /* user context */
            splx(s);
        } while (1);
        /* printf("u_base=0x%x offset=0x%x prot=0x%x\n", u.u_base, u.u_offset, prot); */
        if ((unsigned)u.u_offset < (unsigned)STDIO && prot != ASRW)
            goto bad;
        pageoffs = u.u_offset & (ctob(1)-1);
        count = min((unsigned)(ctob(1) - pageoffs), u.u_count);
        /* printf("pageoffs=%d count=%d u.u_count=%d\n", pageoffs, count, u.u_count); */
        saved_jb = nofault;
        if (!setjmp(jb)) {
            nofault = jb;
            u.u_segflg = 0;
            iomove((caddr_t)u.u_offset, count, flag);
        } else
            u.u_error = ENXIO;
    }
}

    u.u_segflg = 0;
    nofault = saved_jb;
} while(u.u_error == 0 && u.u_count);
return;
}
bad:
u.u_error = ENXIO;
}

```

```
/*
 * mkfspm device
 * Get the size of the Priam disk "device" and make a root
 * filesystem in partition 0. This makes a filesystem using the
 * entire disk except for the boot and swap areas.
 *
 * The raw device is specified, but both the block and character
 * devices are expected to be available (created). The filesystem
 * is made on /dev/pm?a where /dev/rpm?a is the argument.
 */

#include "stdio.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/uioc1.h"
#include "sys/stat.h"
#include "sys/swapsz.h"

main(argc, argv)
char **argv;
{
    register fp;
    register char *device;
    register char *cp;
    int pmsize;
    char spmsize[10];

    device = *(argv+1);
    if ((fp = open(device, 2)) < 0)
        goto out;
    if (ioctl(fp, UIOCSIZE, (caddr_t)&pmsize) < 0)
        goto out;
    cp = (char *)strchr(device, 'r'); /* cp points to 'r' in /dev/rpm?a */
    for ( ; *cp; cp++) /* change to /dev/pm?a */
        *cp = *(cp+1);
    sprintf(spmsize, "%d", pmsize-PMNSWAP-101);
    printf("%s: %s blocks\n", device, spmsize);
    execl("/etc/mkfs1b", "mkfs1b", device, spmsize, 0);
out:
    perror("mkfspm");
    exit(1);
}
```



```

/*
 * (C) 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with Unisoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by Unisoft.
 *
 * Mouse Driver
 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/proc.h"
#include "sys/errno.h"
#include "sys/file.h"
#include "sys/tty.h"
#include "sys/termio.h"
#include "sys/conf.h"
#include "sys/sysinfo.h"
#include "sys/var.h"
#include "sys/reg.h"
#include "sys/buf.h"
#include "setjmp.h"
#include "sys/cops.h"
#include "sys/local.h"
#include "sys/elog.h"
#include "sys/ms.h"
#include "sys/mouse.h"
#include "sys/mmu.h"
#include "sys/kb.h"
#include "sys/al_ioctl.h"

#define splms spl2

/* flags local to ms.c */
char mouseopen; /* active flag */
char msblkd; /* flag to sleep on when blocked */

/* also used in ll.c (vertical retrace interrupt code) */
struct msparms mparm; /* place to save ioctl data */
struct proc msproc; /* controlling process for mouse */
char msvrmsk; /* mask of low bits of retrace counter */

/* values set in kb.c */
extern char ms_plg, ms_btn; /* mouse plugged-in and button state */
extern short ms_row, ms_col; /* last received row and column */
extern time_t lbolt;

#define QUELEN 128
typedef unsigned long mem_t;
int mqlen;
mem_t *mqlo, *mqhi; /* high and low pointers into ... */
mem_t mqbuf[QUELEN]; /* ring buffer for mouse data */
mem_t mtim[QUELEN];

msopen(dev, flag)
dev_t dev;
{
    if (dev != 0) { /* minor device number is wrong */
        u.u_error = ENXIO;
        return;
    }
    if (ms_plg) { /* mouse is not plugged in */
        u.u_error = ENODEV;
        return;
    }
    if (flag != 1) { /* open for writing !! */
        u.u_error = EINVAL;
        return;
    }
}

```

```

if (u.u_ttyd != CONSOLE) { /* Controlling tty not bitmap */
    u.u_error = EPERM;
    return;
}
if (mouseopen++ > 0) { /* already opened */
    u.u_error = EBUSY;
    return;
}
splms();
msproc = u.u_proc;
mqhi = mqlo = mqbuf;
mqlen = 0;
mparm.mp_irate = 28;
mparm.mp_fdlay = 300;
mparm.mp_flags = MF_BUT;
l2copscmd(MOUSEON);
spl0();
}

/*
 * Reset everything since code elsewhere (in ll.c) uses it.
 */
/* ARGSUSED */
msclose(dev, flag)
{
    if (mouseopen <= 0)
        u.u_error = EINVAL;
    else {
        SPL6();
        mouseopen = 0; /* only accessed in ms.c */
        mqhi = mqlo = mqbuf;
        mqlen = 0;
        mparm.mp_irate = 0;
        mparm.mp_fdlay = 0;
        mparm.mp_flags = 0;
        msvrmsk = 0;
        l2copscmd(MOUSEOFF);
        SPL0();
    }
}

msstrategy(bp)
register struct buf *bp;
{
    register mem_t *rp; /* reciever buffer pointer */
    register int i; /* record count */

    /* Make sure the mouse is plugged in and the read request is for a
     * multiple of the record size.
     */
    if (ms_plg) { /* mouse unplugged */
        u.u_error = ENODEV;
        goto fail;
    }
    if (bp->b_bcount % 3) { /* count not multiple of record size */
        u.u_error = EINVAL;
fail:
        bp->b_resid = bp->b_bcount;
        iodone(bp);
        return;
    }
    if (mqlo == mqhi) { /* queue empty */
        if ((int)mparm.mp_irate <= 0) /* mouse shutdown */
            u.u_error = EIO;
        splms();
        if (mparm.mp_flags & MF_BLK) { /* block till record avail */
            msblkd = 1;
            while (msblkd)
                (void) sleep((caddr_t)&msblkd, TTIPRI);
            goto ok;
        }
        spl0();
        goto fail;
    }
    splms(); /* so msintr doesnt screw up que ptrs */
}

```

```

ok:
if (lbolt - mtim[mqlo-mqbuf] >= mparm.mp_fdelay) {
    if (mparm.mp_fdelay)
        msintr(M_FLUSH);
}
rp = (mem t *)bp->b_un.b_addr; /* start of receiver buffer */
for (i=bp->b_bcount; i>0 && mqlo != mqhi; i -- 4) {
    *rp++ = *mqlo++;
    if (mqlo >= mqbuf + QUELEN) mqlo = mqbuf;
}
bp->b_resid = i;
mqlen -- (bp->b_bcount - i) >> 2;
spl0();
iodone(bp);
return;
}

struct buf rmsbuf; /* Buffer for raw input-output */
msread(dev)
{
    if (msproc != u.u_procp) {
        u.u_error = EIO;
        return;
    }
    rmsbuf.b_flags &= ~B_BUSY;
    physio(msstrategy, &rmsbuf, dev, B_READ);
}

msintr(kind)
{
    register struct msrecord *mp;
    union {
        struct msrecord mf_rec;
        unsigned long mf_long;
    } msr;

    if (!mouseopen || mparm.mp_irate == 0) /* mouse not active */
        return;

    if (ms_plg) {
        mqlo = mqhi = mqbuf;
        mqlen = 0;
        return;
    }

    msr.mf_long = 0; /* clear new record */
    mp = &msr.mf_rec; /* get pointer to record structure */

    switch (kind) { /* kind of interrupt */
    case M_FLUSH: /* auto flush (called from msintr,msstrategy) */
        mp->mr_but = 1;
    case M_PLUG: /* automatic reset (called from kbintr) */
        mp->mr_reset = 1;
        mqlo = mqhi = mqbuf;
        mqlen = 1;
        mtim[0] = lbolt;
        *mqhi++ = msr.mf_long;
        return;
    case M_BUT: /* mouse button changed (called from kbintr) */
        if ((mparm.mp_flags & MF_BUT) == 0)
            return;
        break;
    case M_CTL: /* control key changed (called from kbintr) */
        if ((mparm.mp_flags & MF_CTL) == 0)
            return;
        break;
    case M_SFT: /* shift key changed (called from kbintr) */
        if ((mparm.mp_flags & MF_SFT) == 0)
            return;
        break;
    }

    if (mqlen) { /* check for invalid queue data */
        if (lbolt - mtim[mqlo-mqbuf] >= mparm.mp_fdelay) {
            if (mparm.mp_fdelay)
                msintr(M_FLUSH);
            if (mqlen >= QUELEN)
                return;
        } else { /* queue was empty */
            if (mparm.mp_flags & MF_SIG)
                psignal(msproc, SIGMOUS);
        }

        mp->mr_but = ms_btn ? 1 : 0;
        mp->mr_ctl = kb_ctrl ? 1 : 0;
        mp->mr_sft = kb_shift ? 1 : 0;
        mp->mr_time = lbolt & 0xFFF;
        if (kind == M_MOVE) { /* called from kbintr */
            mp->mr_row = ms_row;
            mp->mr_col = ms_col;
        }
        mtim[mqhi - mqbuf] = lbolt;
        *mqhi++ = msr.mf_long;
        mqlen++;
        if (mqhi >= mqbuf + QUELEN)
            mqhi = mqbuf;
        if (msblkd) {
            msblkd = 0;
            wakeup((caddr_t)&msblkd);
        }
    }

    /* ARGSUSED */
    msioctl(dev, cmd, addr, flag)
    dev_t dev;
    int cmd;
    caddr_t addr;
    int flag;
    {
        register struct msparms *mp = &mparm;
        int dlay, rate;

        switch (cmd) {
        case AL_GMOUSE:
            if (mp->mp_qlen == mqlen) {
                mp->mp_otime = lbolt - mtim[mqlo-mqbuf];
                mp->mp_ytime = lbolt - mtim[mqhi-mqbuf-1];
            } else {
                mp->mp_otime = 0;
                mp->mp_ytime = 0;
            }
            if (copyout((char *)mp, addr, sizeof(struct msparms)))
                u.u_error = EFAULT;
            break;
        case AL_SMOUSE:
            dlay = mp->mp_fdelay;
            rate = mp->mp_irate;
            if (copyin(addr, (char *)mp, sizeof(struct msparms)))
                u.u_error = EFAULT;
            else {
                if (mp->mp_fdelay) {
                    if (mp->mp_fdelay < 20) {
                        splms();
                        mp->mp_fdelay = dlay; /* restore */
                        mqlen = 0;
                        mqhi = mqlo = mqbuf;
                        spl0();
                    } else if (mp->mp_fdelay > 4095)
                        mp->mp_fdelay = 4095;
                }
                if (mp->mp_irate != rate) { /* did int rate chg */
                    mp->mp_irate = (mp->mp_irate + 3) & -3;
                    if (mp->mp_irate > 28)
                        mp->mp_irate = 28;
                    splms();
                    if (mp->mp_irate)
                        l2copscmd((char) ((MOUSEON&-7) | (mp->mp_irate>>2)));
                    else
                        l2copscmd(MOUSEOFF);
                    spl0();
                }
            }
        }
    }
}

```

```
        }
        splms();
        if (mp->mp_flags & MF_VRT)
            msvrnsk = mp->mp_flags & MF_VRATE;
        spl0();
    }
    break;
default:
    u.u_error = ENOTTY;
}
}
```

```

/* @(#)msg.c 1.3 */
/*
**      Inter-Process Communication Message Facility.
*/

#include "sys/types.h"
#include "sys/param.h"
#include "sys/dir.h"
#ifdef u3b
#include "sys/istk.h"
#endif
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/seg.h"
#include "sys/proc.h"
#include "sys/buf.h"
#include "sys/errno.h"
#include "sys/map.h"
#include "sys/ipc.h"
#include "sys/msg.h"
#include "sys/system.h"
#ifdef vax
#include "sys/page.h"
#endif
#ifdef u3b
#include "sys/macro.h"
#else
#include "sys/sysmacros.h"
#endif

extern struct map      msgmap[];      /* msg allocation map */
extern struct msqid_ds msgque[];      /* msg queue headers */
extern struct msg      msgh[];        /* message headers */
extern struct msginfo  msginfo;       /* message parameters */
extern char            msgspace[];    /* space for message buffers */
struct msg             *msgfp; /* ptr to head of free header list */
paddr_t               msg;           /* base address of message buffer */
extern time_t         time;          /* system idea of date */

struct ipc_perm        *ipcget();
struct msqid_ds        *msgconv();

/* Convert bytes to msg segments. */
#define btoq(X) ((X + msginfo.msgssz - 1) / msginfo.msgssz)

/* Choose appropriate message copy routine. */
#ifdef pdp11
#define MOVE            msgpimove
#else
#define MOVE            iomove
#endif

/*
**      msgconv - Convert a user supplied message queue id into a ptr to a
**      msqid_ds structure.
*/

struct msqid_ds *
msgconv(id)
register int id;
{
    register struct msqid_ds *qp; /* ptr to associated q slot */

    qp = &msgque[(short)(id % msginfo.msgmni)];
    if((qp->msg_perm.mode & IPC_ALLOC) == 0 ||
        id / msginfo.msgmni != qp->msg_perm.seq) {
        u.u_error = EINVAL;
        return(NULL);
    }
    return(qp);
}

/*
**      msgctl - Msgctl system call.
*/

```

```

msgctl()
{
    register struct a {
        int          msgid,
                  cmd;
        struct msqid_ds *buf;
    } *uap = (struct a *)u.u_ap;
    struct msqid_ds ds; /* queue work area */
    register struct msqid_ds *qp; /* ptr to associated q */

    if((qp = msgconv(uap->msgid)) == NULL)
        return;
    u.u_rvall = 0;
    switch(uap->cmd) {
    case IPC_RMID:
        if(u.u_uid != qp->msg_perm.uid && u.u_uid != qp->msg_perm.cuid
            && !suser())
            return;
        while(qp->msg_first)
            msgfree(qp, (struct msg *)NULL, qp->msg_first);
        qp->msg_cbytes = 0;
        if(uap->msgid + msginfo.msgmni < 0)
            qp->msg_perm.seq = 0;
        else
            qp->msg_perm.seq++;
        if(qp->msg_perm.mode & MSG_RWAIT)
            wakeup((caddr_t)&qp->msg_qnum);
        if(qp->msg_perm.mode & MSG_WWAIT)
            wakeup((caddr_t)qp);
        qp->msg_perm.mode = 0;
        return;
    case IPC_SET:
        if(u.u_uid != qp->msg_perm.uid && u.u_uid != qp->msg_perm.cuid
            && !suser())
            return;
        if(copyin((caddr_t)uap->buf, (caddr_t)&ds, sizeof(ds))) {
            u.u_error = EFAULT;
            return;
        }
        if(ds.msg_qbytes > qp->msg_qbytes && !suser())
            return;
        qp->msg_perm.uid = ds.msg_perm.uid;
        qp->msg_perm.gid = ds.msg_perm.gid;
        qp->msg_perm.mode = (qp->msg_perm.mode & ~0777) |
            (ds.msg_perm.mode & 0777);
        qp->msg_qbytes = ds.msg_qbytes;
        qp->msg_ctime = time;
        return;
    case IPC_STAT:
        if(ipcaccess(&qp->msg_perm, MSG_R))
            return;
        if(copyout((caddr_t)qp, (caddr_t)uap->buf, sizeof(*qp))) {
            u.u_error = EFAULT;
            return;
        }
        return;
    default:
        u.u_error = EINVAL;
        return;
    }
}

/*
**      msgfree - Free up space and message header, relink pointers on q,
**      and wakeup anyone waiting for resources.
*/

msgfree(qp, pmp, mp)
register struct msqid_ds *qp; /* ptr to q of msg being freed */
register struct msg *mp; /* ptr to msg being freed */
register struct pmp *pmp; /* ptr to mp's predecessor */
{
    /* Unlink message from the q. */
    if(pmp == NULL)
        qp->msg_first = mp->msg_next;
    else

```

```

        pmp->msg_next = mp->msg_next;
if(mp->msg_next == NULL)
    qp->msg_last = pmp;
qp->msg_qnum--;
if(qp->msg_perm.mode & MSG_WWAIT) {
    qp->msg_perm.mode &= ~MSG_WWAIT;
    wakeup((caddr_t)qp);
}

/* Free up message text. */
if(mp->msg_ts)
    mfree(msgmap, btoq(mp->msg_ts), mp->msg_spot + 1);

/* Free up header */
mp->msg_next = msgfp;
if(msgfp == NULL)
    wakeup((caddr_t)&msgfp);
msgfp = mp;
}

/*
** msgget - Msgget system call.
*/

msgget()
{
    register struct a {
        key_t key;
        int msgflg;
    }
    *uap = (struct a *)u.u_ap;
    register struct msqid_ds *qp; /* ptr to associated q */
    int s; /* ipcget status return */

    if((qp = (struct msqid_ds *)
        ipcget(uap->key, uap->msgflg, (struct ipc_perm *)msgque,
        msginfo.msgmni, sizeof(*qp), &s)) == NULL)
        return;

    if(s) {
        /* This is a new queue. Finish initialization. */
        qp->msg_first = NULL; qp->msg_last = NULL;
        qp->msg_qnum = 0;
        qp->msg_qbytes = msginfo.msgqmb;
        qp->msg_lspid = 0; qp->msg_lrpid = 0;
        qp->msg_stime = 0; qp->msg_rtime = 0;
        qp->msg_ctime = time;
    }
    u.u_rval1 = qp->msg_perm.seq * msginfo.msgmni + (qp - msgque);
}

/*
** msginit - Called by main(main.c) to initialize message queues.
*/

msginit()
{
    register int i; /* loop control */
    register struct msg *mp; /* ptr to msg begin linked */
#ifdef mc68000
    register int bs; /* message buffer size */
#endif

    /* Allocate physical memory for message buffer. */
#ifdef mc68000
    bs = (long)msginfo.msgseg * msginfo.msgsz;
#endif
#ifdef mc68000
    msg = (paddr_t) msgspace;
#endif
#ifdef pdp11
    if((msg = (paddr_t)ctob((long)(unsigned)malloc(coremap,
        bs=(int)btoc((long)msginfo.msgseg * msginfo.msgsz)))) == 0) {
#endif
#ifdef vax
    if((msg = (paddr_t)sptalloc(bs=btoc(msginfo.msgseg * msginfo.msgsz),
        PG_V | PG_KW, 0)) == NULL) {

```

```

#endif
#ifdef u3b
    if((msg = (paddr_t)kseg(RW, btop(msginfo.msgseg * msginfo.msgsz))) ==
        NULL) {
#endif
#ifdef mc68000
        printf("Can't allocate message buffer.\n");
        msginfo.msgseg = 0;
    }
#endif
    mapinit(msgmap, msginfo.msgmap);
    mfree(msgmap, (int)msginfo.msgseg, 1);
    for(i = 0, mp = msgfp = msgh; ++i < msginfo.msgtbl; mp++)
        mp->msg_next = mp + 1;
#ifdef vax
    maxmem -= bs;
#endif
#ifdef mc68000
    return;
#endif
#ifdef pdp11
    return(bs);
#endif
}

#ifdef pdp11
/*
** msgpimove - PDP 11 pimove interface for possibly large copies.
*/

msgpimove(base, count, mode)
paddr_t base; /* base address */
register unsigned count; /* byte count */
int mode; /* transfer mode */
{
    register unsigned tcount; /* current transfer count */

    while(u.u_error == 0 && count) {
        tcount = count > 8064 ? 8064 : count;
        pimove(base, tcount, mode);
        base += tcount;
        count -= tcount;
    }
}
#endif

/*
** msgrcv - Msgrcv system call.
*/

msgrcv()
{
    register struct a {
        int msqid;
        struct msgbuf *msgp;
        int msgsz;
        long msgtyp;
        int msgflg;
    }
    *uap = (struct a *)u.u_ap;
    register struct msg *mp, /* ptr to msg on q */
        *pmp, /* ptr to mp's predecessor */
        *smp, /* ptr to best msg on q */
        *spmp; /* ptr to smp's predecessor */
    register struct msqid_ds *qp; /* ptr to associated q */
    int sz; /* transfer byte count */

    if((qp = msgconv(uap->msqid)) == NULL)
        return;
    if(ipcaccess(&qp->msg_perm, MSG_R))
        return;
    if(uap->msgsz < 0) {
        u.u_error = EINVAL;
        return;
    }
    smp = NULL; spmp = NULL;
}
findmsg:

```

```

pmp = NULL;
mp = qp->msg_first;
if(uap->msgtyp == 0)
    smp = mp;
else
    for(;;mp;pmp = mp, mp = mp->msg_next) {
        if(uap->msgtyp > 0) {
            if(uap->msgtyp != mp->msg_type)
                continue;
            smp = mp;
            spmp = pmp;
            break;
        }
        if(mp->msg_type <= -uap->msgtyp) {
            if(smp && smp->msg_type <= mp->msg_type)
                continue;
            smp = mp;
            spmp = pmp;
        }
    }
if(smp) {
    if(uap->msgsz < smp->msg_ts)
        if(!(uap->msgflg & MSG_NOERROR)) {
            u.u_error = EZBIG;
            return;
        } else
            sz = uap->msgsz;
    else
        sz = smp->msg_ts;
    (void) copyout((caddr_t)smp->msg_type, (caddr_t)uap->msgp,
        sizeof(smp->msg_type));
    if(u.u_error)
        return;
    if(sz) {
        u.u_base = (caddr_t)uap->msgp + sizeof(smp->msg_type);
        u.u_segflg = 0;
        MOVE((caddr_t)(msg + msginfo.msgsz * smp->msg_spot),
            sz, B_READ);
        if(u.u_error)
            return;
    }
    u.u_rval1 = sz;
    qp->msg_cbytes -= smp->msg_ts;
    qp->msg_lrpid = u.u_procp->p_pid;
    qp->msg_rtime = time;
    curpri = PMSG;
    msgfree(qp, spmp, smp);
    return;
}
if(uap->msgflg & IPC_NOWAIT) {
    u.u_error = ENOMSG;
    return;
}
qp->msg_perm.mode |= MSG_RWAIT;
if(sleep((caddr_t)&qp->msg_qnum, PMSG | PCATCH)) {
    u.u_error = EINTR;
    return;
}
if(msgconv(uap->msgid) == NULL) {
    u.u_error = EIDRM;
    return;
}
goto findmsg;
}
/*
** msgsnd - Msgsnd system call.
*/
msgsnd()
{
    register struct a {
        int      msgid;
        struct msgbuf *msgp;
        int      msgsz;
        int      msgflg;

```

```

    }
    *uap = (struct a *)u.u_ap;
    register struct msgid_ds *qp; /* ptr to associated q */
    register struct msg *mp; /* ptr to allocated msg hdr */
    register int cnt, /* byte count */
    spot; /* msg pool allocation spot */
    long type; /* msg type */

    if((qp = msgconv(uap->msgid)) == NULL)
        return;
    if(ipcaccess(&qp->msg_perm, MSG_W))
        return;
    if((cnt = uap->msgsz) < 0 || cnt > msginfo.msgmax) {
        u.u_error = EINVAL;
        return;
    }
    (void) copyin((caddr_t)uap->msgp, (caddr_t)&type, sizeof(type));
    if(u.u_error)
        return;
    if(type < 1) {
        u.u_error = EINVAL;
        return;
    }
}
getres:
/* Be sure that q has not been removed. */
if(msgconv(uap->msgid) == NULL) {
    u.u_error = EIDRM;
    return;
}
/* Allocate space on q, message header, & buffer space. */
if(cnt + qp->msg_cbytes > qp->msg_qbytes) {
    if(uap->msgflg & IPC_NOWAIT) {
        u.u_error = EAGAIN;
        return;
    }
    qp->msg_perm.mode |= MSG_WWAIT;
    if(sleep((caddr_t)qp, PMSG | PCATCH)) {
        u.u_error = EINTR;
        qp->msg_perm.mode &= ~MSG_WWAIT;
        wakeup((caddr_t)qp);
        return;
    }
    goto getres;
}
if(msgfp == NULL) {
    if(uap->msgflg & IPC_NOWAIT) {
        u.u_error = EAGAIN;
        return;
    }
    if(sleep((caddr_t)&msgfp, PMSG | PCATCH)) {
        u.u_error = EINTR;
        return;
    }
    goto getres;
}
if(cnt && (spot = malloc(msgmap, btoq(cnt))) == NULL) {
    if(uap->msgflg & IPC_NOWAIT) {
        u.u_error = EAGAIN;
        return;
    }
    mapwant(msgmap)++;
    if(sleep((caddr_t)msgmap, PMSG | PCATCH)) {
        u.u_error = EINTR;
        return;
    }
    goto getres;
}
/* Everything is available, copy in text and put msg on q. */
if(cnt) {
    u.u_base = (caddr_t)uap->msgp + sizeof(type);
    u.u_segflg = 0;
    MOVE((caddr_t)(msg + msginfo.msgsz * --spot), cnt, B_WRITE);
    if(u.u_error) {
        mfree(msgmap, btoq(cnt), spot + 1);
        return;
    }

```

```

    }
}
qp->msg_qnum++;
qp->msg_cbytes += cnt;
qp->msg_lspid = u.u_procp->p_pid;
qp->msg_stime = time;
mp = msgfp;
msgfp = mp->msg_next;
mp->msg_next = NULL;
mp->msg_type = type;
mp->msg_ts = cnt;
mp->msg_spot = cnt ? spot : -1;
if(qp->msg_last == NULL) {
    qp->msg_first = mp; qp->msg_last = mp;
} else {
    qp->msg_last->msg_next = mp;
    qp->msg_last = mp;
}
if(qp->msg_perm.mode & MSG_RWAIT) {
    qp->msg_perm.mode &= ~MSG_RWAIT;
    curpri = PMSG;
    wakeup((caddr_t)&qp->msg_qnum);
}
u.u_rvall = 0;
}

/*
** msgsys - System entry point for msgctl, msgget, msgrcv, and msgsnd
** system calls.
*/

msgsys()
{
    int          msgctl(),
                msgget(),
                msgrcv(),
                msgsnd();
    static int   (*calls[])() = { msgget, msgctl, msgrcv, msgsnd };
    register struct a {
        unsigned   id;      /* function code id */
        int        *ap;     /* arg pointer for recvmsg */
    }
    *uap = (struct a *)u.u_ap;

    if(uap->id > 3) {
        u.u_error = EINVAL;
        return;
    }
    u.u_ap = &u.u_arg[1];
    (*calls[uap->id])();
}

```

```
/* @(#)name.c 1.1 */
#include "sys/utsname.h"

#ifdef lint
#define SYS      "sys"
#define NODE     "node"
#define REL      "rel"
#define VER      "ver"
#define MACH     "m68000"
#define TIMESTAMP "timestamp"
#endif
struct utsname utsname = {
    SYS,
    NODE,
    REL,
    VER,
    MACH,
};

char timestamp[] = TIMESTAMP;
```



```

/* #define HOWFAR */

/* @(#)nami.c 1.2 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysm.h"
#include "sys/sysinfo.h"
#include "sys/inode.h"
#include "sys/mount.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/buf.h"
#include "sys/var.h"

/*
 * Convert a pathname into a pointer to
 * an inode. Note that the inode is locked.
 */
/*
 * func = function called to get next char of name
 * &uchar if name is in user space
 * &schar if name is in system space
 * flag = 0 if name is sought
 *       1 if name is to be created
 *       2 if name is to be deleted
 */
struct inode *
namei(func, flag)
int (*func)();
{
    register struct user *up;
    register struct inode *dp;
    register c;
    register char *cp;
    register struct buf *bp;
    register i;
    dev_t d;
    off_t eo;

    /*
     * If name starts with '/' start from
     * root; otherwise start from current dir.
     */
    up = &u;
    sysinfo.namei++;
    c = (*func)();
    if (c == '\0') {
        up->u_error = ENOENT;
        return(NULL);
    }
    if (c == '/') {
        if ((dp = up->u_rdir) == NULL)
            dp = rootdir;
        while(c == '/')
            c = (*func)();
        if (c == '\0' && flag != 0) {
            up->u_error = ENOENT;
            return(NULL);
        }
    } else
        dp = up->u_cdir;
    (void) iget(dp->i_dev, dp->i_number);

loop:
    /*
     * Here dp contains pointer
     * to last component matched.
     */
    if (up->u_error)
        goto out;
    if (c == '\0')
        return(dp);

    /*
     * If there is another component,
     * gather up name into users' dir buffer.
     */
    cp = &up->u_dent.d_name[0];
    while(c != '/' && c != '\0' && up->u_error == 0) {
        if (cp < &up->u_dent.d_name[DIRSIZ])
            *cp++ = c;
        c = (*func)();
    }
    while (cp < &up->u_dent.d_name[DIRSIZ])
        *cp++ = '\0';

#ifdef HOWFAR
    printf("nami:about to scan for '%s'\n", up->u_dent.d_name);
#endif
    while(c == '/')
        c = (*func)();

seloop:
    /*
     * dp must be a directory and
     * must have X permission.
     */
#ifdef HOWFAR
    /* printf("nami:directory mode = 0%o\n", dp->i_mode&0x0000); */
#endif
    if ((dp->i_mode&IFMT) != IFDIR || dp->i_nlink==0)
        up->u_error = ENOTDIR;
    else
        (void) access(dp, IEXEC);
    if (up->u_error)
        goto out;

    /*
     * set up to search a directory
     */
    up->u_offset = 0;
    up->u_count = dp->i_size;
    up->u_pbsize = 0;
    eo = 0;
    bp = NULL;
    if (dp == up->u_rdir)
        if (up->u_dent.d_name[0] == '.')
            if (up->u_dent.d_name[1] == '.')
                if (up->u_dent.d_name[2] == '\0')
                    goto loop;

eloop:
    /*
     * If at the end of the directory,
     * the search failed. Report what
     * is appropriate as per flag.
     */
    if (up->u_count == 0) {
        if (bp != NULL)
            brelse(bp);
        if (flag == 1 && c == '\0') {
            if (access(dp, IWRITE))
                goto out;
            up->u_pdir = dp;
            if (eo)
                up->u_offset = eo - sizeof(struct direct);
            up->u_count = sizeof(struct direct);
            (void) bmap(dp, B_WRITE);
            if (up->u_error)
                goto out;
            return(NULL);
        }
        up->u_error = ENOENT;
        goto out;
    }
}

```

```

/*
 * If done with current block,
 * read the next directory block.
 * Release previous if it exists.
 */
if (up->u_pbsize == 0) {
    daddr_t bn;

    if (bp != NULL)
        brelse(bp);
    sysinfo.dirblk++;
    bn = bmap(dp, B_READ);
    if (up->u_error)
        goto out;
    if (bn < 0) {
        up->u_error = EIO;
        goto out;
    }
    bp = bread(dp->i_dev, bn);
    if (up->u_error) {
        brelse(bp);
        goto out;
    }
}

/*
 * Note first empty directory slot
 * in eo for possible creat.
 * String compare the directory entry
 * and the current component.
 * If they do not match, go back to eloop.
 */
cp = bp->b_un.b_addr + up->u_pboff;
up->u_offset += sizeof(struct direct);
up->u_pboff += sizeof(struct direct);
up->u_pbsize -= sizeof(struct direct);
up->u_count -= sizeof(struct direct);
up->u_dent.d_ino = ((struct direct *)cp)->d_ino;
if (up->u_dent.d_ino == 0) {
    if (eo == 0)
        eo = up->u_offset;
    goto eloop;
}
cp = &((struct direct *)cp)->d_name[0];
for (i=0; i<DIRSIZ; i++)
    if (*cp++ != up->u_dent.d_name[i])
        goto eloop;

/*
 * Here a component matched in a directory.
 * If there is more pathname, go back to
 * cloop, otherwise return.
 */
if (bp != NULL)
    brelse(bp);
if (flag==2 && c=='\0') {
    if (access(dp, IWRITE))
        goto out;
    return(dp);
}
d = dp->i_dev;
if (up->u_dent.d_ino == ROOTINO)
if (dp->i_number == ROOTINO)
if (up->u_dent.d_name[1] == '.')
    for (i=1; i<v_mount; i++)
        if (mount[(short)i].m_flags == MINUSE)
            if (mount[(short)i].m_dev == d) {
                iput(dp);
                dp = mount[(short)i].m_inodp;
                dp->i_count++;
                plock(dp);
                goto seloop;
            }
}

```

```

        iput(dp);
        dp = iget(d, up->u_dent.d_ino);
        if (dp == NULL)
            return(NULL);
        goto cloop;
}

out:
    iput(dp);
    return(NULL);
}

/*
 * Return the next character from the
 * kernel string pointed at by dirp.
 */
schar()
{
    return(*u.u_dirp++ & 0377);
}

/*
 * Return the next character from the
 * user string pointed at by dirp.
 */
uchar()
{
    register c;

    c = fubyte((caddr_t)u.u_dirp++);
    if (c == -1)
        u.u_error = EFAULT;
    return(c);
}

```

```

/* ipc.c 4.20 82/06/20 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/mmu.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/signal.h"
#include "sys/errno.h"
#include "sys/dir.h"
#include "sys/user.h"
#include "sys/proc.h"
#include "sys/file.h"
#include "sys/inode.h"
#include "sys/buf.h"
#include "net/misc.h"
/*
#include "net/mbuf.h"
*/
#include "net/protosw.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "net/in.h"
#include "net/in_sysm.h"

/*
 * Socket system call interface.
 *
 * These routines interface the socket routines to UNIX,
 * isolating the system interface from the socket-protocol interface.
 *
 * TODO:
 * SO_INTNOTIFY
 */

/*
 * Socket system call interface. Copy sa arguments
 * set up file descriptor and call internal socket
 * creation routine.
 */
ssocket()
{
    register struct ua {
        int type;
        struct sockproto *asp;
        struct sockaddr *asa;
        int options;
    } *uap = (struct ua *)u.u_ap;
    struct sockproto sp;
    struct sockaddr sa;
    struct socket *so;
    register struct file *fp;

    if ((fp = falloc((struct inode *)0, FSOCKET|FREAD|FWRITE)) == NULL)
        return;
    if (uap->asp && copyin((caddr_t)uap->asp, (caddr_t)&sp, sizeof(sp)) ||
        uap->asa && copyin((caddr_t)uap->asa, (caddr_t)&sa, sizeof(sa))) {
        u.u_error = EFAULT;
        fp->f_count = 0;
        fp->f_next = ffreelist;
        ffreelist = fp;
        return;
    }
    u.u_error = screate(&so, uap->type,
        uap->asp ? &sp : 0, uap->asa ? &sa : 0, uap->options);
    if (u.u_error)
        goto bad;
    fp->f_socket = (off_t)so;
    return;
bad:
    u.u_ofile[u.u_rvall] = 0;
    fp->f_count = 0;
    fp->f_next = ffreelist;
    ffreelist = fp;
}

}

/*
 * Accept system call interface.
 */
saccept()
{
    register struct a {
        int fdes;
        struct sockaddr *asa;
    } *uap = (struct a *)u.u_ap;
    struct sockaddr sa;
    register struct file *fp;
    struct socket *so;
    int s;

    if (uap->asa && useracc((caddr_t)uap->asa, sizeof(sa), B_WRITE)==0) {
        u.u_error = EFAULT;
        return;
    }
    fp = getf(uap->fdes);
    if (fp == 0)
        return;
    if ((fp->f_flag & FSOCKET) == 0) {
        u.u_error = ENOTSOCK;
        return;
    }
    s = splnet();
    so = (struct socket *)fp->f_socket;
    if ((so->so_state & SS_NBIO) &&
        (so->so_state & SS_CONNAWAITING) == 0) {
        u.u_error = EWOULDBLOCK;
        splx(s);
        return;
    }
    while ((so->so_state & SS_CONNAWAITING) == 0 && so->so_error == 0) {
        if (so->so_state & SS_CANTRCVMORE) {
            so->so_error = ECONNABORTED;
            break;
        }
        (void) sleep((caddr_t)&so->so_timeo, PZERO+1);
    }
    if (so->so_error) {
        u.u_error = so->so_error;
        splx(s);
        return;
    }
    u.u_error = soaccept(so, &sa);
    if (u.u_error) {
        splx(s);
        return;
    }
    if (uap->asa)
        (void) copyout((caddr_t)&sa, (caddr_t)uap->asa, sizeof(sa));
    /* deal with new file descriptor case */
    /* u.u_r_rvall = ... */
    splx(s);
}

/*
 * Connect socket to foreign peer; system call
 * interface. Copy sa arguments and call internal routine.
 */
sconnect()
{
    register struct ua {
        int fdes;
        struct sockaddr *a;
    } *uap = (struct ua *)u.u_ap;
    struct sockaddr sa;
    register struct file *fp;
    register struct socket *so;
    int s;

    if (copyin((caddr_t)uap->a, (caddr_t)&sa, sizeof(sa))) {
        u.u_error = EFAULT;
    }
}

```

```

        return;
    }
    fp = getf(uap->fdes);
    if (fp == 0)
        return;
    if ((fp->f_flag & FSOCKET) == 0) {
        u.u_error = ENOTSOCK;
        return;
    }
    so = (struct socket *)fp->f_socket;
    u.u_error = soconnect(so, &sa);
    if (u.u_error)
        return;
    s = splnet();
    if ((so->so_state & SS_NBIO) &&
        (so->so_state & SS_ISCONNECTING)) {
        u.u_error = EINPROGRESS;
        splx(s);
        return;
    }
    while ((so->so_state & SS_ISCONNECTING) && so->so_error == 0)
        (void) sleep((caddr_t)&so->so_timeo, PZERO+1);
    u.u_error = so->so_error;
    so->so_error = 0;
    splx(s);
}

/*
 * Send data on socket.
 */
ssend()
{
    register struct a {
        int     fdes;
        struct  sockaddr *asa;
        caddr_t cbuf;
        unsigned count;
    } *uap = (struct a *)u.u_ap;
    register struct file *fp;
    struct sockaddr sa;

    fp = getf(uap->fdes);
    if (fp == 0)
        return;
    if ((fp->f_flag & FSOCKET) == 0) {
        u.u_error = ENOTSOCK;
        return;
    }
    u.u_base = uap->cbuf;
    u.u_count = uap->count;
    u.u_segflg = 0;
    if (useracc(uap->cbuf, uap->count, B_READ) == 0 ||
        uap->asa && copyin((caddr_t)uap->asa, (caddr_t)&sa, sizeof (sa))) {
        u.u_error = EFAULT;
        return;
    }
    u.u_error = send((struct socket *)fp->f_socket, uap->asa ? &sa : 0);
    u.u_rvall = uap->count - u.u_count;
}

/*
 * Receive data on socket.
 */
sreceive()
{
    register struct a {
        int     fdes;
        struct  sockaddr *asa;
        caddr_t cbuf;
        u_int   count;
    } *uap = (struct a *)u.u_ap;
    register struct file *fp;
    struct sockaddr sa;

    fp = getf(uap->fdes);
    if (fp == 0)

```

```

        return;
    if ((fp->f_flag & FSOCKET) == 0) {
        u.u_error = ENOTSOCK;
        return;
    }
    u.u_base = uap->cbuf;
    u.u_count = uap->count;
    u.u_segflg = 0;
    if (useracc(uap->cbuf, uap->count, B_WRITE) == 0 ||
        uap->asa && copyin((caddr_t)uap->asa, (caddr_t)&sa, sizeof (sa))) {
        u.u_error = EFAULT;
        return;
    }
    u.u_error = soreceive((struct socket *)fp->f_socket, uap->asa ? &sa : 0);
    if (u.u_error)
        return;
    if (uap->asa)
        (void) copyout((caddr_t)&sa, (caddr_t)uap->asa, sizeof (sa));
    u.u_rvall = uap->count - u.u_count;
}

/*
 * Get socket address.
 */
ssocketaddr()
{
    register struct a {
        int     fdes;
        struct  sockaddr *asa;
    } *uap = (struct a *)u.u_ap;
    register struct file *fp;
    register struct socket *so;
    struct sockaddr addr;

    fp = getf(uap->fdes);
    if (fp == 0)
        return;
    if ((fp->f_flag & FSOCKET) == 0) {
        u.u_error = ENOTSOCK;
        return;
    }
    so = (struct socket *)fp->f_socket;
    u.u_error =
        (*so->so_proto->pr_usrreq)(so, PRU_SOCKADDR, 0, (caddr_t)&addr);
    if (u.u_error)
        return;
    if (copyout((caddr_t)&addr, (caddr_t)uap->asa, sizeof (addr)))
        u.u_error = EFAULT;
}

```



```

/* @(#)pipe.c 1.4 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/systm.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/buf.h"
#include "sys/filsys.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/inode.h"
#include "sys/file.h"
#include "sys/mount.h"
#include "sys/var.h"

/*
 * 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;
    register struct user *up;

    up = &u;
    ip = ialloc(getpdev(), IFIFO, 0);
    if (ip == NULL)
        return;
    rf = falloc(ip, FREAD);
    if (rf == NULL) {
        iput(ip);
        return;
    }
    r = up->u_rvall;
    wf = falloc(ip, FWRITE);
    if (wf == NULL) {
        rf->f_count = 0;
        rf->f_next = ffreelist;
        ffreelist = rf;
        up->u_ofile[r] = NULL;
        iput(ip);
        return;
    }
    up->u_rval2 = up->u_rvall;
    up->u_rvall = r;
    ip->i_count = 2;
    ip->i_frnt = 1;
    ip->i_fwnt = 1;
    prele(ip);
}

/*
 * Open a pipe
 * Check read and write counts, delay as necessary
 */
openp(ip, mode)
register struct inode *ip;
register mode;
{
    if (mode&FREAD) {
        if (ip->i_frnt++ == 0)
            wakeup((caddr_t)&ip->i_frnt);
    }
    if (mode&FWRITE) {
        if (mode&FNDELAY && ip->i_frnt == 0) {
            u.u_error = ENXIO;
            return;
        }
        if (ip->i_fwnt++ == 0)
            wakeup((caddr_t)&ip->i_fwnt);
    }
}

```

```

    if (mode&FREAD) {
        while (ip->i_fwnt == 0) {
            if (mode&FNDELAY || ip->i_size)
                return;
            (void) sleep((caddr_t)&ip->i_fwnt, PPIPE);
        }
    }
    if (mode&FWRITE) {
        while (ip->i_frnt == 0)
            (void) sleep((caddr_t)&ip->i_frnt, PPIPE);
    }
}

/*
 * Close a pipe
 * Update counts and cleanup
 */
closep(ip, mode)
register struct inode *ip;
register mode;
{
    register i;
    daddr_t bn;

    if (mode&FREAD) {
        if ((--ip->i_frnt == 0) && (ip->i_fflag&IFIW)) {
            ip->i_fflag &= ~IFIW;
            wakeup((caddr_t)&ip->i_fwnt);
        }
    }
    if (mode&FWRITE) {
        if ((--ip->i_fwnt == 0) && (ip->i_fflag&IFIR)) {
            ip->i_fflag &= ~IFIR;
            wakeup((caddr_t)&ip->i_frnt);
        }
    }
    if ((ip->i_frnt == 0) && (ip->i_fwnt == 0)) {
        for (i = NFADDR-1; i > 0; i--) {
            bn = ip->i_faddr[i];
            if (bn == (daddr_t)0)
                continue;
            ip->i_faddr[i] = (daddr_t)0;
            free(ip->i_dev, bn);
        }
        ip->i_size = 0;
        ip->i_frptr = 0;
        ip->i_fwptr = 0;
        ip->i_flag |= IUPD|ICRG;
    }
}

/*
 * 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;
        (void) sleep((caddr_t)ip, PINOD);
    }
    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);
    }
}

/*
 * Return the mounted pipe device
 */
dev_t
getpdev()
{
    register struct mount *mp;
    register dev_t dev;

    dev = pipedev;
    for (mp = &mount[0]; mp < (struct mount *)v.va_mount; mp++)
        if (mp->m_flags != MFREE && dev == mp->m_dev &&
            mp->m_bufp->b_un.b_filsys->s_ronly==0)
            return(dev);

    return(rootdev);
}
```

```

/* @(#)prf.c 1.2 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/buf.h"
#include "sys/conf.h"

/*
 * In case console is off,
 * panicstr contains argument to last call to panic.
 */
char *panicstr;

/*
 * Scaled down version of C Library printf.
 * Only %c %s %u %d (==%u) %o %x %D are recognized.
 * Used to print diagnostic information
 * directly on console tty.
 * Since it is not interrupt driven,
 * all system activities are pretty much suspended.
 * Printf should not be used for chit-chat.
 */
/* VARARGS1 */
printf(fmt, x1)
register char *fmt;
unsigned x1;
{
    register c;
    register unsigned int *adx;
    register int b, i, any;
    char *s;

#ifdef PRINTFSTALL
    for (c = 0; c < PRINTFSTALL; c++)
        ;
#endif
    adx = &x1;
loop:
    while((c = *fmt++) != '%') {
        if(c == '\0')
            return;
        (*putchar)(c);
    }
    c = *fmt++;
    if(c == 'd' || c == 'u' || c == 'o' || c == 'x')
        printn((long)*adx, c=='o'? 8: (c=='x'? 16:10));
    else if(c == 'c')
        (*putchar)(*adx);
    else if(c == 'b') {
        b = *adx++;
        s = (char *) *adx;
        printn((long) b, *s++);
        any = 0;
        if (b) {
            (*putchar)('<');
            while (i = *s++ {
                if (b & (1 << (i - 1))) {
                    if (any)
                        (*putchar)(',');
                    any = 1;
                    for (; (c = *s) > 32; s++)
                        (*putchar)(c);
                }
                else
                    for (; *s > 32; s++)
                        ;
            }
            if (any)
                (*putchar)('>');
        }
    }
    else if(c == 's') {
        s = (char *) *adx;
        while(c = *s++)
            (*putchar)(c);
    }
}

```

```

    } else if (c == 'D') {
        printn(*(long *)adx, 10);
        adx += (sizeof(long) / sizeof(int)) - 1;
    }
    adx++;
    goto loop;
}

printn(n, b)
long n;
register b;
{
    register i, nd, c;
    int flag;
    int plmax;
    char d[12];

    c = 1;
    flag = n < 0;
    if (flag)
        n = -n;
    if (b==8)
        plmax = 11;
    else if (b==10)
        plmax = 10;
    else if (b==16)
        plmax = 8;
    if (flag && b==10) {
        flag = 0;
        (*putchar)('-');
    }
    for (i=0; i<plmax; i++) {
        nd = n%b;
        if (flag) {
            nd = (b - 1) - nd + c;
            if (nd >= b) {
                nd -= b;
                c = 1;
            } else
                c = 0;
        }
        d[i] = nd;
        n = n/b;
        if ((n==0) && (flag==0))
            break;
    }
    if (i==plmax)
        i--;
    for (; i>=0; i--) {
        (*putchar)("0123456789ABCDEF"[d[i]]);
    }
}

/*
 * Panic is called on unresolvable fatal errors.
 * It syncs, prints "panic: msg" and then loops.
 */
panic(s)
char *s;
{
    if (s && panicstr)
        printf("Double panic: %s\n", s);
    else {
        if (s)
            panicstr = s;
        update();
        printf("panic: %s\n", panicstr?panicstr:"???");
    }
    for(;;)
        idle();
}

/*
 * prdev prints a warning message.
 * dev is a block special device argument.
 */

```



```
prdev(str, dev)
char *str;
dev_t dev;
{
    register maj;

    maj = bmajor(dev);
    if (maj >= bdevcnt) {
        printf("%s on bad dev %o(8)\n", str, dev);
        return;
    }
    (*bdevsw[maj].d_print)(minor(dev), str);
}

/*
 * prcom prints a diagnostic from a device driver.
 * prt is device dependent print routine.
 */
prcom(prt, bp, er1, er2)
int (*prt)();
register struct buf *bp;
{
    (*prt)(bp->b_dev, "\nDevice error");
    printf("bn = %D er = %o,%o\n", bp->b_blkno, er1, er2);
}
```

```

/* #define HOWFAR */
/* #define UNISOFT          /* allow access to boot partition */
/*
 * Priam Datatower
 *
 * (C) 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with UniSoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by UniSoft.
 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/utsname.h"
#include "sys/buf.h"
#include "sys/eelog.h"
#include "sys/erec.h"
#include "sys/iobuf.h"
#include "sys/system.h"
#include "sys/var.h"
#include "sys/uioc1.h"
#include "sys/al_ioc1.h"
#include "sys/diskformat.h"
#include "setjmp.h"
#include "sys/reg.h"
#include "sys/altblk.h"
#include "sys/cops.h"
#include "sys/pport.h"
#include "sys/priam.h"
#include "sys/swapsz.h"

#define logical(x)      (minor(x) & 7)      /* eight logicals per phys */
#define physical(x)    ((minor(x) & 0xF0) >> 4) /* 10 physical devs */
#define splpm      spl5

/*
 * When the disk is first opened its size is determined and pm_sizes is
 * initialized accordingly (in pmdinit).
 *
 * The first 100 blocks are reserved for the boot program and
 * are inaccessible via unix.
 */
#define MAXBOOT 100
struct pm_sizes {
    daddr_t sz_offset;
    daddr_t sz_size;
} pm_sizes[NPM][8];

#define GETBUF(bp)      splpm(); \
    while (bp->b_flags & B_BUSY) { \
        bp->b_flags |= B_WANTED; \
        (void)sleep((caddr_t)bp, PRIBIO+1); \
    } \
    bp->b_flags |= B_BUSY; \
    spl0()

#define FREEBUF(bp)      splpm(); \
    if (bp->b_flags & B_WANTED) \
        wakeup((caddr_t)bp); \
    bp->b_flags = 0; \
    spl0()

struct iostat pmstat[NPM];
struct iobuf  pmstab = tabinit(PM3,pmstat); /* active buffer header */
struct buf    pmcbuf; /* command buffer */
struct buf    pmrbuf;
char pmiflag[NPM];

```

```

char pmresults[NPMRES]; /* result regs. for last command comp ack */
int pmblks[NPM]; /* number of blocks on disk */
int pmcblkcnt; /* current block count */
struct pmfmtparms pmfmt; /* format parameters for packet-based format cmd */
struct pmfmtstat pmfstat; /* format status from packet-based format cmd */
/* variables needed to share info for interrupt routines */
char pmstate; /* idling or waiting for interrupt */
#define IDLING 0
#define INITING 1
#define READING1 2 /* 1st intr while reading */
#define READING2 3 /* 2nd intr while reading */
#define WRITING1 4 /* 1st intr while writing */
#define WRITING2 5 /* 2nd intr while writing */
#define FMTING1 6 /* 1st intr while formatting */
#define FMTING2 7 /* 2nd intr while formatting */
#define FMTING3 8 /* 3rd intr while formatting */
#define FMTING4 9 /* 4th intr while formatting */
caddr_t pmma; /* current memory address */
char pmerror; /* error value from intr */

```

```

pmpopen(dev)
register dev;
{
    register punit;
    extern char slot[];

    punit = physical(dev);
    if (punit >= NPM) {
        u.u_error = ENXIO;
        return(-1);
    }
    if (slot[punit] != PM3) {
        printf("Unix pmpopen: no Priam card in slot %d\n", punit);
        u.u_error = ENODEV;
        return(-1);
    }
    if (pmiflag[punit] == 0) {
        if (pmdinit(dev)) {
            u.u_error = EIO;
            return(-1);
        }
        pmiflag[punit]++;
    }
    return(0);
}

```

```

/*
 * pmdinit - initialize device (sequence up)
 */
pmdinit(dev)
register dev_t dev;
{
    register struct buf *bp = &pmcbuf;
    register punit, i;
    register struct pm_sizes *sp;
    register daddr_t offset, size;

    punit = physical(dev);
    GETBUF(bp);
    bp->b_dev = dev;
    bp->b_resid = PMRDEVPMS;
    pmblks[punit] = 0;
    pmstrategy(bp);
    iowait(bp);
    i = bp->b_flags & B_ERROR;
    FREEBUF(bp);
    if (i)
        return(-1);
    sp = pm_sizes[punit];
    offset = MAXBOOT + 1; /* avoid boot area */
    size = pmblks[punit] - offset;
    sp[7].sz_offset = offset; /* h = entire */
    sp[7].sz_size = size;
    sp[1].sz_offset = offset; /* b = swap */
    sp[1].sz_size = PMNSWAP;
    offset += PMNSWAP;
}

```

```

size -- PMNSWAP;
if (size < 0) {
    printf("Unix pmdinit: disk size = %d (insufficient)\n", pmmblks[punit]);
    return(-1);
}
sp[0].sz_offset = offset;          /* a = root */
sp[0].sz_size = size;
for (i = 2; i < 7; i++) {
    sp[i].sz_offset = (daddr_t)0; /* c-g =spare */
    sp[i].sz_size = (daddr_t)0;
}
return(0);
}

pmstrategy(bp)
register struct buf *bp;
{
    register punit, lunit, bn;

    punit = physical(bp->b_dev);
    if (bp == &pmcbuf) {          /* if command */
        pmstat[punit].io_misc++; /* errlog: */
        splpm();
        if (pmtab.b_actf == (struct buf *)NULL) /* set up links */
            pmtab.b_actf = bp;
        else
            pmtab.b_actl->av_forw = bp;
        pmtab.b_actl = bp;
        bp->av_forw = (struct buf *)NULL;
    } else {
        lunit = logical(bp->b_dev);
        bn = bp->b_blkno + pm_sizes[punit][lunit].sz_offset;
#ifdef UNISOFT
        if (bp->b_blkno < 0) {
#else UNISOFT
        if (bp->b_blkno < 0 || bn <= MAXBOOT) {
#endif UNISOFT
            prdev("pmstrategy: illegal blkno", bp->b_dev);
            printf("blkno=%d bcount=%d\n", bp->b_blkno, bp->b_bcount);
            bp->b_flags |= B_ERROR;
            iodone(bp);
            return;
        }
        pmstat[punit].io_ops++; /* errlog: */
        bp->b_resid = bn;        /* resid for disksort */
        splpm();
        disksort(&pmtab, bp);
    }
    if (pmtab.b_active == 0)
        pmstart();
    SPL0();
}

pmstart()
{
    register struct buf *bp;
    register offset, bn, lunit, punit;

loop:
    if ((bp = pmtab.b_actf) == (struct buf *)NULL)
        return;
    if (pmtab.b_active == 0) {
        pmtab.b_active = 1;
        if (bp != &pmcbuf)
            bp->b_resid = bp->b_bcount;
    }
    blkacty |= (1<<PM3);
    if (bp == &pmcbuf) {
        if (pmcmd(bp) != 0) { /* b_resid holds the command */
            bp->b_flags |= B_ERROR;
            goto next;
        }
        return;
    }
}

```

```

lunit = logical(bp->b_dev);
punit = physical(bp->b_dev);
offset = bp->b_bcount - bp->b_resid;
bn = bp->b_blkno + btod(offset); /* logical block number */
if (bp->b_resid < BSIZE || bn >= pm_sizes[punit][lunit].sz_size) {
    next:
#ifdef HOWFAR
    if (bp->b_resid != 0)
        printf("Unix pmstart: blkno=%d resid=%d bn=%d\n",
            bp->b_blkno, bp->b_resid, bn);
#endif HOWFAR

    pmtab.b_active = 0;
    pmtab.b_errcnt = 0;
    blkacty &= ~(1<<PM3);
    pmtab.b_actf = bp->av_forw;
    iodone(bp);
    goto loop;
}
bn += pm_sizes[punit][lunit].sz_offset; /* physical block number */
if (pmrw(punit, bn, bp->b_flags&B_READ, bp->b_un.b_addr+offset) != 0) {
    bp->b_flags |= B_ERROR;
    goto next;
}

/* ARGSUSED */
pmioctl(dev, cmd, addr, flag)
dev_t dev;
caddr_t addr;
{
    register punit;
    register struct buf *bp;

    punit = physical(dev);
    if (punit >= NPM) {
        u.u_error = EINVAL;
        return;
    }
    switch (cmd) {
        case UIOSIZE: /* get size of Priam disk */
            if (copyout((caddr_t)&pmmblks[punit], (caddr_t)addr, 4))
                u.u_error = EFAULT;
            break;
        case UIOCFORMAT:
            if (!user()) {
                u.u_error = EPERM;
                return;
            }
            bp = &pmcbuf;
            GETBUF(bp);
            bp->b_dev = dev;
            bp->b_resid = PMPKTXFER; /* stash the command in resid */
            pmstrategy(bp);
            iowait(bp);
            if (bp->b_flags & B_ERROR)
                u.u_error = EIO;
            FREEBUF(bp);
            break;
        default:
            u.u_error = ENOTTY;
            break;
    }
}

pmcmd(bp)
register struct buf *bp;
{
    register struct pm_base *pmhbase;
    register struct pm_base *addr;
    register punit;
    register struct pmfmtparms *fmtp = &pmfmt;

    punit = physical(bp->b_dev);
    pmerror = 0;
    switch (bp->b_resid) {
        case PMRDEVPMS: /* read device parameters (spin up if necessary) */

```



```

    }
    (void)pmackdt(addr); /* Apple's code doesn't check */
    goto wait;
case FMTING4: /* intr after read packet status */
ackcc: (void)pmackcc(addr);
        if ((pmfstat.pm_pstate != PMPRTCOMP)
            || pmfstat.pm_pristat) {
            pmerror = pmfstat.pm_pristat;
            bp->b_flags |= B_ERROR;
        }
    }
    break;
}
goto out;
}
if ((addr->status & CMD_DONE) == 0) { /* block transfer interrupt */
    if (!pmcblkcnt) { /* 2nd interrupt */
        if ((pmstate==READING2) || (pmstate==WRITING2))
            goto done;
        printf("Unix pmintr: state=%d, expecting 2nd\n", pmstate);
        bp->b_flags |= B_ERROR;
        goto out;
    }
    /* 1st interrupt */
    if ((pmstate!=READING1) && (pmstate!=WRITING1)) {
        printf("Unix pmintr: state=%d, expecting 1st\n", pmstate);
        bp->b_flags |= B_ERROR;
        goto out;
    }
    pmcblkcnt--;
    if (pmstate == READING1)
        pmrsect(punit); /* sets pmerror for parity */
    else
        pmwsect(punit);
    pmstate++; /* set to READING2 or WRITING2 */
    addr->cmdreg = PMCBTI; /* clear block transfer intr */
    addr = pmBWIaddr(pmhwbases); /* setup for next intr */
    status = addr->status;
    return;
} else { /* cmd completed (error) */
    if ((pmstate!=READING2) && (pmstate!=WRITING2))
        printf("Unix pmintr: command complete, state=%d\n", pmstate);
done: if (status = pmackcc(addr)) { /* ack intr */
        if ((pmstate==READING2) && (pmresults[0] == PMECCERR))
            /* ECC err requiring scavenger write */
            status = PMECCERR;
        else if (pmresults[0] == PMDTIMOUT)
            /* dev timeout - they reissue read/write */
            status = PMDTIMOUT;
        }
        if (pmerror != status)
            bp->b_flags |= B_ERROR;
    }
out: /*
    * because a single buffer can take several io operations,
    * we leave it to pmstart() to figure out when it's done
    */
    if (bp->b_flags & B_ERROR || bp == &pmcbuf) {
        if (bp->b_flags & B_ERROR) {
            register bn = 0;
            struct deverreg pmreg[2];

            printf("Unix: HARD I/O ERROR on /dev/pm%d%c ",
                punit, logical(bp->b_dev)+'a');
            if (pmerror) {
                if (pmerror & PMPERROR)
                    printf("parity error ");
                if (status = (pmerror & ~PMPERROR))
                    printf("error code 0x%x ", status);
            }
            if (bp != &pmcbuf) {
                bn = bp->b_blkno + btod(bp->b_bcount - bp->b_resid)
                    + pm_sizes[punit][logical(bp->b_dev)].sz_offset;

```

```

                printf("bn %d\n", bn);
            } else printf("\n");
            /* error logging */
            pmtab.io_stp = &pmstat[punit];
            pmreg[0].draddr = (long)0;
            pmreg[0].drvalue = pmerror;
            pmreg[0].drname = "pmerror";
            pmreg[0].drbits = "Priam disk status code";
            pmreg[1].draddr = (long)0;
            pmreg[1].drvalue = pmstate;
            pmreg[1].drname = "pmstate";
            pmreg[1].drbits = "Priam driver state";
            fntberr(&pmtab,
                (unsigned)punit,
                (unsigned)0, /* cylinder */
                (unsigned)0, /* track */
                (unsigned)bn, /* sector */
                (long)(sizeof(pmreg)/sizeof(pmreg[0])), /* regcnt */
                &pmreg[0], &pmreg[1]);
        }
        logberr(&pmtab, 1); /* log hard (unrecovered) error */
    }
    blkacty &= ~(1<<PM3);
    pmtab.b_active = 0;
    pmtab.b_errcnt = 0;
    pmtab.b_actf = bp->av_forw;
    lodone(bp);
} else
    bp->b_resid -- 512;
pmstate = IDLING;
pmstart();
}
/*
* read block to memory at pmma
* from PRIAMASM.TEXT - READ_SECT
*/
pmrsect(punit)
register punit;
{
    register struct pm_base *pmhwbases;
    register struct pm_base *hwaddr;
    register short *waddr;
    register i;

    pmhwbases = pmaddr(punit); /* base address for this card */
    hwaddr = pmBwaddr(pmhwbases); /* byte mode - waiting */
    while ((hwaddr->status & DTREQ) == 0) ;
    hwaddr = pmPaddr(pmhwbases); /* parity checking enabled */
    waddr = (short *)pmma;
    *waddr = hwaddr->data; /* start read of first word */
    i = 12;
    do {
        *waddr = hwaddr->data;
    } while (--i); /* read 24-byte header */
    waddr = (short *)pmma;
    i = 255;
    do {
        *waddr++ = hwaddr->data;
    } while (--i); /* read 510 bytes */
    hwaddr = pmNaddr(hwaddr); /* read last 2 avoiding device cycle */
    *waddr = hwaddr->data;
    hwaddr = pmPaddr(punit); /* parity in io select space */
    pmerror |= (hwaddr->parity & PMPERROR);
}
/*
* write block from memory at pmma
* from PRIAMASM.TEXT - WRITE_SECT
*/
pmwsect(punit)
register punit;
{
    register struct pm_base *pmhwbases;
    register struct pm_base *hwaddr;
    register short *waddr;

```

```

register i;
register short tmp = 0;

pmhbase = pmaddr(punit); /* base address for this card */
hwaddr = pmBwaddr(pmhbase); /* byte mode - waiting */
while ((hwaddr->status & DTREQ) == 0) ;
hwaddr = pmhbase;
i = 12;
do {
    hwaddr->data = tmp; /* write dummy header */
} while ( --i ); /* write 24 bytes */
waddr = (short *)pmma;
i = 256;
do {
    hwaddr->data = *waddr++;
} while ( --i ); /* write 512 bytes */
}

/*
 * wait for register file not busy
 * from PRIAMASM.TEXT and PRIAMCARDASM.TEXT - WAITRF
 */
pmwaitrf(addr)
register struct pm_base *addr;
{
    register i;

    i = TIMELIMIT;
    while ((addr->status & ISR_BUSY) != 0) {
        if (--i)
            continue;
        return(-1);
    }
    return(0);
}

/*
 * acknowledge command completion
 * from PRIAMASM.TEXT and PRIAMCARDASM.TEXT - ACKCC
 */
pmackcc(addr)
register struct pm_base *addr; /* pmBwaddr */
{
    register i;

    i = TIMELIMIT;
    while ((addr->status & CMD_DONE) == 0) { /* wait for cmd done */
        if (--i)
            continue;
        printf("pmackcc: timeout waiting for CMD_DONE\n");
        return(-1);
    }
    pmresults[0] = addr->r0; /* transaction status */
    pmresults[1] = addr->r1;
    pmresults[2] = addr->r2;
    pmresults[3] = addr->r3;
    pmresults[4] = addr->r4;
    pmresults[5] = addr->r5;
    addr->cmdreg = 0; /* send command acknowledge (clrb) */
    i = TIMELIMIT;
    while ((addr->status & CMD_DONE) != 0) { /* wait 'til reset */
        if (--i)
            continue;
        printf("pmackcc: timeout waiting for CMD_DONE reset\n");
        return(-1);
    }
    return(pmresults[0] & PMCTYPE); /* error completion type */
}

/*
 * acknowledge data transfer
 * from PRIAMASM.TEXT - ACKDT
 * Their code does not check the return value, so who knows what'll happen
 * if it times out?
 */
pmackdt(addr)

```

```

register struct pm_base *addr; /* pmBwaddr */
{
    register i;

    addr->cmdreg = PMCBTI; /* clear block transfer intr */
    i = TIMELIMIT;
    while ((addr->status & BTR_INT) != 0) {
        if (--i)
            continue;
        printf("Unix pmackdt: timeout\n");
        return(-1);
    }
    addr = pmIaddr(addr); /* pmBWIaddr */
    i = addr->status & BTR_INT; /* setup for next intr */
    return(0);
}

/*
 * pmcinit - initialize controller (turn on motor)
 * called from oem7init (config.c)
 */
pmcinit(unit)
register unit;
{
    register struct pm_base *addr;
    register i;

    addr = pmaddr(unit); /* hwbase */
    addr = pmBwaddr(addr); /* byte mode - waiting */
    if ((addr->status & CMD_DONE) != 0) { /* cmd complete must be false */
        if (pmwaitrf(addr)) {
            printf("timeout with status CMD_DONE\n");
            goto err;
        }
        addr->cmdreg = 0; /* command ack power up (clrb) */
        i = TIMELIMIT;
        while ((addr->status & CMD_DONE) == 0) {
            if (--i)
                continue;
            printf("timeout waiting for power up\n");
            goto err;
        }
    }
    if (pmwaitrf(addr)) {
        printf("timeout before software reset\n");
        goto err;
    }
    addr->cmdreg = PMRESET; /* issue software reset */
    if ((i=pmackcc(addr)) != PMICOMP) {
        printf("software reset error 0x%x\n", i);
        goto err;
    }
    if (pmwaitrf(addr)) {
        printf("timeout before read mode\n");
        goto err;
    }
    addr->p0 = 0; /* (clrb) */
    addr->cmdreg = PMRMODE;
    if (pmackcc(addr)) {
        printf("read mode cmd ack failed\n");
        goto err;
    }
    if (pmwaitrf(addr)) {
        printf("timeout after read mode\n");
        goto err;
    }
    addr->p0 = 0; /* (clrb) */
    if (pmresults[3] == 2) {
        printf("Smart E controller not implemented\n");
        goto err;
    }
    addr->p1 = PMLOGSECT; /* set mode "logical sector mode" */
    addr->p2 = 0; /* (clrb) */
    addr->cmdreg = PMSMODE;
    if (pmackcc(addr)) {
        printf("set mode cmd ack failed\n");

```

```
        goto err;
    }
    if (pmwaitrf(addr)) {
        printf("timeout after set mode\n");
        goto err;
    }
    addr->p0 = 0x40;
    addr->p1 = PMPSEL1;
    addr->p2 = PM1PARMS;
    addr->cmdreg = PMSETP;
    if (pmackcc(addr)) {
        printf("set parms 1 cmd ack failed\n");
        goto err;
    }
    if (pmwaitrf(addr)) {
        printf("timeout after set parms 1\n");
        goto err;
    }
    addr->p0 = 0x40;
    addr->p1 = PMPSEL0;
    addr->p2 = PM0PARMS;
    addr->cmdreg = PMSETP;
    if (pmackcc(addr)) {
        printf("set parms 0 cmd ack failed\n");
        goto err;
    }
}

return 0;
err: printf("Unix pmcinit: can't initialize controller\n");
return(-1);
}

pmread(dev)
dev_t dev;
{
    physio(pmstrategy, &pmrbuf, dev, B_READ);
}

pmwrite(dev)
dev_t dev;
{
    physio(pmstrategy, &pmrbuf, dev, B_WRITE);
}

pmprint(dev, str)
char *str;
{
    printf("%s on priam drive %d, slice %d\n", str, (dev>>4)&0xF, dev&7);
}
}
```

```

/*#define NOERROR */
#define MOREASM
/*
 * SY6522 disk driver
 */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/utsname.h"
#include "sys/buf.h"
#include "sys/elog.h"
#include "sys/erec.h"
#include "sys/iobuf.h"
#include "sys/system.h"
#include "sys/var.h"
#include "sys/ttold.h"
#include "setjmp.h"
#include "sys/profile.h"
#include "sys/pport.h"
#include "sys/d_profile.h"
#include "sys/cops.h"
#include "sys/swapsz.h"

#ifdef notdef
/* these are in d_profile.h */
#define logical(x) (minor(x) & 7) /* eight logicals per phys */
#define interleave(x) (minor(x) & 0x8) /* interleave bit for swapping */
#define physical(x) ((minor(x) & 0xF0) >> 4) /* 10 physical devs */
#endif

char pro_secmap[NSEC] = {
/*11*/ 0, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
};

/* Logical Units */
/* The first 100 blocks are reserved for the boot program and
are inaccessible via unix.
*/
#define MAXBOOT 100
struct prlmap {
daddr_t pm_beg; /* Base address in blocks */
daddr_t pm_len; /* Number of blocks in logical device */
} prlmap[] = {
/* a */ {PRNSWAP+101, 16955}, /* root filesystem on 10 Meg. disk */
/* b */ {101, PRNSWAP}, /* swap area (2400 blocks) */
/* c */ {PRNSWAP+101, 7227}, /* root filesystem on 5 Meg. disk */
/* d */ {9728, 9728}, /* 2nd filesystem on 10 Meg. disk */
/* e */ {0, 0}, /* unused */
/* f */ {0, 7168}, /* old root filesystem (old a) */
/* g */ {7168, 2496}, /* old swap (old b) */
/* h */ {101, 19355}, /* f.s. using entire 10 Meg. disk */
};
/* THESE MAY REPLACE f AND g ABOVE */
/* f {4101, 15355}, /* alternate root f.s. on 10 Meg. disk */
/* g {101, 4000}, /* alternate swap */

struct iostat prostat[NPPDEVs];
struct iobuf protab = tabinit(PR0,prostat); /* active buffer header */
struct buf rprobuf; /* Raw input-output buffer */
struct proheader rphbuf;
struct proheader phbuf;

#ifdef NOERROR
#define ERROR(x) printf("HARD DISK ERROR "); printf x
char *pro_ufmt = "ASSERTION (%s) FAILED IN PROC %s ";
#define ASSERT(e, p, m, x) if (!(e)) {\
printf(pro_ufmt,"e","p");\
printf m;\
printf ("\\n");\
x;};
#else

```

```

#define ERROR(x)
#define ASSERT(e, p, m, x)
#endif

/* Contrast change ok flag. Maintained by the disk driver. When 0 the
 * parallel port is not in use and may be switched to allow console contrast
 * changes. If the contrast is waiting for the disk then 'l2_rcflag' is one.
 * When convenient and this is set the disk will call l2ramp.
 */
int ppinuse;
extern char l2_rcflag;

/*
 * proopen - check for existence of controller
 */
proopen(dev)
{
int i;
register struct device_d *devp;
int prointr();
extern char slot[];

i = physical(dev);
if (i) { /* for expansion slot check slot number and type */
if (!PPOK(i) || (slot[PPSLOT(i)] != PR0)) {
u.u_error = ENXIO;
return 1;
}
}
devp = pro_da[i];
u.u_error = 0;
if (iocheck(&devp->d_ifr)) {
{ asm("nop"); }
if (prodata[i].pd_da != devp) { /* not already setup */
if (setppint((prodata[i].pd_da = devp),prointr))
goto fail;
if (proinit(&prodata[i])) {
freepin(devp);
goto fail;
}
}
} else {
fail:
u.u_error = ENXIO;
prodata[i].pd_da = (struct device_d *)0;
return 1;
}
return 0;
}

/*
 * prostrategy - set up to start the transfer
 */
prostrategy(bp)
register struct buf *bp;
{
int pun = physical(bp->b_dev);
register struct prodata *p = &prodata[pun];
register struct buf *up;

if (!p->pd_da) {
printf("Attempt to read/write unopened profile device\\n");
printf("bp=%x dev=%x (Unit %d)\\n",bp,bp->b_dev,pun);
p->pd_err = "device not open";
goto haderr;
}
if (bp->b_blkno >= 0 &&
(bp->b_blkno < prlmap[logical(bp->b_dev)].pm_len)) {
bp->av_forw = (struct buf *)NULL; /* last of all bufs and */
bp->ul_forw = (struct buf *)NULL; /* last of units bufs */

SPL6(); /* must be highest of all ints for this code*/
if (protab.b_actf == NULL)
protab.b_actf = bp; /* empty - put on front */
else
protab.b_acti->av_forw = bp; /* else put at end */
}
}

```



```

    protab.b_actl = bp;

    if (p->pd_actv == 0) {          /* controller inactive */
        p->pd_actv = bp;          /* start of unit blk list */
    }

/* Since we might fail before ever getting an interrupt
 * we must be prepared to do the buffer cleanup here also.
 */
    while (prostart(p)) {        /* start up the transfer */
        bp->b_resid = bp->b_bcount;
        bp->b_flags |= B_ERROR;
        prorelse(p, bp);
    }
} else {                          /* link onto unit list */
    for (up=p->pd_actv; up->ul_forw; up=up->ul_forw)
        ;
    up->ul_forw = bp;
}

    SPL0();
    return;
}
p->pd_err = "invalid blkno";
haderr:
    bp->b_resid = bp->b_bcount;
    bp->b_flags |= B_ERROR;
    iodone(bp);
    return;
}

/*
 * Release finished buffer and unlink from list. Two lists are maintained.
 * The av_forw pointers are used to link all the buffers in use by the driver
 * onto the protab iobuf header. The av_back pointers (dubbed ul_forw) are
 * used to link together the buffers into unit lists (headed by the prodata
 * entry for that unit).
 */
prorelse (p, bp)
register struct prodata *p;
register struct buf *bp;
{
    register struct buf *up;

    if (protab.b_actf == bp) {     /* first buffer */
        if ((protab.b_actf = bp->av_forw) == (struct buf *)0)
            protab.b_actl = (struct buf *)0;
    } else {                       /* middle or last buffer */
        for (up=protab.b_actf; up->av_forw != bp; up=up->av_forw)
            if (!up->av_forw) panic("prorelse: buf list error");
        up->av_forw = bp->av_forw;
        if (!up->av_forw && (protab.b_actl == bp))
            protab.b_actl = up;
    }

    p->pd_actv = bp->ul_forw;      /* next buf for this unit */
    iodone(bp);
}

/*
 * proinit - initialize drive first time or after severe error
 */
proinit(p)
    struct prodata *p;
{
    register char zero = 0;
    register struct device_d *devp = p->pd_da;
    int pl;

    pl = spl6();
    devp->d_acr = zero;
    devp->d_pcr = 0x6B;          /* set controller CA2 pulse mode strobe */
    devp->d_ddra = zero;        /* set port A bits to input */
    if (devp == PFADDR)
        devp->d_ddrb &= 0x5C;   /* set BSY and OCD to input */
    else
        devp->d_ddrb &= 0xFC;   /* two or four port cards */

```

```

    devp->d_ddrb |= 0x1C;       /* set port B bits 2,3,4 to out */
    devp->d_irb &= ~DEN;        /* set enable = true */
    devp->d_irb |= CMD|DRW;     /* set command = false set direction = in */
    devp->d_t2cl = zero;
    devp->d_t2ch = zero;
    devp->d_ier = FIRQ|FCA1;
    zero = devp->d_irb;
    if (zero & OCD) {
        p->pd_state = SERR;
        splx(pl);
        return 1;
    }
    p->pd_state = SCMD;
    splx(pl);
    return 0;
}

/*
 * proread - process read from disk
 */
proread(dev)
dev_t dev;
{
    physio(prostrategy, &rprobuf, dev, B_READ);
}

/*
 * prowrite - process write to disk
 */
prowrite(dev)
dev_t dev;
{
    physio(prostrategy, &rprobuf, dev, B_WRITE);
}

/*
 * prostart - initiate the next logical io operation
 */
prostart(p)
register struct prodata *p;
{
    register struct buf *bp = p->pd_actv;

    if (!bp)
        return 0;
    ASSERT(bp&#amp;p, prostart, ("bp=x%x p=x%x", bp, p), while(1))

    p->pd_limit = prlmap[logical(bp->b_dev)].pm_beg; /* logical offset */
    p->pd_blkno = bp->b_blkno + p->pd_limit;          /* starting blk # */
#ifdef UNISOFT
    if (p->pd_blkno <= MAXBOOT)                    /* don't allow access to boot */
        return(-1);
#endif
    p->pd_limit += prlmap[logical(bp->b_dev)].pm_len; /* max blk # + 1 */
    p->pd_bcount = bp->b_bcount;
    p->pd_addr = bp->b_un.b_addr;
    return procmd(bp->b_flags, bp->b_dev, p->pd_blkno, (unsigned)p->pd_bcount);
}

/*
 * Procmd - initiate next physical io operation
 */
procmd(func, dev, bn, ct)
    register daddr_t bn;
    unsigned ct;
{
    register int pun = physical(dev);
    register struct prodata *p = &prodata[pun];
    register struct cmd *pc;

    ASSERT(ct!=0, Procmd, ("ct=%d", ct), while(1))

#ifdef UNISOFT
    if (bn <= MAXBOOT)                            /* check again to be sure */
        return(-1);
#endif
}

```

```

if (p->pd_state == SERR) {
    if (proinit(p))          /* initialize drive */
        return -1;
}

/* controller should not be busy now */

/* Build Command (ok to send extra bytes on write cmd) */
pc = &p->pd_cmdb;
pc->p_cmd = (func & B_READ) ? PROREAD : PROWRITE;
pc->p_high = bn >> 16;
pc->p_mid = bn >> 8;
if (interleave(dev) == 0)
    pc->p_low = bn;
else
    pc->p_low = (bn & 0xF0) | pro_secmap[bn&0xF];
pc->p_retry = 10;
pc->p_thold = 3;
p->pd_state = SCMD;
p->pd_nextst = SCMD;

if (prochk(p,SCMD)) {      /* sync controller to cmd state */
    if (!prochk(p,SCMD))  /* if it failed it should work now */
        return 0;
    p->pd_err = "cant force disk into CMD state";
    p->pd_state = SERR;
    return -1;
}
return 0;
}

printr(pun)
int pun;
{
    register struct device_d *devp; /* a5 */
    register char *cp;             /* a4 */
    register char csum;           /* d7 */          /*NOTUSED*/
    register char zero = 0;       /* d6 */
    register struct buf *bp;
    register struct prodata *p = &prodata[pun];
    register short i;
    struct proheader *ph;
    devp = p->pd_da;

    (void) spl6(); /* added April 4/84 to prevent panic in prorelse */
    /* changed from spl5 August 30/84 to fix multi-user bug on 2/10 */
    /* ASSERT((stats&BSY)==BSY,printr,("disk %d busy: state=%d, irb=x%x",pun,p->pd_state,stats),return(devp->d_ifr = devp->dcif);p->pd_addr;

#ifdef lint
    csum = 0;
    i = csum;
#endif

if ((bp = p->pd_actv) == 0) { /* spurious interrupt */
    /* printf("Spurious INT on profile dev %d [at %x]\n",pun,devp); */
    devp->d_ddrb |= 0x80; /* setup for a reset */
    devp->d_irb |= 0x80;
    devp->d_ddrb &= 0x7F;
    devp->d_ifr = devp->d_ifr; /* reset interrupt trap */
    /* proinit(p); */
    return;
}

/* ASSERT(bp!=0,printr,("dsk=%d ier=x%x ifr=x%x state=%d",pun,devp->d_ier&255,devp->d_ifr&255,p->pd_state),return devp->d_ifr=0x7F) */
ASSERT((p->pd_state!=SERR&&p->pd_state!=SSTOP),printr,("b_flags=0x%x",bp->b_flags),while(1))
ASSERT(physical(bp->b_dev)==pun,printr,("dev=x%x unit=%d",bp->b_dev,pun),while(1))

devp->d_ifr = devp->d_ifr; /* reset interrupt trap */
/*
 * Note that IO operations fail when OCD.
 * This may result in a 'panic'. Allowing it to
 * block and be restarted has it problems also.
 */
if (devp->d_irb & OCD) { /* cable disconnected ? */
    p->pd_err = "Open Cable Disconnect";
    goto haderr;
}

```

```

if (p->pd_state == SCMD) { /* Send command */
    devp->d_irb &= ~DRW; /* set dir-out */
    devp->d_ddra = 0xFF; /* set port A bits to output */
    /* Now send command */
    cp = (char *)(&p->pd_cmdb);
    if (*cp == PROREAD)
        p->pd_nextst = SRDBLK;
    else
        p->pd_nextst = SWRTD;

    devp->d_ira = *cp++; devp->d_ira = *cp++; devp->d_ira = *cp++;
    devp->d_ira = *cp++; devp->d_ira = *cp++; devp->d_ira = *cp;

    devp->d_irb |= DRW;
    devp->d_ddra = zero;
    if (prochk(p,p->pd_nextst)) {
        p->pd_err = "failed to issue cmd to disk";
        goto haderr;
    }
    return; /* will send data or get status next */
}

if (p->pd_state == SRDBLK || p->pd_state == SFINI) /* Read status word */
    devp->d_irb |= DRW;
devp->d_ddra = zero;
p->pd_sbuf = 0;
cp = (char *)(&p->pd_sbuf);
*cp++ = devp->d_ira;
*cp++ = devp->d_ira;
*cp++ = devp->d_ira;
*cp = devp->d_ira;
p->pd_sbuf &= ~STATMSK; /* mask off redund stat bits */
if (p->pd_sbuf) {
    ERROR(("dev %d: state=%d status=x%x\n",p->prodata,p->pd_state,p->pd_sbuf));
    p->pd_err = "bad status";
    goto haderr;
}

if (p->pd_state == SRDBLK) { /* Read successful so pickup data */
    ASSERT(p->pd_bcount>0,printr,(""),goto haderr)

    i = sizeof(rphbuf) - 1; /* sizeof header */
    cp = (char *)(&rphbuf);
    do *cp++ = devp->d_ira;
    while (--i != -1);

    i = min(SECSIZE, (unsigned)p->pd_bcount);

    if ((i & 3) == 0) {
        i = (i >> 2) - 1;
        do {
            asm (" movb a5(%9),a4(%9)");
            asm (" movb a5(%9),a4(%9)");
            asm (" movb a5(%9),a4(%9)");
            asm (" movb a5(%9),a4(%9)");
            /* asm (" movb a5(%9),d6 "); */
            /* asm (" movb d6,a4(%9)"); */
            /* asm (" eorb d6,d7 "); */
            /* asm (" movb a5(%9),d6 "); */
            /* asm (" movb d6,a4(%9)"); */
            /* asm (" eorb d6,d7 "); */
            /* asm (" movb a5(%9),d6 "); */
            /* asm (" movb d6,a4(%9)"); */
            /* asm (" eorb d6,d7 "); */
            /* asm (" movb a5(%9),d6 "); */
            /* asm (" movb d6,a4(%9)"); */
            /* asm (" eorb d6,d7 "); */
        } while (--i != -1); /* optimizes to DBRA */
    } else {
        i--;
        do {
            asm (" movb a5(%9),a4(%9)");
            /* asm (" movb a5(%9),d6 "); */
            /* asm (" movb d6,a4(%9)"); */
            /* asm (" eorb d6,d7 "); */
        }
    }
}

```

```

        ) while (--i != -1);
#ifdef MOREASM
    }
#endif
}

/* Determine if IO operation is completed or spans another block. */
p->pd_bcount -= min(SECSIZE, (unsigned)p->pd_bcount);
if (p->pd_bcount <= 0 || ++p->pd_blkno >= p->pd_limit) {
    bp->b_resid = p->pd_bcount;
    prorelse(p, bp);
    if (p->prodata == 0) ppinuse = 0;      /* contrast */

    if (prostart(p)) { /* startup next buf in chain */
        p->pd_err = "prostart failed on next blk";
        goto haderr;
    }
    return; /* next op started or no next op */
}
p->pd_addr += SECSIZE; /* setup for next block of io trans */
if (procmd(bp->b_flags, bp->b_dev, p->pd_blkno, (unsigned)p->pd_bcount)) {
    p->pd_err = "Procmd failed to continue operation";
    goto haderr;
}
return; /* end of i/o or beg of next blk */
}

if (p->pd_state == SWRTD) { /* send data */
    p->pd_nextst = SFINI;
    ASSERT(p->pd_bcount > 0, printr, (""), while(1))
    devp->d_irb &= ~DRW; /* set dir=out */
    devp->d_ddra = 0xFF; /* set port A bits to output */

    ph = &phbuf;
    ph->ph_fileid = p->pd_blkno ? 0xAAAA;
    i = sizeof(phbuf) - 1;
    cp = (char *) (ph);
    do devp->d_ira = *cp++;
    while (--i != -1);
    cp = (char *) p->pd_addr; /* place to get data from */

    i = min(SECSIZE, (unsigned)p->pd_bcount);
#ifdef MOREASM
    if ((i & 3) == 0) {
        i = (i >> 2) - 1;
        do {
            asm (" movb a4%, a5%(9) ");
            asm (" movb a4%, a5%(9) ");
            asm (" movb a4%, a5%(9) ");
            asm (" movb a4%, a5%(9) ");
            /* asm (" movb a4%, d6 "); */
            /* asm (" eorb d6, d7 "); */
            /* asm (" movb d6, a5%(9) "); */
            /* asm (" movb a4%, d6 "); */
            /* asm (" eorb d6, d7 "); */
            /* asm (" movb d6, a5%(9) "); */
            /* asm (" movb a4%, d6 "); */
            /* asm (" eorb d6, d7 "); */
            /* asm (" movb d6, a5%(9) "); */
            /* asm (" movb a4%, d6 "); */
            /* asm (" eorb d6, d7 "); */
            /* asm (" movb d6, a5%(9) "); */
        } while (--i != -1); /* optimizes to DBRA */
    } else {
        i--;
        do {
            asm (" movb a4%, a5%(9) ");
            /* asm (" movb a4%, d6 "); */
            /* asm (" eorb d6, d7 "); */
            /* asm (" movb d6, a5%(9) "); */
        } while (--i != -1);
    }
#endif
}

#ifdef MOREASM
}
#endif
if (prochk(p, SPERFORM)) {

```

```

        p->pd_err = "didn't get to perform state";
        goto haderr;
    }
    return; /* will pick up status next intr */
}
p->pd_err = "invalid state";
haderr:
do {
    bp->b_resid = p->pd_bcount;
    bp->b_flags |= B_ERROR;
    ERROR(("dev %d: %s\n", p->prodata, p->pd_err));
    prorelse(p, bp);
    p->pd_state = SERR;
} while (prostart(p));
}

/*
 * Get in sync with disk.
 * (subroutine FINDD2 & CHKRSR from 'profrom.text' document)
 * Expects enable==true and Direction==in at start.
 * If disk response is 'state' then returns 0 (success)
 * otherwise fails (returns -1 if timeout and cur state if bad state).
 */
prochk(p, state)
    register struct prodata *p;
{
    register struct device_d *devp = p->pd_da;
    register zero = 0;
    register i;
    int resp;
    /* while ((devp->d_irb & BSY) == 0); */
    ASSERT((devp->d_irb & BSY) == BSY, prochk, ("state=%d [waiting]", state), goto err)
    /* while ((devp->d_irb & BSY) == 0); */

    if (state == SCMD && (p->prodata == 0)) { /* PP inuse */
        if (l2_rcflag) /* ramp in progress */
            l2ramp(2); /* so help it along */
        ppinuse = 1;
    }
    devp->d_irb |= DRW; /* set input mode */
    devp->d_ddra = zero; /* set port A bits to input */
    devp->d_irb &= ~CMD; /* set cmd and enable bufs */

    i = RSPTIME; /* about lms */
    while (devp->d_irb & BSY && i--); /* wait sig that resp byte is ready */

    resp = PIDL; /* reply to use if resp byte wrong */
    if (i > 0) { /* didn't timeout */
        i = devp->d_ira; /* get response from disk */
        if (i == state) /* got correct state */
            resp = PGO; /* reply to use if resp byte correct */
        devp->d_irb &= ~(DRW|CMD); /* set dir=out cmd=true */
        devp->d_ddra = 0xFF; /* set port A bits to output */
        devp->d_ira = resp; /* send reply (GO or RESET) */
        devp->d_ier = FIRQ|FCAL; /* enable interrupts */
        devp->d_irb |= CMD; /* sig disk to read resp */
        p->pd_state = p->pd_nextst; /* setup next state */
        return (i == state) ? 0 : i;
    }
err:
    if (p->prodata == 0) ppinuse = 0; /* reset ppinuse flag */
    p->pd_state = SERR;
    p->pd_err = "EXCESSIVE DISK DELAY -- (is the drive plugged in?)";
    ERROR(("dev %d: %s\n", p->prodata, p->pd_err));
    devp->d_ddra = zero; /* set port A bits to input */
    devp->d_irb |= CMD|DRW; /* set dir=in, disable buffers */
    devp->d_ier = ~FIRQ; /* disable all interrupts */
    return (-1);
}

/* ARGSUSED */
proioctl(dev, cmd, addr, flag)
    dev_t dev;
    int cmd;
    caddr_t addr;
    int flag; /* NOTUSED */

```

```
{
    struct prodata *p = &prodata[physical(dev)];
    switch (cmd) {
    case TIOCGETP:
        if (copyout((caddr_t)&p->pd_flags, addr, sizeof(p->pd_flags)))
            u.u_error = EFAULT;
        break;
    case TIOCSETP:
        if (copyin(addr, (caddr_t)&p->pd_flags, sizeof(p->pd_flags)))
            u.u_error = EFAULT;
        break;
    default:
        u.u_error = ENOTTY;
    }
}
proprint(dev, str)
char *str;
{
    printf("%s on pro drive %d, slice %d\n", str, (dev>>4)&0xF, dev&7);
}
```

```

/* proto.c 4.22 82/06/20 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/mmu.h"
#include "sys/sysmacros.h"
#include "net/misc.h"
#include "net/socket.h"
#include "net/protosw.h"
#include "net/in.h"
#include "net/in_sysm.h"

/*
 * Protocol configuration table and routines to search it.
 *
 * SHOULD INCLUDE A HEADER FILE GIVING DESIRED PROTOCOLS
 */

/*
 * Local protocol handler.
 */

/*
 * TCP/IP protocol family: IP, ICMP, UDP, TCP.
 */
int ip_output();
int ip_init(), ip_slowtimo(), ip_drain();
int icmp_input();
int udp_input(), udp_ctlinput();
int udp_usrreq();
int udp_init();
int tcp_input(), tcp_ctlinput();
int tcp_usrreq();
int tcp_init(), tcp_fasttimo(), tcp_slowtimo(), tcp_drain();
int rip_input(), rip_output();

/*
 * IMP protocol family: raw interface.
 * Using the raw interface entry to get the timer routine
 * in is a kludge.
 */
#include "net/imp.h"
#if NIMP > 0
int rimp_output(), hostslowtimo();
#endif

/*
 * PUP-I protocol family: raw interface
 */
#include "net/pup.h"
#if NPUP > 0
int rpup_output();
#endif

/*
 * Sundries.
 */
int raw_init(), raw_usrreq(), raw_input(), raw_ctlinput();

struct protosw protosw[] = {
{ 0, 0, 0, 0,
  0, ip_output, 0, 0,
  0,
  ip_init, 0, ip_slowtimo, ip_drain,
},
{ 0, PF_INET, IPPROTO_ICMP, 0,
  icmp_input, 0, 0, 0,
  0, 0, 0, 0,
},
{ SOCK_DGRAM, PF_INET, IPPROTO_UDP, PR_ATOMIC|PR_ADDR,
  udp_input, 0, udp_ctlinput, 0,
  udp_usrreq,
  udp_init, 0, 0, 0,
},

```

```

{ SOCK_STREAM, PF_INET, IPPROTO_TCP, PR_CONNREQUIRED|PR_WANTRCVD,
  tcp_input, 0, tcp_ctlinput, 0,
  tcp_usrreq,
  tcp_init, tcp_fasttimo, tcp_slowtimo, tcp_drain,
},
{ 0, 0, 0, 0,
  raw_input, 0, raw_ctlinput, 0,
  raw_usrreq,
  raw_init, 0, 0, 0,
},
#ifdef SRI
{ SOCK_RAW, PF_INET, 15, PR_ATOMIC|PR_ADDR,
  rip_input, rip_output, 0, 0,
  raw_usrreq,
  0, 0, 0, 0,
},
#endif
{ SOCK_RAW, PF_INET, IPPROTO_RAW, PR_ATOMIC|PR_ADDR,
  rip_input, rip_output, 0, 0,
  raw_usrreq,
  0, 0, 0, 0,
}
}
#if NIMP > 0
{ SOCK_RAW, PF_IMPLINK, 0, PR_ATOMIC|PR_ADDR,
  0, rimp_output, 0, 0,
  raw_usrreq,
  0, 0, hostslowtimo, 0,
}
#endif
#if NPUP > 0
{ SOCK_RAW, PF_PUP, 0, PR_ATOMIC|PR_ADDR,
  0, rpup_output, 0, 0,
  raw_usrreq,
  0, 0, 0, 0,
}
#endif
};

#define NPROTOSW (sizeof(protosw) / sizeof(protosw[0]))

struct protosw *protoswLAST = &protosw[NPROTOSW-1];

/*
 * Operations on protocol table and protocol families.
 */

/*
 * Initialize all protocols.
 */
pffinit()
{
    register struct protosw *pr;

    for (pr = protoswLAST; pr >= protosw; pr--)
        if (pr->pr_init)
            (*pr->pr_init)();
}

/*
 * Find a standard protocol in a protocol family
 * of a specific type.
 */
struct protosw *
pffindtype(family, type)
    int family, type;
{
    register struct protosw *pr;

    if (family == 0)
        return (0);
    for (pr = protosw; pr <= protoswLAST; pr++)
        if (pr->pr_family == family && pr->pr_type == type)
            return (pr);
    return (0);
}

```

```

}

/*
 * Find a specified protocol in a specified protocol family.
 */
struct protosw *
pffindproto(family, protocol)
    int family, protocol;
{
    register struct protosw *pr;

    if (family == 0)
        return (0);
    for (pr = protosw; pr <= protoswLAST; pr++)
        if (pr->pr_family == family && pr->pr_protocol == protocol)
            return (pr);
    return (0);
}

#ifdef notdef
pfcctlinput(cmd, arg)
    int cmd;
    caddr_t arg;
{
    register struct protosw *pr;

    for (pr = protosw; pr <= protoswLAST; pr++)
        if (pr->pr_ctlinput)
            (*pr->pr_ctlinput)(cmd, arg);
}
#endif

/*
 * Slow timeout on all protocols.
 */
pfslowtimo()
{
    register struct protosw *pr;

    for (pr = protoswLAST; pr >= protosw; pr--)
        if (pr->pr_slowtimo)
            (*pr->pr_slowtimo)();
}

pffasttimo()
{
    register struct protosw *pr;

    for (pr = protoswLAST; pr >= protosw; pr--)
        if (pr->pr_fasttimo)
            (*pr->pr_fasttimo)();
}

```

```

/* pty.c 4.21 82/03/23 */

/*
 * Pseudo-teletype Driver
 * (Actually two drivers, requiring two entries in 'cdevsw')
 */

/*
 * billn -- 12/15/82. Mercilessly hacked for system 3. To do
 * Remote input editing, etc, need to rework it again
 * from the original.
 */

#include "net/pty.h"

#if NPTY > 0
#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/mmu.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/tty.h"
#include "sys/ttold.h"
#include "sys/termio.h"
#include "sys/ioctl.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/errno.h"
#include "sys/user.h"
#include "sys/conf.h"
#include "sys/buf.h"
#include "sys/file.h"
#include "sys/proc.h"
#include "sys/var.h"
#include "net/misc.h"

#define BUFSIZ 100 /* Chunk size iomoved from user */

/*
 * pts == /dev/tty[pP]?
 * ptc == /dev/pty[pP]?
 */
struct tty pt_tty[NPTY];
struct pt_ioctl {
    int pt_flags;
    int pt_gensym;
    struct proc *pt_selr, *pt_selw;
    int pt_send;
} pt_ioctl[NPTY];

#define PF_RCOLL 0x01
#define PF_WCOLL 0x02
#define PF_NBIO 0x04
#define PF_PKT 0x08 /* packet mode */
#define PF_STOPPED 0x10 /* user told stopped */
#define PF_REMOTE 0x20 /* remote and flow controlled input */
#define PF_NOSTOP 0x40
#define PF_WTIMER 0x80 /* waiting for timer to flush */
/* billn -- kludge */
#define PF_PTSOPEN 0x100 /* pts side is open */
#define PF_PTCOPEN 0x200 /* ptc side is open */

/*ARGSUSED*/
ptsopen(dev, flag)
    dev_t dev;
{
    register struct tty *tp;
    register struct pt_ioctl *pti = &pt_ioctl[minor(dev)];

    if (dev >= NPTY) {
        u.u_error = ENXIO;
        return;
    }
    tp = &pt_tty[dev];
    if ((tp->t_state & ISOPEN) == 0) {

```

```

/* No features (nor raw mode) */
tp->t_cflag = 0; tp->t_oflag = 0;
tp->t_iflag = 0; tp->t_lflag = 0;
ttinit(tp); /* Set up default chars */
}
if (tp->t_proc) /* Ctrlr still around. */
    tp->t_state |= CARR_ON;
while ((tp->t_state & CARR_ON) == 0) {
    tp->t_state |= WOPEN;
    (void) sleep((caddr_t)&tp->t_rawq, TTIPRI);
}
(*linesw[tp->t_line].l_open)(tp);
pti->pt_flags |= PF_PTSOPEN;
}

ptsclose(dev)
    dev_t dev;
{
    register struct tty *tp;
    register struct pt_ioctl *pti = &pt_ioctl[minor(dev)];

    tp = &pt_tty[dev];
    (*linesw[tp->t_line].l_close)(tp);
    pti->pt_flags &= ~PF_PTSOPEN;
    if ((pti->pt_flags & PF_PTCOPEN) == 0) /* other end already gone? */
        tp->t_proc = 0;
}

ptsread(dev)
    dev_t dev;
{
    register struct tty *tp = &pt_tty[dev];
    register struct pt_ioctl *pti = &pt_ioctl[minor(dev)];

    if (tp->t_proc)
        (*linesw[tp->t_line].l_read)(tp);
    wakeup((caddr_t)&tp->t_rawq.c_cf);
    if (pti->pt_selw) {
        selwakeup(pti->pt_selw, pti->pt_flags & PF_WCOLL);
        pti->pt_selw = 0;
        pti->pt_flags &= ~PF_WCOLL;
    }
}

/*
 * Write to pseudo-tty.
 * Wakeups of controlling tty will happen
 * indirectly, when tty driver calls ptsstart.
 */
ptswrite(dev)
    dev_t dev;
{
    register struct tty *tp;

    tp = &pt_tty[dev];
    if (tp->t_proc)
        (*linesw[tp->t_line].l_write)(tp);
}

ptcwakeup(tp)
    struct tty *tp;
{
    struct pt_ioctl *pti = &pt_ioctl[tp - &pt_tty[0]];
    int s = spl5(); /* any NZ spl will lockout ptcimer */

    if (pti->pt_selr) {
        selwakeup(pti->pt_selr, pti->pt_flags & PF_RCOLL);
        pti->pt_selr = 0;
        pti->pt_flags &= ~PF_RCOLL;
    }
    if (pti->pt_selw) {
        selwakeup(pti->pt_selw, pti->pt_flags & PF_WCOLL);
        pti->pt_selw = 0;
        pti->pt_flags &= ~PF_WCOLL;
    }
}

```

```

    wakeup((caddr_t)&tp->t_outq.c_cf);
    splx(s);
}

ptctimer()
{
    register struct tty *tp = &pt_tty[0];
    register struct pt_ioctl *pti = &pt_ioctl[0];
    register i;

    timeout(ptctimer, (caddr_t)0, v.v_hz >> 2);
    for (i=0; i<NPTY; i++, pti++, tp++) {
        if ((pti->pt_flags & PF_WTIMER) == 0)
            continue;
        pti->pt_flags &= ~PF_WTIMER;
        if (tp->t_proc == 0)
            continue;
        ptcwakeup(tp);
    }
}

/*ARGSUSED*/
ptcopen(dev, flag)
dev_t dev;
int flag;
{
    register struct tty *tp;
    struct pt_ioctl *pti;
    static first;
    extern int ptsstart();

    if (first == 0) {
        first++;
        ptctimer();
    }
    if (dev >= NPTY) {
        u.u_error = ENXIO;
        return;
    }
    tp = &pt_tty[dev];
    if (tp->t_proc) {
        u.u_error = EIO;
        return;
    }
    tp->t_iflag = ICRNL|ISTRIP|IGNPAR;
    tp->t_oflag = OPOST|ONLCR|TAB3;
    tp->t_lflag = ISIG|ICANON; /* no echo */
    tp->t_proc = ptsstart;
    if (tp->t_state & WOPEN)
        wakeup((caddr_t)&tp->t_rawq);
    tp->t_state |= CARR_ON;
    pti = &pt_ioctl[dev];
    pti->pt_flags = 0;
    pti->pt_send = 0;
    pti->pt_flags |= PF_PTCOPEN;
}

ptcclose(dev)
dev_t dev;
{
    register struct tty *tp;
    register struct pt_ioctl *pti = &pt_ioctl[minor(dev)];

    tp = &pt_tty[dev];
    if (tp->t_state & ISOPEN)
        signal(tp->t_pgrp, SIGHUP);
    tp->t_state &= ~CARR_ON; /* virtual carrier gone */
    ttyflush(tp, FREAD, FWRITE);
    pti->pt_flags &= ~PF_PTCOPEN;
    if ((pti->pt_flags & PF_PTSOPEN) == 0) /* other end already gone? */
        tp->t_proc = 0; /* mark closed */
}

ptcread(dev)
dev_t dev;
{
    register struct tty *tp;
    register struct pt_ioctl *pti;
    register c;

    tp = &pt_tty[dev];
    if ((tp->t_state & (CARR_ON|ISOPEN)) == 0)
        return;
    pti = &pt_ioctl[dev];
    if (pti->pt_flags & PF_PKT) {
        if (pti->pt_send) {
            (void) passc(pti->pt_send);
            pti->pt_send = 0;
            return;
        }
        (void) passc(0);
    }
    while (tp->t_outq.c_cc == 0 || (tp->t_state & TTSTOP)) {
        if (pti->pt_flags & PF_NBIO) {
            u.u_error = EWOULDBLOCK;
            return;
        }
        (void) sleep((caddr_t)&tp->t_outq.c_cf, TTIPRI);
    }
    while (tp->t_outq.c_cc && (c = getc(&tp->t_outq)) >= 0)
        if (passc(c) < 0)
            break;
    tp->t_state &= ~BUSY;
    if (tp->t_state & OASLP) {
        tp->t_state &= ~OASLP;
        wakeup((caddr_t)&tp->t_outq);
    }
    if (tp->t_state & TTIOV && tp->t_outq.c_cc == 0) {
        tp->t_state &= ~TTIOV;
        wakeup((caddr_t)&tp->t_oflag);
    }
}

/*
 * System 5 does not have nbio normally
 */
/* #define IF_NBIO */

ptcwrite(dev)
dev_t dev;
{
    register struct tty *tp;
    register char *cp, *ce;
    register int cc, c, ctmp;
    char locbuf[BUFSIZ];

#ifdef IF_NBIO
    int cnt = 0;
#endif

    tp = &pt_tty[dev];
    if ((tp->t_state & (CARR_ON|ISOPEN)) == 0)
        return;
    do {
        cc = MIN(u.u_count, BUFSIZ);
        cp = locbuf;
        iomove(cp, cc, B_WRITE);
        if (u.u_error)
            break;
        ce = cp + cc;
    }
    again:
    while (cp < ce) {
        while (tp->t_delct && tp->t_rawq.c_cc >= TTYHOG - 2) {
            wakeup((caddr_t)&tp->t_rawq);
        }
#ifdef IF_NBIO
        if (tp->t_state & TS_NBIO) {
            u.u_count += ce - cp;
            if (cnt == 0)
                u.u_error = EWOULDBLOCK;
            return;
        }
#endif
    }
}

#endif

```



```

        /* Better than just flushing it! */
        /* Wait for something to be read */
        (void) sleep((caddr_t)&tp->t_rawq.c_cf, TTOPRI);
        goto again;
    }
    c = *cp++;
    if (tp->t_iflag & IXON) {
        ctmp = c & 0177;
        if (tp->t_state & TTSTOP) {
            if (c == CSTART || tp->t_iflag & IXANY)
                (*tp->t_proc)(tp, T_RESUME);
        } else
            if (c == CSTOP)
                (*tp->t_proc)(tp, T_SUSPEND);
        if (c == CSTART || c == CSTOP)
            continue;
    }
    if (tp->t_rbuf.c_ptr != NULL) {
        if (tp->t_iflag & ISTRIP)
            c &= 0177;
        *tp->t_rbuf.c_ptr = c;
        tp->t_rbuf.c_count--;
        (*linesw[tp->t_line].l_input)(tp);
    }
}

#ifdef IF_NBIO
    cnt++;
#endif
} while (u.u_count);
}

ptcioc1(dev, cmd, addr, flag)
caddr_t addr;
dev_t dev;
{
    register struct tty *tp = &pt_tty[minor(dev)];
    register struct pt_ioctl *pti = &pt_ioctl[dev];

    if (cmd == TIOCPKT) {
        int packet;
        if (copyin((caddr_t)addr, (caddr_t)&packet, sizeof(packet))) {
            u.u_error = EFAULT;
            return;
        }
        if (packet)
            pti->pt_flags |= PF_PKT;
        else
            pti->pt_flags &= ~PF_PKT;
        return;
    }
    if (cmd == FIONBIO) {
        int nbio;
        if (copyin((caddr_t)addr, (caddr_t)&nbio, sizeof(nbio))) {
            u.u_error = EFAULT;
            return;
        }
        if (nbio)
            pti->pt_flags |= PF_NBIO;
        else
            pti->pt_flags &= ~PF_NBIO;
        return;
    }
    /* IF CONTROLLER STTY THEN MUST FLUSH TO PREVENT A HANG ??? */
    if ((cmd==TIOCSETP)|| (cmd==TCSETAW)) {
        while (getc(&tp->t_outq) >= 0);
        tp->t_state &= ~BUSY;
    }
    ptsioc1(dev, cmd, addr, flag);
}

/*ARGSUSED*/
ptsioc1(dev, cmd, addr, flag)
register caddr_t addr;
register dev_t dev;
{
    register struct tty *tp = &pt_tty[dev];

```

```

register struct pt_ioctl *pti = &pt_ioctl[dev];
register int stop;

```

```

if (ttiocom(tp, cmd, (int)addr, dev) == 0)

```

```

;
/* else...?? */
stop = tp->t_iflag&IXON;
if (pti->pt_flags & PF_NOSTOP) {
    if (stop) {
        pti->pt_send &= TIOCPKT_NOSTOP;
        pti->pt_send |= TIOCPKT_DOSTOP;
        pti->pt_flags &= ~PF_NOSTOP;
        ptcwakeup(tp);
    }
} else {
    if (stop == 0) {
        pti->pt_send &= ~TIOCPKT_DOSTOP;
        pti->pt_send |= TIOCPKT_NOSTOP;
        pti->pt_flags |= PF_NOSTOP;
        ptcwakeup(tp);
    }
}
}

```

```

ptsstart(tp, cmd)

```

```

register struct tty *tp;

```

```

{

```

```

    register struct pt_ioctl *pti = &pt_ioctl[tp - &pt_tty[0]];
    extern ttrstr();

```

```

    switch(cmd) {

```

```

    case T_TIME:
        tp->t_state &= ~TIMEOUT;
        goto start;

```

```

    case T_WFLUSH:

```

```

        if (tp->t_outq.c_cc) {
            while (getc(&tp->t_outq) >= 0)
                ;
            tp->t_state &= ~BUSY;
        }
        /* fall through */

```

```

    case T_RESUME:

```

```

        tp->t_state &= ~TTSTOP;
        wakeup((caddr_t)&tp->t_outq.c_cf);
        /* fall through */

```

```

    case T_OUTPUT:

```

```

start:

```

```

        if (tp->t_state&(TIMEOUT|TTSTOP|BUSY))
            break;

```

```

        if (tp->t_state&TTIOW && tp->t_outq.c_cc==0) {
            tp->t_state &= ~TTIOW;
            wakeup((caddr_t)&tp->t_oflag);
        }

```

```

        if (pti->pt_flags & PF_STOPPED) {
            pti->pt_flags &= ~PF_STOPPED;
            pti->pt_send = TIOCPKT_START;
        }

```

```

        if (tp->t_outq.c_cc < 200) {
            pti->pt_flags |= PF_WTIMER;
            return;
        }

```

```

        pti->pt_flags &= ~PF_WTIMER;
        tp->t_state |= BUSY;
        ptcwakeup(tp);

```

```

        if (tp->t_state&OASLP &&
            tp->t_outq.c_cc <= tlowat[tp->t_cflag&CBAUD]) {
            tp->t_state &= ~OASLP;
            wakeup((caddr_t)&tp->t_outq);
        }

```

```

        break;

```

```

    case T_SUSPEND:

```

```

        tp->t_state |= TTSTOP;

```

```

    pti->pt_flags |= PF_STOPPED;
    pti->pt_send |= TIOCPRT_STOP;
    break;

case T_BLOCK:
    tp->t_state |= TBLOCK;
    tp->t_state &= ~TTXON;
    if (tp->t_outq.c_cc > 0)
        wakeup((caddr_t)&tp->t_outq.c_cf);
    break;

case T_RFLUSH:
    if (!(tp->t_state&TBLOCK))
        break;

case T_UNBLOCK:
    tp->t_state &= ~(TTXOFF|TBLOCK);
    if (tp->t_outq.c_cc > 0)
        wakeup((caddr_t)&tp->t_outq.c_cf);
    break;

case T_BREAK:
    tp->t_state |= TIMEOUT;
    timeout(ttrstrt, (caddr_t)tp, v.v_hz/4);
    break;
}
}

ptcselect(dev, rw)
dev_t dev;
int rw;
{
    register struct tty *tp = &pt_tty[minor(dev)];
    struct pt_ioctl *pti = &pt_ioctl[minor(dev)];
    struct proc *p;
    int s;

    if ((tp->t_state&(CARR_ON|ISOPEN)) == 0)
        return (1); /* ??? billn */
    s = spl7();
    switch (rw) {

case FREAD:
        if (tp->t_outq.c_cc && (tp->t_state&TTSTOP) == 0) {
            splx(s);
            return (1);
        }
        if ((p = pti->pt_selr) && p->p_wchan == (caddr_t)&selwait)
            pti->pt_flags |= PF_RCOLL;
        else
            pti->pt_selr = u.u_procp;
        break;

case FWRITE:
        splx(s);
        return 1;
#endif notdef
        /*
         * only do this if using "PF_REMOTE" ala 4.1a . So why keep
         * it? -- history...
         */
        if (tp->t_rawq.c_cc == 0) {
            splx(s);
            return (1);
        }
        if ((p = pti->pt_selw) && p->p_wchan == (caddr_t)&selwait)
            pti->pt_flags |= PF_WCOLL;
        else
            pti->pt_selw = u.u_procp;
#endif
        break;
    }
    splx(s);
    return (0);
}
#endif

```

```
/* pup.c 4.2 82/06/20 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/system.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "net/in.h"
#include "net/in_system.h"
#include "net/af.h"
#include "net/pup.h"

#ifdef PUP
pup_hash(spup, hp)
    struct sockaddr_pup *spup;
    struct afhash *hp;
{
    hp->afh_nethash = spup->spup_addr.pp_net;
    hp->afh_hosthash = spup->spup_addr.pp_host;
    if (hp->afh_hosthash < 0)
        hp->afh_hosthash = -hp->afh_hosthash;
}

pup_netmatch(spup1, spup2)
    struct sockaddr_pup *spup1, *spup2;
{
    return (spup1->spup_addr.pp_net == spup2->spup_addr.pp_net);
}
#endif
```

```

/*      raw_cb.c      4.16      83/02/10      */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "net/misc.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "net/mbuf.h"
#include "net/protosw.h"

#include "net/if.h"
#include "net/raw_cb.h"

/*
 * Routines to manage the raw protocol control blocks.
 *
 * TODO:
 *   hash lookups by protocol family/protocol + address family
 *   take care of unique address problems per AF?
 *   redo address binding to allow wildcards
 */

/*
 * Allocate a control block and a nominal amount
 * of buffer space for the socket.
 */
raw_attach(so)
    register struct socket *so;
{
    struct mbuf *m;
    register struct rawcb *rp;

    /* billn, meld w/ old note don't gtclr -- will it work?
    m = m_getclr(M_DONTWAIT, MT_PCB);
    */
    m = m_get(M_DONTWAIT);
    if (m == 0)
        return (ENOBUFS);
    if (sbreserve(&so->so_snd, RAWSNDQ) == 0)
        goto bad;
    if (sbreserve(&so->so_rcv, RAWRCVQ) == 0)
        goto bad2;
    rp = mtod(m, struct rawcb *);
    rp->rcb_socket = so;
    insque(rp, &rawcb);
    so->so_pcb = (caddr_t)rp;
    rp->rcb_pcb = 0;
    return (0);

bad2:
    sbrelease(&so->so_snd);

bad:
    (void) m_free(m);
    return (ENOBUFS);
}

/*
 * Detach the raw connection block and discard
 * socket resources.
 */
raw_detach(rp)
    register struct rawcb *rp;
{
    struct socket *so = rp->rcb_socket;

    so->so_pcb = 0;
    sofree(so);
    remque(rp);
    m_freem(dtom(rp));
}

```

```

/*
 * Disconnect and possibly release resources.
 */
raw_disconnect(rp)
    struct rawcb *rp;
{
    rp->rcb_flags &= ~RAW_FADDR;
    /* billn -- meld with old...
    if (rp->rcb_socket->so_state & SS_NOFDREF)
    */
    if (rp->rcb_socket->so_state & SS_USERGONE)
        raw_detach(rp);
}

#ifdef notdef
raw_bind(so, nam)
    register struct socket *so;
    struct mbuf *nam;
{
    struct sockaddr *addr = mtod(nam, struct sockaddr *);
    register struct rawcb *rp;

    if (ifnet == 0)
        return (EADDRNOTAVAIL);
}
/*
#include "../h/domain.h"
*/
#include "net/in.h"
#include "net/in_system.h"
/* BEGIN DUBIOUS */
/*
 * Should we verify address not already in use?
 * Some say yes, others no.
 */
switch (addr->sa_family) {
case AF_IMPLINK:
case AF_INET:
    if (((struct sockaddr_in *)addr)->sin_addr.s_addr &&
        if_ifwithaddr(addr) == 0)
        return (EADDRNOTAVAIL);
    break;
}

#endif PUP
/*
 * Curious, we convert PUP address format to internet
 * to allow us to verify we're asking for an Ethernet
 * interface. This is wrong, but things are heavily
 * oriented towards the internet addressing scheme, and
 * converting internet to PUP would be very expensive.
 */
case AF_PUP: {
#include "../netpup/pup.h"
    struct sockaddr_pup *spup = (struct sockaddr_pup *)addr;
    struct sockaddr_in inpup;

    bzero((caddr_t)&inpup, (unsigned)sizeof(inpup));
    inpup.sin_family = AF_INET;
    inpup.sin_addr.s_net = spup->sp_net;
    inpup.sin_addr.s_impno = spup->sp_host;
    if (inpup.sin_addr.s_addr &&
        if_ifwithaddr((struct sockaddr *)&inpup) == 0)
        return (EADDRNOTAVAIL);
    break;
}
}

#endif
default:
    return (EAFNOSUPPORT);
}
/* END DUBIOUS */
rp = sotorawcb(so);

```

```
        bcopy((caddr_t)addr, (caddr_t)&rp->rcb_laddr, sizeof (*addr));
        rp->rcb_flags |= RAW_LADDR;
        return (0);
    }
#endif

/*
 * Associate a peer's address with a
 * raw connection block.
 */
raw_connaddr(rp, nam)
    struct rawcb *rp;
    struct mbuf *nam;
{
    struct sockaddr *addr = mtod(nam, struct sockaddr *);

    bcopy((caddr_t)addr, (caddr_t)&rp->rcb_faddr, sizeof(*addr));
    rp->rcb_flags |= RAW_FADDR;
}
```

```

/*    raw_ip.c        4.13    82/06/20    */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/system.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/socket.h"
#include "net/protosw.h"
#include "net/socketvar.h"
#include "net/if.h"
#include "net/in.h"
#include "net/in_system.h"
#include "net/ip.h"
#include "net/ip_var.h"
#include "net/raw_cb.h"
#include "net/route.h"
#include "errno.h"

/*
 * Raw interface to IP protocol.
 */

static struct sockaddr_in ripdst = { AF_INET };
static struct sockaddr_in ripsrc = { AF_INET };
static struct sockproto ripproto = { PF_INET };
/*
 * Setup generic address and protocol structures
 * for raw_input routine, then pass them along with
 * mbuf chain.
 */
rip_input(m)
    struct mbuf *m;
{
    register struct ip *ip = mtod(m, struct ip *);
    register int hlen = ip->ip_hl << 2;

    ripproto.sp_protocol = ip->ip_p;
    ripdst.sin_addr = ip->ip_dst;
    ripsrc.sin_addr = ip->ip_src;
    m_adj(m, hlen);
    raw_input(m, &ripproto, (struct sockaddr *)&ripsrc,
              (struct sockaddr *)&ripdst);
}

/*
 * Generate IP header and pass packet to ip_output.
 * Tack on options user may have setup with control call.
 */
rip_output(m0, so)
    struct mbuf *m0;
    struct socket *so;
{
    register struct mbuf *m;
    register struct ip *ip;
    int len = 0, error;
    struct rawcb *rp = sotorawcb(so);
    struct sockaddr_in *sin;

    /*
     * Calculate data length and get an mbuf
     * for IP header.
     */
    for (m = m0; m; m = m->m_next)
        len += m->m_len;
    m = m_get(M_DONTWAIT);
    if (m == 0) {
        error = ENOBUFS;
        goto bad;
    }

    /*
     * Fill in IP header as needed.
     */
}

```

```

m->m_off = MMAXOFF - sizeof(struct ip);
m->m_len = sizeof(struct ip);
m->m_next = m0;
ip = mtod(m, struct ip *);
ip->ip_p = so->so_proto->pr_protocol;
ip->ip_len = sizeof(struct ip) + len;
if (rp->rcb_flags & RAW_LADDR) {
    sin = (struct sockaddr_in *)&rp->rcb_laddr;
    if (sin->sin_family != AF_INET) {
        error = EAFNOSUPPORT;
        goto bad;
    }
    ip->ip_src.s_addr = sin->sin_addr.s_addr;
} else
    ip->ip_src.s_addr = 0;
ip->ip_dst = ((struct sockaddr_in *)&rp->rcb_faddr)->sin_addr;
ip->ip_ttl = MAXTTL;
return (ip_output(m, (struct mbuf *)0, (struct route *)0, 1));

bad:
    m_freem(m);
    return (error);
}

```

```

/*      raw_usrreq.c      4.25      83/02/10      */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "net/protosw.h"
#include "net/in.h"
#include "net/in_sysm.h"
#include "net/if.h"
#include "net/af.h"
#include "errno.h"
#include "net/raw_cb.h"

/*
 * Initialize raw connection block q.
 */
raw_init()
{
    rawcb.rcb_next = rawcb.rcb_prev = &rawcb;
    rawinrq.ifq_maxlen = IFQ_MAXLEN;
}

/*
 * Raw protocol interface.
 */
raw_input(m0, proto, src, dst)
    struct mbuf *m0;
    struct sockproto *proto;
    struct sockaddr *src, *dst;
{
    register struct mbuf *m;
    struct raw_header *rh;
    int s;

    /*
     * Rip off an mbuf for a generic header.
     */
    /* billn -- meld with old
     m = m_get(M_DONTWAIT, MT_HEADER);
     */
    m = m_get(M_DONTWAIT);
    if (m == 0) {
        m_freem(m0);
        return;
    }
    m->m_next = m0;
    m->m_len = sizeof(struct raw_header);
    rh = mtod(m, struct raw_header *);
    rh->raw_dst = *dst;
    rh->raw_src = *src;
    rh->raw_proto = *proto;

    /*
     * Header now contains enough info to decide
     * which socket to place packet in (if any).
     * Queue it up for the raw protocol process
     * running at software interrupt level.
     */
    s = splimp();
    if (IF_QFULL(&rawinrq))
        m_freem(m);
    else
        IF_ENQUEUE(&rawinrq, m);
    splx(s);
    schednetisr(NETISR_RAW);
}

```

```

}

/*
 * Raw protocol input routine.  Process packets entered
 * into the queue at interrupt time.  Find the socket
 * associated with the packet(s) and move them over.  If
 * nothing exists for this packet, drop it.
 */
rawintr()
{
    int s;
    struct mbuf *m;
    register struct rawcb *rp;
    register struct protosw *lproto;
    register struct raw_header *rh;
    struct socket *last;

next:
    s = splimp();
    IF_DEQUEUE(&rawinrq, m);
    splx(s);
    if (m == 0)
        return;
    rh = mtod(m, struct raw_header *);
    last = 0;
    for (rp = rawcb.rcb_next; rp != &rawcb; rp = rp->rcb_next) {
        lproto = rp->rcb_socket->so_proto;
        if (lproto->pr_family != rh->raw_proto.sp_family)
            continue;
        if (lproto->pr_protocol &&
            lproto->pr_protocol != rh->raw_proto.sp_protocol)
            continue;

        /*
         * We assume the lower level routines have
         * placed the address in a canonical format
         * suitable for a structure comparison.
         */
#define equal(a1, a2) \
        (bcmp((caddr_t)&(a1), (caddr_t)&(a2), sizeof (struct sockaddr)) == 0)
        if ((rp->rcb_flags & RAW_LADDR) &&
            !equal(rp->rcb_laddr, rh->raw_dst))
            continue;
        if ((rp->rcb_flags & RAW_FADDR) &&
            !equal(rp->rcb_faddr, rh->raw_src))
            continue;
        if (last) {
            struct mbuf *n;
            if ((n = m_copy(m->m_next, 0, (int)M_COPYALL)) == 0)
                goto nospace;
            if (sbappendaddr(&last->so_rcv, &rh->raw_src, n) == 0) {
                /* should notify about lost packet */
                m_freem(n);
                goto nospace;
            }
            sorwakeup(last);
        }
    }
nospace:
    last = rp->rcb_socket;
}
if (last) {
    m = m_free(m); /* header */
    if (sbappendaddr(&last->so_rcv, &rh->raw_src, m) == 0)
        goto drop;
    sorwakeup(last);
    goto next;
}
drop:
    m_freem(m);
    goto next;
}

/*ARGSUSED*/
raw_ctlinput(cmd, arg)
    int cmd;
    caddr_t arg;
{
}

```

```

    if (cmd < 0 || cmd > PRC_NCMLS)
        return;
    /* INCOMPLETE */
}

/*ARGSUSED*/
raw_usrreq(so, req, m, nam)
struct socket *so;
int req;
struct mbuf *m, *nam;
{
    register struct rawcb *rp = sotorawcb(so);
    int error = 0;

    if (rp == 0 && req != PRU_ATTACH)
        return (EINVAL);

    switch (req) {

    /*
     * Allocate a raw control block and fill in the
     * necessary info to allow packets to be routed to
     * the appropriate raw interface routine.
     */
    case PRU_ATTACH:
        if ((so->so_state & SS_PRIV) == 0)
            return (EACCESS);

        if (rp)
            return (EINVAL);
        error = raw_attach(so);
        break;

    /*
     * Destroy state just before socket deallocation.
     * Flush data or not depending on the options.
     */
    case PRU_DETACH:
        if (rp == 0)
            return (ENOTCONN);
        raw_detach(rp);
        break;

    /*
     * If a socket isn't bound to a single address,
     * the raw input routine will hand it anything
     * within that protocol family (assuming there's
     * nothing else around it should go to).
     */
    case PRU_CONNECT:
        if (rp->rcb_flags & RAW_FADDR)
            return (EISCONN);
        raw_connaddr(rp, nam);
        soisconnected(so);
        break;

    case PRU_DISCONNECT:
        if ((rp->rcb_flags & RAW_FADDR) == 0)
            return (ENOTCONN);
        raw_disconnect(rp);
        soisdisconnected(so);
        break;

    /*
     * Mark the connection as being incapable of further input.
     */
    case PRU_SHUTDOWN:
        socantsendmore(so);
        break;

    /*
     * Ship a packet out. The appropriate raw output
     * routine handles any massaging necessary.
     */
    case PRU_SEND:
        if (nam) {
            if (rp->rcb_flags & RAW_FADDR)
                return (EISCONN);
            raw_connaddr(rp, nam);
        } else if ((rp->rcb_flags & RAW_FADDR) == 0)
            return (ENOTCONN);
        error = (*so->so_proto->pr_output)(m, so);
        if (nam)
            rp->rcb_flags &= ~RAW_FADDR;
        break;

    case PRU_ABORT:
        raw_disconnect(rp);
        sofree(so);
        soisdisconnected(so);
        break;

    /*
     * Not supported.
     */
    case PRU_ACCEPT:
    case PRU_RCVD:
    case PRU_CONTROL:
    case PRU_SENSE:
    case PRU_RCVOOB:
    case PRU_SENDOOB:
        error = EOPNOTSUPP;
        break;

    case PRU_SOCKADDR:
        bcopy((caddr_t)&rp->rcb_laddr, mtod(nam, caddr_t),
            sizeof (struct sockaddr));
        nam->m_len = sizeof (struct sockaddr);
        break;

    default:
        panic("raw_usrreq");
    }
    return (error);
}

```



```

/* @(#)rdwri.c 1.4 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/inode.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/proc.h"
#include "sys/errno.h"
#include "sys/buf.h"
#include "sys/conf.h"
#include "sys/file.h"
#include "sys/system.h"
#include "sys/tty.h"

/*
 * Read the file corresponding to
 * the inode pointed at by the argument.
 * The actual read arguments are found
 * in the variables:
 *   u_base      core address for destination
 *   u_offset    byte offset in file
 *   u_count     number of bytes to read
 *   u_segflg   read to kernel/user/user I
 */
readi(ip)
register struct inode *ip;
{
    register struct user *up;
    register struct buf *bp;
    register dev_t dev;
    register daddr_t bn;
    register unsigned on, n;
    register type;
    register struct tty *tp;
    struct cdevsw *cdevp;

    up = &u;
    if (up->u_count == 0)
        return;
    if (up->u_offset < 0) {
        up->u_error = EINVAL;
        return;
    }
    type = ip->i_mode&IFMT;
    switch(type) {
    case IFCHR:
        dev = (dev_t)ip->i_rdev;
        ip->i_flag |= IACC;
        cdevp = &cdevsw[(short)major(dev)];
        if (tp = cdevp->d_ttys) {
            tp += (short)(minor(dev)&077);
            (*linesw[(short)tp->t_line].l_read)(tp);
        } else
            (*cdevp->d_read)(minor(dev));
        break;

    case IFIFO:
        while (ip->i_size == 0) {
            if (ip->i_fwcnt == 0)
                return;
            if (up->u_fmode&FNDELAY)
                return;
            ip->i_fflag |= IFIR;
            prele(ip);
            (void) sleep((caddr_t)&ip->i_fwcnt, PPIPE);
            plock(ip);
        }
        up->u_offset = ip->i_frptr;

    case IFBLK:
    case IFREG:
    case IFDIR:
    do {
        bn = bmap(ip, B_READ);
        if (up->u_error)

```

```

        break;
    on = up->u_pboff;
    if ((n = up->u_pbsize) == 0)
        break;
    dev = up->u_pbdev;
    if ((long)bn<0) {
        if (type != IFREG)
            break;
        bp = getebk();
        clrbuf(bp);
    } else if (up->u_rablock)
        bp = breada(dev, bn, up->u_rablock);
    else
        bp = bread(dev, bn);
    if (bp->b_resid) {
        n = 0;
    }
    if (n!=0)
        iomove(bp->b_un.b_addr+on, (int)n, B_READ);
    if (type == IFIFO) {
        ip->i_size -= n;
        if (up->u_offset >= PIPSIZ)
            up->u_offset = 0;
        if ((on+n) == FsBSIZE(dev) && ip->i_size < (PIPSIZ-FsBSIZE(dev)))
            bp->b_flags &= ~B_DELRWI;
    }
    brelse(bp);
    ip->i_flag |= IACC;
    } while (up->u_error==0 && up->u_count!=0 && n!=0);
    if (type == IFIFO) {
        if (ip->i_size)
            ip->i_frptr = up->u_offset;
        else {
            ip->i_frptr = 0;
            ip->i_fwptr = 0;
        }
        if (ip->i_fflag&IFIW) {
            ip->i_fflag &= ~IFIW;
            curpri = PPIPE;
            wakeup((caddr_t)&ip->i_fwcnt);
        }
    }
    break;
}
default:
    up->u_error = ENODEV;
}

/*
 * Write the file corresponding to
 * the inode pointed at by the argument.
 * The actual write arguments are found
 * in the variables:
 *   u_base      core address for source
 *   u_offset    byte offset in file
 *   u_count     number of bytes to write
 *   u_segflg   write to kernel/user/user I
 */
writei(ip)
register struct inode *ip;
{
    register struct user *up;
    register struct buf *bp;
    register dev_t dev;
    register daddr_t bn;
    register unsigned n, on;
    register type;
    register struct tty *tp;
    struct cdevsw *cdevp;

    up = &u;
    if (up->u_offset < 0) {
        up->u_error = EINVAL;
        return;
    }

```

```

}
type = ip->i_mode&IFMT;
switch (type) {
case IFCHR:
    dev = (dev_t)ip->i_rdev;
    ip->i_flag |= IUPD|ICHG;
    cdevp = &cdevsw[(short)major(dev)];
    if (tp = cdevp->d_ttys) {
        tp += (short)(minor(dev)&077);
        (*linesw[(short)tp->t_line].l_write)(tp);
    } else
        (*cdevp->d_write)(minor(dev));
    break;
case IFIFO:
loop:
    usave = 0;
    while ((up->u_count+ip->i_size) > PIPSI2) {
        if (ip->i_frct == 0)
            break;
        if ((up->u_count > PIPSI2) && (ip->i_size < PIPSI2)) {
            usave = up->u_count;
            up->u_count = PIPSI2 - ip->i_size;
            usave -= up->u_count;
            break;
        }
        if (up->u_fmode&FNDELAY)
            return;
        ip->i_fflag |= IFIW;
        prele(ip);
        (void) sleep((caddr_t)&ip->i_fwcnt, PPIPE);
        plock(ip);
    }
    if (ip->i_frct == 0) {
        up->u_error = EPIPE;
        psignal(up->u_procp, SIGPIPE);
        break;
    }
    up->u_offset = ip->i_fwptr;
case IFBLK:
case IFREG:
case IFDIR:
while (up->u_error==0 && up->u_count!=0) {
    bn = bmap(ip, B_WRITE);
    if (up->u_error)
        break;
    on = up->u_pboff;
    n = up->u_pbsize;
    dev = up->u_pbdev;
    if (n == FsBSIZE(dev))
        bp = getblk(dev, bn);
    else if (type==IFIFO && on==0 && ip->i_size < (PIPSI2-FsBSIZE(dev)))
        bp = getblk(dev, bn);
    else
        bp = bread(dev, bn);

    iomove(bp->b_un.b_addr+on, (int)n, B_WRITE);
    if (up->u_error)
        brelease(bp);
    else if (up->u_fmode&FSYNC)
        bwrite(bp);
    else if (type == IFBLK) {
        /* IFBLK not delayed for tapes */
        bp->b_flags |= B_AGE;
        bawrite(bp);
    } else
        bdwrite(bp);
    if (type == IFREG || type == IFDIR) {
        if (up->u_offset > ip->i_size)
            ip->i_size = up->u_offset;
    } else if (type == IFIFO) {
        ip->i_size += n;
        if (up->u_offset == PIPSI2)
            up->u_offset = 0;
    }
}
ip->i_flag |= IUPD|ICHG;

```

```

}
if (type == IFIFO) {
    ip->i_fwptr = up->u_offset;
    if (ip->i_fflag&IFIR) {
        ip->i_fflag &= ~IFIR;
        curpri = PPIPE;
        wakeup((caddr_t)&ip->i_frct);
    }
    if (up->u_error==0 && usave!=0) {
        up->u_count = usave;
        goto loop;
    }
}
break;
default:
    up->u_error = ENODEV;
}
}
/*
 * Move n bytes at byte location
 * &bp->b_un.b_addr[0] to/from (flag) the
 * user/kernel (u.segflg) area starting at u.base.
 * Update all the arguments by the number
 * of bytes moved.
 */
iomove(cp, n, flag)
register caddr_t cp;
register n;
{
    register struct user *up;
    register t;

    if (n==0)
        return;
    up = &u;
    if (up->u_segflg != 1) {
        if (flag==B_WRITE)
            t = copyin(up->u_base, (caddr_t)cp, n);
        else
            t = copyout((caddr_t)cp, up->u_base, n);
        if (t) {
            up->u_error = EFAULT;
            return;
        }
    }
    else
        if (flag == B_WRITE)
            bcopy(up->u_base, (caddr_t)cp, n);
        else
            bcopy((caddr_t)cp, up->u_base, n);
    up->u_base += n;
    up->u_offset += n;
    up->u_count -= n;
    return;
}

```

```
char _Version[] = "(C) Copyright 1984 UniSoft Corp., Version V.1.0";
char _Origin[] = "UniSoft Systems of Berkeley";
```

```
/*
 * reboot rootdev
 *   Change rootdev, pipedev, dumpdev, swapdev and nswap
 *   in the incore copy of unix and then restart unix.
 *   This is used during installation after a minimal
 *   filesystem has been set up on the hard disk. It
 *   results in the first boot of unix from the hard disk.
 */

#include "stdio.h"
#include "nmaddrs.h"
#include "sys/types.h"
#include "sys/config.h"
#include "sys/sysmacros.h"
#include "sys/swapsz.h"
#include "sys/reboot.h"

#define USAGE "usage: reboot rootdev"

main(argc, argv)
char **argv;
{
    register fp, cfp;
    short rootdev;
    long nswap;
    int i;

    if (argc != 2)
        perr(USAGE);
    rootdev = (short)strtol(*(argv+1), (char **)NULL, 16);
    if ((fp = open("/dev/mem", 2)) < 0)
        perr("cannot open /dev/mem");

    if (lseek(fp, ROOTDEV, 0) < 0)
        perr("lseek to rootdev %x failed", ROOTDEV);
    if (write(fp, &rootdev, 2) != 2)
        perr("write of rootdev 0x%x at %x failed", rootdev, ROOTDEV);
    printf("rootdev = 0x%x\n", rootdev);

    if (lseek(fp, PIPEDEV, 0) < 0)
        perr("lseek to pipedev %x failed", PIPEDEV);
    if (write(fp, &rootdev, 2) != 2)
        perr("write of pipedev 0x%x at %x failed", rootdev, PIPEDEV);
    printf("pipedev = 0x%x\n", rootdev);

    if (lseek(fp, DUMPDEV, 0) < 0)
        perr("lseek to dumpdev %x failed", DUMPDEV);
    if (write(fp, &rootdev, 2) != 2)
        perr("write of dumpdev 0x%x at %x failed", rootdev, DUMPDEV);
    printf("dumpdev = 0x%x\n", rootdev);

    rootdev++; /* now it's swapdev */
    if (lseek(fp, SWAPDEV, 0) < 0)
        perr("lseek to swapdev %x failed", SWAPDEV);
    if (write(fp, &rootdev, 2) != 2)
        perr("write of swapdev 0x%x at %x failed", rootdev, SWAPDEV);
    printf("swapdev = 0x%x\n", rootdev);

    if (lseek(fp, NSWAP, 0) < 0)
        perr("lseek to nswap %x failed", NSWAP);
    if (major(rootdev) == PR0)
        nswap = PRNSWAP;
    else if (major(rootdev) == CV2)
        nswap = CVNSWAP;
    else if (major(rootdev) == PM3)
        nswap = PMNSWAP;
    else
        perr("cannot determine size of swapdev");
    if (write(fp, &nswap, 4) != 4)
        perr("write of nswap %d at %x failed", nswap, NSWAP);
    printf("nswap = %d\n", nswap);
    for (i=0; i<200000; i++) ;
}
```

```
if ((cfp = open("/dev/console", 2)) < 0)
    perr("cannot open /dev/console");
ioctl(cfp, RESTART, (caddr_t)0); /* jump to start of unix */
perr("restart failed");
}

perr(mes, par)
char *mes, *par;
{
    fprintf(stderr, mes, par);
    fprintf(stderr, "\n");
    perror("reboot");
    exit(1);
}
```

```

/*
 * (C) 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with Unisoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by Unisoft.
 *
 * reinit.c - reinitialize parts of data segment for restarting unix
 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/file.h"
#include "sys/tty.h"
#include "sys/termio.h"
#include "sys/conf.h"
#include "sys/sysinfo.h"
#include "sys/var.h"
#include "sys/reg.h"
#include "setjmp.h"
#include "sys/kb.h"
#include "sys/sysmacros.h"
#include "sys/iobuf.h"
#include "sys/map.h"
#define CMAPSIZ 50 /* also in conf.c */
#define SMAPSIZ 50 /* also in conf.c */

extern mpid; /* used by newproc to decide whether first "init" or not */

extern struct iostat prostat[];
extern struct iobuf protab;
extern struct iostat snstat[];
extern struct iobuf snstab;
extern struct iostat cvstat[];
extern struct iobuf cvtab;

extern char *kb_keytab;
extern char ToLA[];
extern char kb_altkp;
extern int (*te_putc)();
extern int vt_putc();
extern char vt_tabset[];
/*
 * reinit - called from RESTART console ioctl call in unix
 */
reinit()
{
    extern int teslotsused;
    register i;

    mpid = 0;

    retabinit(&protab, PR0, prostat);
    retabinit(&snstab, SN1, snstat);
    retabinit(&cvtab, CV2, cvstat);

    kb_keytab = ToLA;
    kb_shft = kb_lock = kb_altkp = 0;
    te_putc = vt_putc;
    for (i = 0; i < 88; i++)
        vt_tabset[i] = 0;

    remapinit(&coremap[0], CMAPSIZ);
    remapinit(&swapmap[0], SMAPSIZ);
    /*
     * reset tecmar four port card
     */
    teslotsused = 0;
}

retabinit(tab, dev, stat)
register struct iobuf *tab;
struct iostat stat[];
{
    tab->b_flags = 0;
    tab->b_forw = 0;
    tab->b_back = 0;
    tab->b_actf = 0;
    tab->b_actl = 0;
    tab->b_dev = makedev(dev, 0);
    tab->b_active = 0;
    tab->b_errcnt = 0;
    tab->io_erec = 0;
    tab->io_nreg = 0;
    tab->io_addr = 0;
    tab->io_stp = stat;
    tab->io_start = 0;
    tab->io_s1 = 0;
    tab->io_s2 = 0;
}

remapinit(map, szmap)
struct map *map;
{
    map->m_size = szmap - 2;
    map->m_addr = 0;
    for (map++, szmap--; szmap > 0; map++, szmap--) {
        map->m_size = 0;
        map->m_addr = 0;
    }
}

```

```
/*
 * (C) 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with Unisoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by Unisoft.
 *
 * "rom_mon" is used by the software power off feature (see kb.c)
 * and the reboot system call (see config.c, doboot) to return to
 * the ROM monitor in "Customer Mode".
 */
```

```
#include "sys/mmu.h"
typedef int (*pfri)();
#define ROMADDR (pfri) (0xFE0084)
```

```
rom_mon()
{
    asm(" movl    #0,d0");      /* no error code */
    asm(" subl   a2,a2");      /* no icon */
    asm(" subl   a3,a3");      /* no message */
    asm(" movl   0x1000,sp");
    ROMADDR();
}
```

```

/* route.c 4.16 83/02/10 */
#include "net/misc.h"
#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/system.h"
#include "net/mbuf.h"
#include "net/socket.h"
#include "net/protosw.h"
#include "sys/ioctl.h"
#include "net/in.h"
#include "net/in_system.h"
#include "net/if.h"
#include "net/af.h"

#include "net/route.h"
#include "errno.h"

int rttrash; /* routes not in table but not freed */
/*
 * Packet routing routines.
 */
rtalloc(ro)
register struct route *ro;
{
    register struct rtentry *rt, *rtmin;
    register struct mbuf *m;
    register unsigned hash;
    register int (*match)();
    struct afhash h;
    struct sockaddr *dst = &ro->ro_dst;
    int af = dst->sa_family;

    if (ro->ro_rt && ro->ro_rt->rt_ifp) /* XXX */
        return;
    if (af >= AF_MAX)
        return;
    (*afswitch[af].af_hash)(dst, &h);
    rtmin = 0, hash = h.afh_hosthash;
    for (m = rthost[hash % RTHASHSIZ]; m; m = m->m_next) {
        rt = m->rt;
        if (rt->rt_hash != hash)
            continue;
        if ((rt->rt_flags & RTF_UP) == 0 ||
            (rt->rt_ifp->if_flags & IFF_UP) == 0)
            continue;
        if (bcmp((caddr_t)&rt->rt_dst, (caddr_t)dst, sizeof(*dst)))
            continue;
        if (rtmin == 0 || rt->rt_use < rtmin->rt_use)
            rtmin = rt;
    }
    if (rtmin)
        goto found;

    hash = h.afh_nethash;
    match = afswitch[af].af_netmatch;
    for (m = rtnet[hash % RTHASHSIZ]; m; m = m->m_next) {
        rt = m->rt;
        if (rt->rt_hash != hash)
            continue;
        if ((rt->rt_flags & RTF_UP) == 0 ||
            (rt->rt_ifp->if_flags & IFF_UP) == 0)
            continue;
        if (rt->rt_dst.sa_family != af || !(*match)(rt->rt_dst, dst))
            continue;
        if (rtmin == 0 || rt->rt_use < rtmin->rt_use)
            rtmin = rt;
    }
}
found:
ro->ro_rt = rtmin;
if (rtmin)
    rtmin->rt_refcnt++;
}

rtfree(rt)
    register struct rtentry *rt;
{
    if (rt == 0)
        panic("rtfree");
    rt->rt_refcnt--;
    if (rt->rt_refcnt == 0 && (rt->rt_flags & RTF_UP) == 0) {
        rttrash++;
        (void) m_free(dtom(rt));
    }
}

/*
 * Carry out a request to change the routing table. Called by
 * interfaces at boot time to make their "local routes" known
 * and for ioctl's.
 */
rtrequest(req, entry)
int req;
register struct rtentry *entry;
{
    register struct mbuf *m, **mprev;
    register struct rtentry *rt;
    struct afhash h;
    unsigned hash;
    int af, s, error = 0, (*match)();
    struct ifnet *ifp;

    af = entry->rt_dst.sa_family;
    if (af >= AF_MAX)
        return (EAFNOSUPPORT);
    (*afswitch[af].af_hash)(entry->rt_dst, &h);
    if (entry->rt_flags & RTF_HOST) {
        hash = h.afh_hosthash;
        mprev = &rthost[hash % RTHASHSIZ];
    } else {
        hash = h.afh_nethash;
        mprev = &rtnet[hash % RTHASHSIZ];
    }
    match = afswitch[af].af_netmatch;
    s = splimp();
    for (; m = *mprev; mprev = &m->m_next) {
        rt = m->rt;
        if (rt->rt_hash != hash)
            continue;
        if (entry->rt_flags & RTF_HOST) {
#define equal(a1, a2) \
        (bcmp((caddr_t)(a1), (caddr_t)(a2), sizeof(struct sockaddr)) == 0)
            if (!equal(&rt->rt_dst, &entry->rt_dst))
                continue;
        } else {
            if (rt->rt_dst.sa_family != entry->rt_dst.sa_family ||
                (*match)(rt->rt_dst, &entry->rt_dst) == 0)
                continue;
        }
        if (equal(&rt->rt_gateway, &entry->rt_gateway))
            break;
    }
    switch (req) {
    case SIOCDELRT:
        if (m == 0) {
            error = ESRCH;
            goto bad;
        }
        *mprev = m->m_next;
        if (rt->rt_refcnt > 0) {
            rt->rt_flags &= ~RTF_UP;
            rttrash++;
            m->m_next = 0;
        } else
            (void) m_free(m);
        break;
    case SIOCADDRT:
        if (m) {

```

```
        error = EEXIST;
        goto bad;
    }
    ifp = if_ifwithaddr(&entry->rt_gateway);
    if (ifp == 0) {
        ifp = if_ifwithnet(&entry->rt_gateway);
        if (ifp == 0) {
            error = ENETUNREACH;
            goto bad;
        }
    }
    /* billn -- meld with old
    m = m_get(M_DONTWAIT, MT_RTABLE);
    */
    m = m_get(M_DONTWAIT);
    if (m == 0) {
        error = ENOBUFS;
        goto bad;
    }
    *mprev = m;
    m->m_off = MMIOFF;
    m->m_len = sizeof (struct rtenry);
    rt = mtod(m, struct rtenry *);
    rt->rt_hash = hash;
    rt->rt_dst = entry->rt_dst;
    rt->rt_gateway = entry->rt_gateway;
    rt->rt_flags =
        RTF_UP | (entry->rt_flags & (RTF_HOST|RTF_GATEWAY));
    rt->rt_refcnt = 0;
    rt->rt_use = 0;
    rt->rt_ifp = ifp;
    break;
}
bad:
    splx(s);
    return (error);
}
/*
 * Set up a routing table entry, normally
 * for an interface.
 */
rtinit(dst, gateway, flags)
    struct sockaddr *dst, *gateway;
    int flags;
{
    struct rtenry route;

    bzero((caddr_t)&route, sizeof (route));
    route.rt_dst = *dst;
    route.rt_gateway = *gateway;
    route.rt_flags = flags;
    (void) rtrequest((int)SIOCADDRT, &route);
}

```

```

/* #define      HOWFAR */
/*
 * Real time clock driver
 *
 * (C) 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with Unisoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by Unisoft.
 *
 * Only reads and writes of 4 bytes (sizeof time_t, the standard unix
 * representation of time) are allowed for the real time clock.
 * The rtime structure contains the time most recently read or written.
 */
struct rtime {
    See page 35 LHRM
    time_t  rt_tod;      Seconds since the Epoch (unix time)
    long    rt_alm;      Seconds remaining to trigger alarm (unused)
    short   rt_year;     year          (0 - 15)
    short   rt_day;      julian day     (1 - 366)
    short   rt_hour;     hour           (0 - 23)
    short   rt_min;      minute         (0 - 59)
    short   rt_sec;      second         (0 - 59)
    short   rt_tenth;    tenths of a second (0 - 9)
};
*/
#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/utsname.h"
#include "sys/buf.h"
#include "sys/eelog.h"
#include "sys/erec.h"
#include "sys/iobuf.h"
#include "sys/system.h"
#include "sys/var.h"
#include "setjmp.h"
#include "sys/reg.h"
#include "sys/tty.h"
#include "sys/local.h"

struct rtime rtime;
int rtcwait;      /* flag to sleep on waiting for read to complete */

/*
 * Read the real time clock. The command to do this is issued through
 * the COPS. In response, the keyboard gets 6 special interrupts with
 * the data. kbintr stores this data in rtime and calls rtcsettod when
 * the sixth one has been handled. rtcsettod calculates the time as
 * unix likes to think of it, which is as it is returned.
 */
/* ARGSUSED */
rtcread(dev)
dev_t dev;
{
    time_t rt;

    if (u.u_count != sizeof(time_t)) {
        u.u_error = ENXIO;
        return;
    }
    rt = rtdoread();
    iomove((caddr_t)&rt, sizeof(time_t), B_READ);
}

/*
 * This routine is normally called from rtcread. However the kernel
 * reads the clock by calling rtdoread directly from clkset.
 */
rtdoread()

```

```

{
    l2copscmd(READCLOCK);
    rtcwait = 1;
    while (rtcwait)
        (void) sleep((caddr_t)&rtcwait, TTIPRI);
    return(rtime.rt_tod);
}

/*
 * Set rt_tod (real time in seconds since epoch) given the real
 * time clock time currently in the rest of the rtime structure.
 * Those values (rt_year, rt_day, rt_hour, rt_min, rt_sec) are
 * set in response to l2copscmd(READCLOCK), and are returned through
 * special keyboard interrupts (kbintr).
 */
#define dsize(x) (x%4==0 ?366 :365)
#define YEAR 70
rtcsettod ()
{
    register struct rtime *p = &rtime;
    register short i, j;

    i = -1;
    for (j=YEAR; j-YEAR<p->rt_year; j++)
        i += dsize(j);
    p->rt_tod = i + p->rt_day;      /* Days since epoch */
    p->rt_tod *= 24;
    p->rt_tod += p->rt_hour;
    p->rt_tod *= 60;
    p->rt_tod += p->rt_min;
    p->rt_tod *= 60;
    p->rt_tod += p->rt_sec;
#ifdef HOWFAR
    printf("DATE: %d.%d %d:%d.%d\n", p->rt_year, p->rt_day,
           p->rt_hour, p->rt_min, p->rt_sec);
#endif
    if (rtcwait) {
        rtcwait = 0;
        wakeup((caddr_t)&rtcwait);
    }
}

/*
 * Set the real time clock to the time indicated. Note that nt is in
 * seconds since midnight 1/1/1970 GMT. The timezone has already been
 * corrected for.
 */
/* ARGSUSED */
rtcwrite(dev)
dev_t dev;
{
    register struct rtime *p = &rtime;
    register long nt;
    register long i, j;

    if (u.u_count != sizeof(time_t)) {
        u.u_error = ENXIO;
        return;
    }
    iomove((caddr_t)&rtime.rt_tod, sizeof(time_t), B_WRITE);
    nt = p->rt_tod;
    i = nt % 86400;      /* 86400 = 24*60*60 = number secs./day */
    j = nt / 86400;
    if (i < 0) {
        i += 86400;
        j -= 1;
    }
    nt = j;
    p->rt_sec = i % 60;
    i /= 60;
    p->rt_min = i % 60;
    i /= 60;
    p->rt_hour = i % 24;
    j = YEAR;
    for (j=YEAR; i-dsize(j); j++) {
        if (nt < i)

```



```
        break;
    nt -- i;
}
p->rt_year = j - YEAR;
if (p->rt_year < 10) {
#ifdef HOWFAR
    printf("can't remember dates before 1980\n");
#endif HOWFAR
    u.u_error = EINVAL;
    return;
}
p->rt_day = ++nt;
l2copscmd(SETCLOCK); /* stop clock, start setup mode */
for (i=0; i<5; i++)
    l2copscmd(CLKNIABLE); /* no alarm implemented yet */
l2copscmd(CLKNIABLE | (p->rt_year - 10));
i = p->rt_day / 10;
j = i / 10;
l2copscmd((char)(CLKNIABLE | j));
l2copscmd((char)(CLKNIABLE | (i % 10)));
l2copscmd(CLKNIABLE | (p->rt_day % 10));
l2copscmd(CLKNIABLE | (p->rt_hour / 10));
l2copscmd(CLKNIABLE | (p->rt_hour % 10));
l2copscmd(CLKNIABLE | (p->rt_min / 10));
l2copscmd(CLKNIABLE | (p->rt_min % 10));
l2copscmd(CLKNIABLE | (p->rt_sec / 10));
l2copscmd(CLKNIABLE | (p->rt_sec % 10));
l2copscmd(STRTCLOCK); /* start clock, leave timer disabled */
/* l2copscmd(READCLOCK); /* Now read it back */
}
```

```

/*
 * (C) 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with Unisoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by Unisoft.
 *
 * co.c - "console" (ie, bitmap screen and keyboard) driver for the lisa.
 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/file.h"
#include "sys/tty.h"
#include "sys/termio.h"
#include "sys/conf.h"
#include "sys/sysinfo.h"
#include "sys/var.h"
#include "sys/req.h"
#include "setjmp.h"
#include "sys/ioctl.h"
#include "sys/kb.h"
#include "sys/al_ioctl.h"
#ifdef SUNIX
#include "sys/reboot.h"
#endif
#include "sys/mmu.h"
#include "sys/cops.h"
#include "sys/l2.h"

int coproc();
extern int co_cnt;
extern struct tty co_tty[];
extern struct ttyptr co_ttptr[];

extern char bmbck, bmnorm; /* pointer to screen -- initialized in bminit */
extern char *bmscrn;

/* calls to the putc routine are made indirectly through
 * the te_putc pointer which is used to
 * keep track of the current state for escape character
 * processing, ie, although initialized to point to
 * the normal putc, an escape character causes other
 * functions to process the next character(s)
 */
extern int vt_putc();
int (*te_putc)()=vt_putc;
#ifdef SUNIX
extern caddr_t start;
#endif
*/

struct device {
    char csr; /* Command status register */
    char idum[2]; /* fillers */
    char dbuf; /* data buffer */
};

/* ARGSUSED */
coopen(dev, flag)
register dev;
{
    register struct tty *tp;

    if (dev >= co_cnt) {
        u.u_error = ENXIO;
        return;
    }
}

}

tp = co_ttptr[dev].tt_tty;
tp->t_index = dev;
SPL6();
if ((tp->t_states(ISOPEN|WOPEN)) == 0) {
    tp->t_proc = coproc;
    ttinit(tp);
    tp->t_state = WOPEN | CARR_ON;
    if (dev == CONSOLE) {
        tp->t_iflag = ICRNL | ISTRIP;
        tp->t_oflag = OPOST | ONLCR | TAB3;
        tp->t_lflag = ISIG | ICANON | ECHO | ECHOK;
        tp->t_cflag = cspeed | CS8 | CREAD | HUPCL;
    }
}
SPL0();
(*linesw[tp->t_line].l_open)(tp);
}

/* ARGSUSED */
coclose(dev, flag)
{
    register struct tty *tp;

    tp = co_ttptr[dev].tt_tty;
    (*linesw[tp->t_line].l_close)(tp);
}

coread(dev)
{
    struct tty *tp;

    tp = co_ttptr[dev].tt_tty;
    (*linesw[tp->t_line].l_read)(tp);
}

cowrite(dev)
{
    struct tty *tp;

    tp = co_ttptr[dev].tt_tty;
    (*linesw[tp->t_line].l_write)(tp);
}

/* ARGSUSED */
coioctl(dev, cmd, arg, mode)
{
    int i;

    switch (cmd) {
    case AL_SBVOL:
        l2_bvol = arg & 7;
        break;
    case AL_SBPITCH:
        l2_bpitch = arg & 0x1FFF;
        break;
    case AL_SBTIME:
        if (arg <= 0)
            arg = 1;
        if (arg > 10 * v.v_hz) /* limit to 10 seconds */
            arg = 10 * v.v_hz;
        l2_btime = arg;
        break;
    case AL_SDIMTIME:
        l2_dtime = arg;
        l2_dtrap = lbolt + l2_dtime;
        break;
    case AL_SDIMCONT:
        l2_dimcont = (~arg) & 0xFF;
        break;
    case AL_SDIMRATE:
        l2_crate = arg;
        break;
    case AL_SCONTRAST:
        l2_defcont = (~arg) & 0xFF;
    }
}

```

```

        l2_desired = l2_defcont;
        l2ramp(0);
        break;
case AL_SREPWAIT:
    kb_repwait = arg;
    break;
case AL_SREPDELAY:
    kb_repdelay = arg;
    break;
case AL_GBVOL:
    i = l2_bvol;
    if (copyout((caddr_t)&i, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GBPITCH:
    if (copyout((caddr_t)&l2_bpitch, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GBTIME:
    if (copyout((caddr_t)&l2_bttime, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GDINTIME:
    if (copyout((caddr_t)&l2_dtime, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GDIMCONT:
    i = (~l2_dimcont) & 0xFF;
    if (copyout((caddr_t)&i, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GDIMRATE:
    if (copyout((caddr_t)&l2_crate, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GCONTRAST:
    i = (~l2_defcont) & 0xFF;
    if (copyout((caddr_t)&i, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GREPWAIT:
    i = kb_repwait & 0xFFFF;
    if (copyout((caddr_t)&i, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GREPDELAY:
    i = kb_repdelay & 0xFFFF;
    if (copyout((caddr_t)&i, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_GBMADDR:
    if (copyout((caddr_t)&bmscrn, (caddr_t)arg, 4))
        u.u_error = EFAULT;
    break;
case AL_REVVIDEO:
    if (arg > 0)
        i = 0;
    else if (arg == 0)
        i = -1;
    else
        i = (bmbck)? 0 : -1;
    if (bmbck != i) {
        bmswitch();
        bmsinv();
    }
    bmbck = i;
    bmnormal = bmbck;
    break;
#endif SUNIX
case RESTART: /* jump to the start of unix */
    reinit();
    ((int (*)())0xC000)();
    break;
#endif SUNIX
default:
    (void) ttiocom(co_ttptr[0].tt_tty, cmd, arg, mode);

```

```

        break;
    }
}
coproc(tp, cmd)
register struct tty *tp;
{
    register struct cblock *tbuf;
    extern ttrstrt();

    switch (cmd) {
case T_TIME:
    tp->t_state &= ~TIMEOUT;
    goto start;

case T_WFLUSH:
    tbuf = &tp->t_tbuf;
    tbuf->c_size -= tbuf->c_count;
    tbuf->c_count = 0;
    /* fall through */
case T_RESUME:
    tp->t_state &= ~TTSTOP;
    goto start;

case T_OUTPUT:
    if (tp->t_state & (TTSTOP|TIMEOUT|BUSY))
        break;
    tbuf = &tp->t_tbuf;
    if ((tbuf->c_ptr == 0) || (tbuf->c_count == 0)) {
        if (tbuf->c_ptr)
            tbuf->c_ptr -= tbuf->c_size;
        if (!(CPRES & (*linesw[tp->t_line].l_output)(tp)))
            break;
    }
    (*te_putc)((*tbuf->c_ptr++)&0x7f);
    tbuf->c_count--;
    sysinfo.xmint++; /* this is the xmit interrupt */
    splx(spl1());
    goto start;

case T_SUSPEND:
    tp->t_state |= TTSTOP;
    break;

case T_BLOCK:
    break;

case T_RFLUSH:
    if (!(tp->t_state&TBLOCK))
        break;
    /* fall through */

case T_UNBLOCK:
    break;

case T_BREAK:
    break;
    }

}

cointr(dev)
{
    register struct cblock *cbp;
    register int c, lent, flg;
    struct tty *tp;
    register char ctmp;
    char lbuf[3];

    sysinfo.rcvint++;
    c = kb_chrbuf;
    tp = co_ttptr[dev].tt_tty;
    if (tp->t_rbuf.c_ptr == NULL)
        return;

#ifdef NULLDEBUG
#endif

```

```

    if( c == 0x00 ) {
        sccdebug();
        return;
    }
#endif
    if (tp->t_iflag & IXON) {
        ctmp = c & 0177;
        if (tp->t_state & TTSTOP) {
            if (ctmp == CSTART || tp->t_iflag & IXANY)
                (*tp->t_proc)(tp, T_RESUME);
        } else {
            if (ctmp == CSTOP)
                (*tp->t_proc)(tp, T_SUSPEND);
        }
        if (ctmp == CSTART || ctmp == CSTOP)
            return;
    }
    /*
    * Check for errors
    */
    lcnt = 1;
    flg = tp->t_iflag;
    if (flg&ISTRIP)
        c &= 0177;
    else {
        if (c == 0377 && flg&PARMRK) {
            lbuf[1] = 0377;
            lcnt = 2;
        }
    }
    /*
    * Stash character in r_buf
    */
    cbp = &tp->t_rbuf;
    if (lcnt != 1) {
        lbuf[0] = c;
        while (lcnt) {
            *cbp->c_ptr++ = lbuf[--lcnt];
            if (--cbp->c_count == 0) {
                cbp->c_ptr -= cbp->c_size;
                (*linesw[tp->t_line].l_input)(tp);
            }
        }
        if (cbp->c_size != cbp->c_count) {
            cbp->c_ptr -= cbp->c_size - cbp->c_count;
            (*linesw[tp->t_line].l_input)(tp);
        }
    } else {
        *cbp->c_ptr = c;
        cbp->c_count--;
        (*linesw[tp->t_line].l_input)(tp);
    }
}

/*
* This version of putchar writes directly to the bitmap display
* for those last-ditch situations when you just have to get stuff to the CRT.
*/
coputchar(c)
register c;
{
    (*te_putc)(c & 0x7F);
    if (c == '\n')
        (*te_putc)('\r');
}

```

```

/*
 * Configuration information
 */

/* #define    DISK_0 1 */

#define NBUF 30
#define NINODE 50
#define NFILE 60
#define NMOUNT 8
#define CMAPSIZ 50 /* also in reinit.c */
#define SMAPSIZ 50 /* also in reinit.c */
#define CXMAPSIZ 50
#define NCALL 15
#define NPROC 30
#define NTEXT 20
#define NSVTEXT 20
#define NCLIST 100
#define STACKGAP 8
#define NSABUF 5
#define POWER 0
#define MAXUP 25
#define NHBUF 64
#define NPBUF 4
#define NFLOW 200
#define X25LINKS 1
#define X25BUFS 256
#define X25MAPS 30
#define X25BYTES (16*1024)
#define CSIBNUM 20
#define VPMBSZ 8192
#define MMSG 1
#define MSGMAP 100
#define MSGMAX 8192
#define MSGMNB 16384
#define MSGMNI 50
#define MSGSSZ 8
#define MSGTQL 40
#define MSGSEG 1024
#define SEMA 1
#define SEMMAP 10
#define SEMMNI 10
#define SEMMNS 60
#define SEMMNU 30
#define SEMMSL 25
#define SEMOPH 10
#define SEMUME 10
#define SEMVMX 32767
#define SEMAEM 16384
#define SHMEM 1
#define SHMMAX (128*1024)
#define SHMMIN 1
#define SHMMNI 100
#define SHMBRK 16
#define SHMALL 512
#define STIHBUF (ST_0*4)
#define STORBUF (ST_0*4)
#define STNPRNT (ST_0>>2)
#define STIBSZ 8192
#define STOBSZ 8192

#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/conf.h"
#include "sys/cpuid.h"
#include "sys/space.h"
#include "sys/io.h"
#include "sys/termio.h"
#include "sys/reg.h"
#include "sys/scc.h"
#include "sys/pport.h"
#include "sys/swapsz.h"

extern nodev(), nulldev();
extern proopen(), proread(), prowrite(), prostrategy(), proprint(), proioctl();
extern snbopen(), snbclose(), snbclose(), snbclose(), snread(), snwrite(), snstrategy(), snprint(), sni
extern cvopen(), cvread(), cvwrite(), cvstrategy(), cvprint();
extern pmopen(), pmread(), pmwrite(), pmstrategy(), pmprint(), pmioctl();
extern coopen(), coclose(), coread(), cowrite(), coloctl();
extern syopen(), syread(), sywrite(), syioctl();
extern mmread(), mmwrite();
extern scopen(), scclose(), scread(), scwrite(), scioclt();
extern erropen(), errclose(), errread();
extern proread(), prowrite(), proioctl();
extern ejioclt();
extern msopen(), msclose(), msread(), msioctl();
extern lpopen(), lpclose(), lpwrite(), lpioctl();
extern skopen(), skclose(), skwrite();
extern rtcread(), rtcwrite();
extern teopen(), teclose(), teread(), tewrite(), teioclt();

#ifdef UCB NET
extern int ptsopen(), ptsclose(), ptsread(), ptswrite();
extern int ptcopen(), ptcclose(), ptcread(), ptcwrite();
extern int ptsioclt(), ptcioclt();
#endif

struct bdevsw bdevsw[] = {
    proopen, nulldev, prostrategy, proprint, /* 0 */
    snbopen, snbclose, snstrategy, snprint, /* 1 */
    cvopen, nulldev, cvstrategy, cvprint, /* 2 */
    pmopen, nulldev, pmstrategy, pmprint, /* 3 */
};

struct cdevsw cdevsw[] = {
    coopen, coclose, coread, cowrite, coloclt, 0, /* 0 */
    syopen, nulldev, syread, sywrite, syioclt, 0, /* 1 */
    nulldev, nulldev, mmread, mmwrite, nodev, 0, /* 2 */
    erropen, errclose, errread, nodev, nodev, 0, /* 3 */
    scopen, scclose, scread, scwrite, scioclt, 0, /* 4 */
    proopen, nulldev, proread, prowrite, proioctl, 0, /* 5 */
    snbopen, snbclose, snread, snwrite, snioclt, 0, /* 6 */
    nulldev, nulldev, nodev, nodev, ejioclt, 0, /* 7 */
    lpopen, lpclose, nodev, lpwrite, lpioctl, 0, /* 8 */
    msopen, msclose, msread, nodev, msioctl, 0, /* 9 */
    skopen, skclose, nodev, skwrite, nodev, 0, /* 10 */
    cvopen, nulldev, cvread, cvwrite, nulldev, 0, /* 11 */
    pmopen, nulldev, pmread, pmwrite, pmioctl, 0, /* 12 */
    nulldev, nulldev, rtcread, rtcwrite, nulldev, 0, /* 13 */
    teopen, teclose, teread, tewrite, teioclt, 0, /* 14 */
};

#ifdef UCB NET
    nodev, nodev, nodev, nodev, nodev, 0, /* 15 */
    nodev, nodev, nodev, nodev, nodev, 0, /* 16 */
    nodev, nodev, nodev, nodev, nodev, 0, /* 17 */
    nodev, nodev, nodev, nodev, nodev, 0, /* 18 */
    nodev, nodev, nodev, nodev, nodev, 0, /* 19 */
    ptcopen, ptcclose, ptcread, ptcwrite, ptcioclt, 0, /* ptc 20 */
    ptsopen, ptsclose, ptsread, ptswrite, ptsioclt, 0, /* pts 21 */
#endif

};

int bdevcnt = sizeof(bdevsw)/sizeof(bdevsw[0]);
int cdevcnt = sizeof(cdevsw)/sizeof(cdevsw[0]);

#ifdef SUNIX /* Sony (installation) root filesystem */
dev_t rootdev = makedev(1, 0);
dev_t pipedev = makedev(1, 0);
dev_t dumpdev = makedev(1, 0);
/* nswap and swapdev are set in lisainit in config.c */
dev_t swapdev = makedev(0, 1);
daddr_t swp1 = 0;
int nswap = PRNSWAP;
#else SUNIX /* Profile root filesystem */
#define ROOTBASE 0 /* (port * 16) for port=0,1,2,4,5,7, or 8 */
dev_t rootdev = makedev(0, ROOTBASE);
dev_t pipedev = makedev(0, ROOTBASE);
dev_t dumpdev = makedev(0, ROOTBASE);

```

```

dev_t swapdev = makedev(0, ROOTBASE + 1);
daddr_t swplo = 0;
int nswap = PRNSWAP;
#endif SUNIX

int (*dump)() = nulldev;
int dump_addr = 0x0000;

int (*pwr_clr[])() = {
    (int (*)())0
};

int (*dev_init[])() = {
    (int (*)())0
};

#ifdef SCC_CONSOLE
int sputc();
int (*putchar)() = sputc;
#else
int coputchar();
int (*putchar)() = coputchar;
#endif

#ifdef UCB_NET
#define PTC_DEV 20
int ptc_dev = PTC_DEV;
#endif

int co_cnt = 1;
struct tty co_tty[1];

struct ttyptr co_ttptr[] = {
    1, &co_tty[0], /* tt_addr field not used */
    0,
};

int sc_cnt = NSC;
struct tty_sc_tty[NSC];
char sc_modem[NSC];

struct ttyptr_sc_ttptr[] = {
    0xFCD240, &sc_tty[1],
    0xFCD242, &sc_tty[0],
    0,
};

struct scline_sc_line[] = {
    W9BRESET, (4000000/16), /* clock frequency b */
    W9ARESET, (4000000/16), /* clock frequency a */
};

#if NTE != 0
int te_cnt = NTE;
struct tty_te_tty[NTE];
char te_dparam[NTE];
char te_modem[NTE];

struct ttyptr_te_ttptr[NTE+1]; /* +1 for pstat */
#endif

/*
 * pointers to ttyptr structures for terminal monitoring programs
 */
struct ttyptr *tty_stat[] = {
    co_ttptr,
    sc_ttptr,
#if NTE != 0
    te_ttptr,
#endif
    0
};

/*
 * tty output low and high water marks

```

```

*/
#define TTHIGH
#ifdef TTLOW
#define M 1
#define N 1
#endif
#ifdef TTHIGH
#define M 3
#define N 1
#endif
int tthiwat[16] = {
    0*M, 60*M, 60*M, 60*M, 60*M, 60*M, 60*M, 120*M,
    120*M, 180*M, 180*M, 240*M, 240*M, 240*M, 100*M, 100*M,
};
int ttlowat[16] = {
    0*N, 20*N, 20*N, 20*N, 20*N, 20*N, 20*N, 40*N,
    40*N, 60*N, 60*N, 80*N, 80*N, 80*N, 50*N, 50*N,
};

/*
 * Default terminal characteristics
 */
char ttochar[NCC] = {
    CINTR,
    CQUIT,
    CERASE,
    CKILL,
    CEOF,
    0,
    0,
    0
};

#ifdef lint
/* LINTLIBRARY */
forlint()
{
    bmintr();
    nmkey();
    llintr((struct args *)0);
    kbintr();
    scintr((struct args *)0);
    pmintr((struct args *)0);
    ebintr();
    netintr();
}
#endif

#ifdef UCB_NET
#include <net/misc.h>
#include <net/ubavar.h>
extern struct uba_driver ebdriver;
struct uba_device ubdinit[] = {
    /* driver, unit, addr, flags*/
    { ebdriver, 0, (caddr_t)5, 0x59002908 }, /* net 89 */
    0
};

int iff_noarp = 0; /* 0 -> do ARP; not 0 -> no ARP */
#endif

```

```

/*
 * This file contains
 * 1. oem modifiable configuration personality parameters
 * 2. oem modifiable system specific kernel personality code
 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/map.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/proc.h"
#include "sys/buf.h"
#include "sys/iobuf.h"
#include "sys/reg.h"
#include "sys/file.h"
#include "sys/inode.h"
#include "sys/seg.h"
#include "sys/acct.h"
#include "sys/sysinfo.h"
#include "sys/var.h"
#include "sys/ipc.h"
#include "sys/shm.h"
#include "sys/termio.h"

#include "sys/conf.h"
#include "sys/cops.h"
#include "sys/pport.h"
#include "sys/local.h"
#include "sys/l2.h"
#include "sys/kb.h"
#include "sys/swapsz.h"

/*char oemmsg[] = "UniSoft Systems distribution system release 1.5";*/
char oemmsg[] = "UniSoft Systems pre-distribution system (release 1.5+)";

int  speed = B9600;      /* default console speed */
int  parityno = 28;     /* parity interrupt vector */
int  cmask = CMASK;     /* default file creation mask */
int  cdlimit = CDLIMIT; /* default file size limit */
char slot[NSLOTS];     /* card ID numbers for expansion cards */

/*
 * Kernel initialization functions.
 * Called from main.c while at spl7 in the kernel.
 */
oem7init() /* alias (formerly) "lisainit" */
{
#ifdef SUNIX
    int dev;
    extern dev_t swapdev;
    extern int nswap;
#endif
    extern struct rtime rtime;
    extern int pmvect[];
    extern int tevect[];
    register short *sidp; /* slot ID pointer */
    register slotid, i;
    register long *ip;

    l2init(); /* setup the COPS ports */

    /* This mess disables the verticle retrace interrupt, for now.
    */
    do {
        VRON = 1;
    } while ((STATUS & S_VR) != 0);
    do {
        VROFF = 1;
    } while ((STATUS & S_VR) == 0);

    /* Some of the initialization requires that interrupts be enabled to
    * pick up coded sequences from the keyboard cops. If interrupts were
    * masked out then the time returned by READCLOCK would fill the
    * buffer and KBENABLE, which also returns a value, would have trouble.
    */
    SPL1(); /* ok, do it to me */
    l2copscmd(MOUSEOFF); /* shut off mouse interrupts */
    l2copscmd(READCLOCK); /* get time of day */
    l2copscmd(KBENABLE); /* enable keyboard */

    sninit(); /* Sony initialization */

    /* Wait 'til the clock data (from READCLOCK) and keyboard ID (from
    KBENABLE) have come in, and the keyboard is back in NORMALWAIT */
    while (kb_state);
    time = rtime.rt_tod;

    SPL7(); /* it should be at level 7 for the rest (?) */

    /* Find out what's in each of the expansion slots.
    */
    for (i = 0, sidp = SLOTS; i < NSLOTS; i++, sidp++) {
        slot[i] = 0xFF; /* not supported */
        slotid = *sidp & SLOTMASK;
        if (!slotid) {
            if (iocheck((caddr_t) (STDIO+i*0x4000+1))) {
                printf("Expansion slot %d: quad serial card\n",
                    i+1);
                if (teinit(i) == 0) {
                    /*
                     * point to interrupt vector,
                     * set tecmar quad serial board inter loc,
                     * and initialize hwr
                     */
                    ip = &((long *) 0)[EXPIVECT+devtoslot(i)];
                    *ip = (long)tevect + (long)(devtoslot(i)<<2);
                }
            }
            continue;
        }
        printf("Expansion slot %d: ", i+1);
        switch (slotid) {
            case ID_APLNET:
                printf("aplnet card\n");
                break;
            case ID_PRO:
                printf("ProFile card\n");
                break;
            case ID_2PORT:
                printf("two port card\n");
                slot[i] = PR0; /* valid */
                break;
            case ID_PRIAM:
                printf("Priam card\n");
                ip = &((long *) 0)[EXPIVECT+devtoslot(i)]; /* point to int vector */
                *ip = (long)pmvect + (long)(devtoslot(i)<<2); /* set to Priam intr */
                if (pmcinit(i) == 0) /* initialize controller */
                    slot[i] = PM3; /* valid */
                break;
            default:
                printf("card ID 0x%x\n", slotid);
        }
    }
    scinit(); /* SCC serial initialization */
#ifdef UCB_NET
    netinit();
#endif
    /* Now enable the verticle retrace interrupt, used for the system clock.
    */
    do {
        VRON = 1;
    } while ((STATUS & S_VR) != 0);
#ifdef SUNIX
    SPL0();
    /* This is the first unix booted during installation so find swapdev. */

```

```

if (rootdev == makedev(SN1, 0)) {
    while (chkdev(dev = getdevnam()))
        printf("Unable to use that device\nTry again:\n");
    printf("\n\nswapdev = 0x%x\n\n", dev);
    swapdev = dev;
    if (major(dev) == PR0) nswap = PRNSWAP;
    else if (major(dev) == PM3) nswap = PMNSWAP;
    else if (major(dev) == CV2) nswap = CVNSWAP;
    else panic("cannot determine size of swapdev");
}
#endif SUNIX
}

/*
 * Kernel initialization functions.
 * Called from main.c while at spl0 in the kernel.
 */
oem0init()
{
}

/*
 * parityerror()
 * Called from trap for parity error traps via
 * interrupt level "parityno" (conf.c).
 * Should return non-zero for fatal errors.
 * Should return zero for a transient warning error.
 */
parityerror()
{
    printf("parity error\n");
    return(-1);
}

/*
 * reboot the system
 * called from reboot function
 */
doreboot()
{
    kb_state = SHUTDOWN; /* SHUTDOWN (see kb.c)*/
    SPL7(); /* extreme priority */
    rom_mon(); /* return to the ROM monitor */
    /*NOTREACHED*/
}

/*
 * OEM supplied subroutine called on process exit
 */
/* ARGSUSED */
oemexit(p)
register struct proc *p;
{
#ifdef lint
    /* for lint use p */
    p->p_flag++;
#endif
}

struct device_d *pro_da[NPPDEVS] = {
    /* DEV Description */
    /* 0x00 parallel port */
    (struct device_d *) (STDIO+0x2000), /* 0x10 FPC port 0 slot 1 */
    (struct device_d *) (STDIO+0x2800), /* 0x20 FPC port 1 slot 1 */
    (struct device_d *) (STDIO+0x3000), /* 0x30 FPC port 2 slot 1 !!!*/
    (struct device_d *) (STDIO+0x6000), /* 0x40 FPC port 0 slot 2 */
    (struct device_d *) (STDIO+0x6800), /* 0x50 FPC port 1 slot 2 */
    (struct device_d *) (STDIO+0x7000), /* 0x60 FPC port 2 slot 2 !!!*/
    (struct device_d *) (STDIO+0xA000), /* 0x70 FPC port 0 slot 3 */
    (struct device_d *) (STDIO+0xA800), /* 0x80 FPC port 1 slot 3 */
    (struct device_d *) (STDIO+0xB000) /* 0x90 FPC port 2 slot 3 !!!*/
};
int (*pi_fnc[NPPDEVS])(); /* slots for interrupt handler addresses */

/* Set the interrupt handler for a given parallel port controller.
 */

```

```

setppint(addr, fnc)
struct device_d *addr;
int (*fnc)();
{
    register int i;
    extern int cvint(), prointr(), lpintr();

    for (i=0; i<NPPDEVS; i++)
        if (pro_da[i] == addr) { /* found dev number */
            if (pi_fnc[i]) { /* in use */
                if (pi_fnc[i] == fnc) /* same handler */
                    return 0;
                if (pi_fnc[i] == prointr)
                    printf("ALREADY assigned to profile\n");
                else if (pi_fnc[i] == lpintr)
                    printf("ALREADY assigned to lp\n");
                else if (pi_fnc[i] == cvint)
                    printf("ALREADY assigned to corvus\n");
                else
                    printf("Assigned to unknown handler at 0x%x\n",pi_fnc[i]);
                break;
            }
            pi_fnc[i] = fnc;
            return 0;
        }
    return 1;
}

/* Free the interrupt handler slot for a given controller.
 */
freeppin(addr)
struct device_d *addr;
{
    register int i;

    for (i=0; i<NPPDEVS; i++)
        if (pro_da[i] == addr) {
            pi_fnc[i] = 0;
            return;
        }
}

/*
 * ppintr - handle interrupt from parallel port controllers
 */
ppintr(ap)
struct args *ap;
{
    register int i, j;
    register char a;
    register struct device_d *dp;
    int (*fnc)(), ebintr(), prointr(), cvint(), lpintr();
    extern char lpfllg[];

    if((i = ap->a_dev) == 0) { /* special case for pp 0 */
        if(fnc == pi_fnc[i]) {
            fnc(i);
            return;
        }
    }
    j = i + 2;
    while (i < j) {
        dp = pro_da[i];
        if ((a = dp->d_ifr) & FCAL) {
            asm("nop");
            dp->d_ifr = a; /* reset interrupt */
            if (fnc == pi_fnc[i]) {
                if (fnc == lpintr)
                    lpfllg[i] = 0;
                else if (fnc != ebintr &&
                    fnc != prointr &&
                    fnc != cvint) {
                    printf("pi_fnc[%d] = 0x%x invalid!!\n",
                        i, fnc);
                    return;
                }
            }
        }
        i++;
    }
}

```



```

        fnc(i);
        return;
    }
#endif INTDUMP
    ppdump(i,dp);
#endif INTDUMP
    return;
}
    i++;
}
}

#endif INTDUMP
ppdump(n, p)
register struct device_d *p;
{
    printf("pport %d: ",n);
    printf("ifr=%x acr=%x pcr=%x ddra=%x ddrb=%x irb=%x\n",
        p->d_ifr&0xFF, p->d_acr&0xFF, p->d_pcr&0xFF, p->d_ddra&0xFF,
        p->d_ddrb&0xFF, p->d_irb&0xFF);
}
#endif INTDUMP

/*
 * called from clock if there's a panic in progress
 */
clkstop()
{
    VROFF = 1; /* disable vertical retrace intr */
}

nmikey()
{
    int i;
    register short status;

    /* added 7/25/84 to provide more info than "NMI key".
     * (taken from section 2.8 of Lisa Theory of Operations)
     */
    printf("non-maskable interrupt: ");
    status = STATUS;
    if (status & S_SMEMERR)
        printf("soft memory error\n");
    else if (status & S_HMEMERR)
        printf("hard memory error\n");
    else
        printf("power failure/keyboard reset\n");
}

#endif HOWFAR
showbus();
#endif HOWFAR
for (i=0xc00000; i>0; i--); /* delay */
}

#endif SUNIX
/* Get swap device name
 */
getdevnam ()
{
    char *p, *gets();
    int unit, dev;

retry:
    printf("\n\nWhere is the swap area?\n");
    printf("Enter: 'p' for builtin disk or a profile disk\n");
    printf("        'c' for Corvus disk\n");
    printf("        'pm' for Priam disk\n");
    p = gets();
    switch (p[0]) {
    case 'p':
        dev = PR0;
        if (p[1] == 'm')
            dev = PM3;
        break;
    case 'c':
        dev = CV2;
        break;
    default:
        printf("Invalid input. Try again.\n");
        goto retry;
    }
}

printf("Where will the disk be?\n");
if ((dev == PR0) || (dev == CV2)) {
    printf("Enter: '0' for builtin port\n");
    printf("        '1' for Expansion Slot 1, Bottom Port\n");
    printf("        '2' for Expansion Slot 1, Top Port\n");
    printf("        '4' for Expansion Slot 2, Bottom Port\n");
    printf("        '5' for Expansion Slot 2, Top Port\n");
    printf("        '7' for Expansion Slot 3, Bottom Port\n");
    printf("        '8' for Expansion Slot 3, Top Port\n");
    p = gets();
    switch (p[0]) {
    case '0':
    case '1':
    case '2':
    case '4':
    case '5':
    case '7':
    case '8':
        unit = p[0] - '0';
        break;
    default:
        printf("Invalid input. Try again.\n");
        goto retry;
    }
} else { /* dev == PM3 */
    printf("Enter: '0' for Slot 1\n");
    printf("        '1' for Slot 2\n");
    printf("        '2' for Slot 3\n");
    p = gets();
    switch (p[0]) {
    case '0':
    case '1':
    case '2':
        unit = p[0] - '0';
        break;
    default:
        printf("Invalid input. Try again.\n");
        goto retry;
    }
}
return makedev(dev, (unit<<4) | 1);
}

chkdev(d)
{
    return(*bdevsw[bmajor(d)].d_open)(minor(d), FREAD | FWRITE);
}

/*
 * This version of getchar reads directly from the keyboard in order to get
 * swapdev when the parallel port is not available. It will not work once
 * the console has been formally opened.
 */
char kb_getchr;
cogetchar()
{
    SPL0();
    while(kb_state); /* wait for kb driver to finish special cmd */
    kb_getchr = 1; /* wait flag */
    while (kb_getchr); /* wait for it to happen */
    return kb_chrbuf;
}

/* Kernel get string routine.
 * Useful for getting information from the console before the system
 * comes up. The getchar routine will not work once the console has
 * been opened.
 */
int (*getchar)() = cogetchar;
extern int (*putchar)();

char getsbuf[100];

```

```
char *
gets ()
{
    register char *p;
    register char c;
    extern short kb_keycount;

    p = getsbuf;
    while (c = (*getchar)()) {
        switch (c) {
            case '\r':
            case '\n':
                goto out;
            case '\b':
                if (p > getsbuf) {
                    p--;
                }
                break;
            case '\0':
            case '\x01F': /* line kill */
                if (p > getsbuf) {
                    p = getsbuf;
                    c = '\n'; /* echo a newline */
                }
                break;
            default:
                *p++ = c;
        }
        (*putchar)(c);
        if (p >= getsbuf + sizeof(getsbuf)) {
            printf("\nInput line too long, try again ...\n");
            p = getsbuf;
        }
    }
out:
    *p = '\0';
    (*putchar)('\n');
    return getsbuf;
}
#endif SUNIX
```

```

/*
 * (C) 1984 UniSoft Corp. of Berkeley CA
 *
 * UniPlus Source Code. This program is proprietary
 * with Unisoft Corporation and is not to be reproduced
 * or used in any manner except as authorized in
 * writing by Unisoft.
 *
 * Interrupt handler for level 2 interrupts.
 * keyboard, mouse, real time clock, on/off switch
 */

#include "sys/param.h"
#include "sys/types.h"
#include "sys/mmu.h"
#include "sys/reg.h"
#include "sys/local.h"
#include "sys/cops.h"
#include "sys/keyboard.h"
#include "sys/mouse.h"
#include "sys/ms.h"
#include "sys/l2.h"
#include "sys/kb.h"

extern struct rtime rtime;

char *kb_keytab = ToLA;
char kb_altkp; /* are we in alternate keypad mode? (set in vt100.c) */

char pportplug;
char ms_plg, ms_btn;
short ms_row, ms_col;

kbintr()
{
    register char i;
    register struct device_e *p = COPSADDR;
    register char a, ud;
    register int tmp;
    extern time_t lbolt;
#ifdef SUNIX
    extern char kb_getchr; /* flag for polling keyboard */
#endif
    a = p->e_ifr; /* Read and save reason for interrupt */
    if (a & FCA1) { /* keyboard input */
        i = p->e_ira; /* get keyboard/mouse input */
        if (a & FTIMER1)
            a = COPSADDR->e_tlcl; /* prime timer */
    } else { /* no character input */
        a = COPSADDR->e_tlcl; /* prime timer */
        if (--kb_reptrap == 0)
            kbrepeat(); /* possible char repeat */
        return;
    }

    switch (kb_state) {
    case NORMALWAIT: /* IDLE LOOP */
        ud = i & 0x80; /* whether up or down keycode */
        l2_dtrap = lbolt + l2_dtime; /* reset dim delay */
        if (l2_dimmed) /* restore screen intensity */
            l2undim();
        a = kb_keytab[i & 0x7F]; /* convert to ascii */
        if (ud) { /* "key went down" bit */
            /* click(); /* key click */
            if (ARROW(i,a)) { /* check arrow keys */
                kb_chrbuf = Esc; /* send 3-char sequence */
                cointr(0);
                kb_chrbuf = '{';
                cointr(0);
                kb_chrbuf = a;
                cointr(0);
                goto out;
            }
            if (kb_altkp) { /* send special sequence for keypad chars */
                if (a = altkpad[i & 0x7F]) { /* is it in the keypad? */
                    kb_chrbuf = Esc; /* send 3-char sequence */
                    cointr(0);
                    kb_chrbuf = 'O';
                    cointr(0);
                    kb_chrbuf = a;
                    cointr(0);
                    goto out;
                }
                a = kb_keytab[i & 0x7F]; /* reset to ascii */
            }
            if (a >= 0) { /* ascii ? */
                kb_keycount++;
                if (kb_ctrl) a &= 0x1F;
                else if (kb_keycount == 1) {
                    kb_reptrap = kb_reptrap;
                    kb_lastc = a;
                } else kb_reptrap = 0;
                kb_chrbuf = a;
                cointr(0);
                goto out;
            }
        } else { /* key went up */
            kb_reptrap = 0;
            if (a >= 0) {
                if (kb_keycount-- < 0) {
                    kb_keycount = 0;
                }
                goto out;
            }
        }

        switch (a & 0xF) {
        case KB_CTRL: kb_ctrl = ud; kb_reptrap = 0; msintr(M_CTL); goto out;
        case KB_SHFT: kb_shft = ud; kbsetcvtab(); msintr(M_SFT); goto out;
        case KB_LOCK: kb_lock = ud; kbsetcvtab(); goto out;
        case KB_OFF: printf("[SOFT OFF %x]\n",i); goto out;
        case KB_MSP: ms_plg = ud; msintr(M_PLUG); goto out;
        case KB_MSB: ms_btn = ud; msintr(M_BUT); goto out;
        case KB_PPORT: pportplug = ud; goto out;
        case KB_D2B: printf("[disk1button %x]\n",i); goto out;
        case KB_D2P: printf("[disk1 %x]\n",i); goto out;
        case KB_D1B: printf("[disk2button %x]\n",i); goto out;
        case KB_D1P: printf("[disk2 %x]\n",i); goto out;
        case KB_STATE: if (ud == 0) {
            kb_state = MOUSERD;
        } else
            kb_state = RESETCODE;
            goto out;
        default: printf("invalid key[0x%x]\n",i);
        }
        goto out;
    case MOUSERD: /* PICKUP Y axis change in mouse pos */
        kb_state = YMOUSE;
        ms_col = (short)i;
        goto out;
    case YMOUSE: /* PICKUP Y axis change in mouse pos */
        kb_state = NORMALWAIT;
        ms_row = (short)i;
        msintr(M_MOVE);
        goto out;
    case RESETCODE: /* special condition */
        switch (i & 0xFF) {
        case KB_KBCOPS: /* keyboard cops failure detected */
            printf("KEYBOARD COPS FAILURE\n");
            break;
        case KB_IOCOPS: /* IO board cops failure detected */
            printf("IO BOARD COPS FAILURE\n");
            break;
        case KB_UNPLUG: /* keyboard unplugged */
            kb_chrbuf = 's' & 0x1F; /* cntl S */
            cointr(0);
            break;
        case KB_CLOCKT: /* clock timer interrupt */
            printf("Real Time Clock interrupt\n");
            break;
        }
    }
}

```

```

case KB_SFTOFF: /* soft power switch */
    kb_state = SHUTDOWN;
    printf("Shutting down...\n");
    update();
    for (tmp = 0; tmp < 500000; tmp++);
    l2_crate = 2;
    l2_desired = TOTALDIM; /* completely blacken screen */
    l2ramp(0);
    l2ramp(0);
    SPL7(); /* extreme priority */
    l2copscmd(SHUTOFF);
    rom_mon(); /* return to the ROM monitor */
    /*NOTREACHED*/
default: switch (i & 0xF0) {
case KB_RESERVED:
    printf("[Reserved keycode 0x%x]\n",i);
    break;
case KB_RDCLK:
    rtime.rt_year = (i & 0xF) + 10;
    kb_state = CLKREAD;
    goto out;
default:
    kb_idcode = i;
    kb_chrbuf = 'q' & 0x1F; /* cntl Q */
    cointr(0);
    printf("Keyboard type 0x%x\n",i);
}
}
kb_state = NORMALWAIT;
goto out;
case CLKREAD:
    rtime.rt_day = ((i & 0xF0) >> 4) * 10 + (i & 0xF) * 10;
    kb_state++;
    goto out;
case CLKREAD+1:
    rtime.rt_day += (i & 0xF0) >> 4;
    rtime.rt_hour = (i & 0xF) * 10;
    kb_state++;
    goto out;
case CLKREAD+2:
    rtime.rt_hour += (i & 0xF0) >> 4;
    rtime.rt_min = (i & 0x0F) * 10;
    kb_state++;
    goto out;
case CLKREAD+3:
    rtime.rt_min += (i & 0xF0) >> 4;
    rtime.rt_sec = (i & 0x0F) * 10;
    kb_state++;
    goto out;
case CLKREAD+4:
    rtime.rt_sec += (i & 0xF0) >> 4;
    rtime.rt_tenth = i & 0x0F;
    kb_state = NORMALWAIT;
    rtcsettod();
    goto out;
case SHUTDOWN:
    goto out;
}
out: ;
#ifdef SUNIX
    if (!ud)
        kb_getchr = 0;
#endif
}

kbrepeat ()
{
    kb_reptrap = kb_repdlay; /* reset repeat timeout */

    if (kb_keycount == 1) {
        kb_chrbuf = kb_lastc;
/* click(); /* key click */
        cointr(0);
    } else
        kb_reptrap = 0; /* reset repeat timeout */
}

```

```

kbsetcvtab()
{
    kb_keytab = ccvtab[(kb_shft?2:0)+(kb_lock?1:0)];
    kb_reptrap = 0;
}

```

```

/* (#)slp.c 1.8 */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/proc.h"
#include "sys/context.h"
#include "sys/text.h"
#include "sys/system.h"
#include "sys/sysinfo.h"
#include "sys/map.h"
#include "sys/file.h"
#include "sys/inode.h"
#include "sys/buf.h"
#include "sys/var.h"
#include "sys/ipc.h"
#include "sys/shm.h"
#include "sys/errno.h"
#include "sys/scat.h"

typedef int mem_t;

#define NHSQUE 64 /* must be power of 2 */
#define sqhash(X) ((int)(X) >> 3) & (NHSQUE-1))
struct proc *hsque[NHSQUE];
char runin, runout, runrun, curpri;
struct proc *curproc, *runq;

/*
 * sleep according to a cpu adjusted priority
 */
asleep(chan, pri)
caddr_t chan;
{
    return(sleep(chan, pri + ((u.u_procp->p_cpu&0xFF) >> 5)));
}

/*
 * Give up the processor till a wakeup occurs
 * on chan, at which time the process
 * enters the scheduling queue at priority pri.
 * The most important effect of pri is that when
 * pri<=PZERO a signal cannot disturb the sleep;
 * if pri>PZERO signals will be processed.
 * Callers of this routine must be prepared for
 * premature return, and check that the reason for
 * sleeping has gone away.
 */
#define TZERO 10
sleep(chan, disp)
caddr_t chan;
{
    register struct proc *rp = u.u_procp;
    register struct proc **q = sqhash(chan);
    register s;

    s = splhi();
    if (panicstr) {
        SPL0();
        splx(s);
        return(0);
    }
    rp->p_stat = SSLEEP;
    rp->p_wchan = chan;
    rp->p_link = *q;
    *q = rp;
    if (rp->p_time > TZERO)
        rp->p_time = TZERO;
    if (((rp->p_pri = (disp&PMASK)) > PZERO) &&
        if (rp->p_sig && issig()) {
            rp->p_wchan = 0;
            rp->p_stat = SRUN;
            *q = rp->p_link;
            SPL0();
            goto psig;
        }
        SPL0();
        if (runin != 0) {
            runin = 0;
            wakeup((caddr_t)&runin);
        }
        swtch();
        if (rp->p_sig && issig())
            goto psig;
    } else {
        SPL0();
        swtch();
    }
    splx(s);
    return(0);
}

/*
 * If priority was low (>PZERO) and there has been a signal,
 * if PCATCH is set, return 1, else
 * execute non-local goto to the qsav location.
 */
psig:
    splx(s);
    if (disp&PCATCH)
        return(1);
#ifdef NONSCATLOAD
    resume(u.u_procp->p_addr, u.u_qsav);
#else
    resume(ixtoc(u.u_procp->p_scat), u.u_qsav);
#endif
    /* NOTREACHED */
}

/*
 * Wake up all processes sleeping on chan.
 */
wakeup(chan)
register caddr_t chan;
{
    register struct proc *p;
    register struct proc **q;
    register s;

    s = splhi();
    for (q = sqhash(chan); p = *q; )
        if (p->p_wchan==chan && p->p_stat==SSLEEP) {
            p->p_stat = SRUN;
            p->p_wchan = 0;
            /* take off sleep queue, put on run queue */
            *q = p->p_link;
            p->p_link = runq;
            runq = p;
            if (!(p->p_flag&SLOAD)) {
                p->p_time = 0;
                /* defer setrun to avoid breaking link chain */
                if (runout > 0)
                    runout = -runout;
            } else if (p->p_pri < curpri)
                runrun++;
        } else
            q = &p->p_link;
    if (runout < 0) {
        runout = 0;
        setrun(&proc[0]);
    }
    splx(s);
}

setrq(p)
register struct proc *p;
{
    register struct proc *q;
    register s;

```

```

s = splhi();
for(q=runq; q!=NULL; q=q->p_link)
    if (q == p) {
        printf("proc on q\n");
        goto out;
    }
p->p_link = runq;
runq = p;
out:
    splx(s);
}

/*
 * Set the process running;
 * arrange for it to be swapped in if necessary.
 */
setrun(p)
register struct proc *p;
{
    register struct proc **q;
    register s;

    s = splhi();
    if (p->p_stat == SSLEEP) {
        /* take off sleep queue */
        for (q = sqhash(p->p_wchan); *q != p; q = &(*q)->p_link)
            *q = p->p_link;
        p->p_wchan = 0;
    } else if (p->p_stat == SRUN) {
        /* already on run queue - just return */
        splx(s);
        return;
    }
    /* put on run queue */
    p->p_stat = SRUN;
    p->p_link = runq;
    runq = p;
    if (!(p->p_flag&SLOAD)) {
        p->p_time = 0;
        if (runout > 0) {
            runout = 0;
            setrun(&proc[0]);
        }
    } else if (p->p_pri < curpri)
        runrun++;
    splx(s);
}

/*
 * The main loop of the scheduling (swapping) process.
 * The basic idea is:
 * see if anyone wants to be swapped in;
 * swap out processes until there is room;
 * swap him in;
 * repeat.
 * The runout flag is set whenever someone is swapped out.
 * Sched sleeps on it awaiting work.
 *
 * Sched sleeps on runin whenever it cannot find enough
 * memory (by swapping out or otherwise) to fit the
 * selected swapped process. It is awakened when the
 * memory situation changes and in any case once per second.
 */
sched()
{
    register struct proc *rp, *p;
    register outage, inage;
    int maxbad;
    int tmp;

    /*
     * find user to swap in;
     * of users ready, select one out longest
     */

```

```

loop:
    SPL6();
    outage = -20000;
#ifdef NONSCATLOAD
    for (rp = &proc[0]; rp < (struct proc *)v.ve_proc; rp++)
        if (rp->p_stat==SRUN && (rp->p_flag&SLOAD) == 0 &&
            rp->p_time > outage) {
            p = rp;
            outage = rp->p_time;
        }
#else
    for (rp = &proc[0]; rp < (struct proc *)v.ve_proc; rp++)
        if ((rp->p_flag&(SSWAPIT|SLOAD)) == (SSWAPIT|SLOAD)) {
            p = rp;
            SPL0();
            goto swapit;
        } else if (rp->p_stat==SRUN && (rp->p_flag&SLOAD) == 0 &&
            rp->p_time > outage) {
            p = rp;
            outage = rp->p_time;
        }
#endif
    /*
     * If there is no one there, wait.
     */
    if (outage == -20000) {
        runout++;
        (void) sleep((caddr_t)&runout, PSWP);
        goto loop;
    }
    SPL0();

    /*
     * See if there is memory for that process;
     * if so, swap it in.
     */
    if (swapi(p))
        goto loop;

    /*
     * none found.
     * look around for memory.
     * Select the largest of those sleeping
     * at bad priority; if none, select the oldest.
     */
    SPL6();
    p = NULL;
    maxbad = 0;
    inage = 0;
    for (rp = &proc[0]; rp < (struct proc *)v.ve_proc; rp++) {
        if (rp->p_stat==SZOMB ||
            (rp->p_flag&(SSYS|SLOCK|SLOAD))!=SLOAD)
            continue;
        if (rp->p_textp && rp->p_textp->x_flag&XLOCK)
            continue;
        if (rp->p_stat==SSLEEP || rp->p_stat==SSTOP) {
            tmp = rp->p_pri - PZERO + rp->p_time;
            if (maxbad < tmp) {
                p = rp;
                maxbad = tmp;
            }
        } else if (maxbad<=0 && rp->p_stat==SRUN) {
            tmp = rp->p_time + rp->p_nice - NZERO;
            if (tmp > inage) {
                p = rp;
                inage = tmp;
            }
        }
    }
    SPL0();
    /*
     * Swap found user out if sleeping at bad pri,
     * or if he has spent at least 2 seconds in memory and
     * the swapped-out process has spent at least 2 seconds out.
     */

```

```

    * Otherwise wait a bit and try again.
    */
    if (maxbad>0 || (outage>=2 && inage>=2)) {
#ifdef NONSCATLOAD
        swapit:
#endif
        p->p_flag &= ~SLOAD;
        xswap(p, 1, 0);
        goto loop;
    }
    SPL6();
    runin++;
    (void) sleep((caddr_t)&runin, PSWP);
    goto loop;
}

/*
 * Swap a process in.
 */
#ifdef NONSCATLOAD
swapin(p)
register struct proc *p;
{
    register struct text *xp;
    register int a, x;
    int ta;

    if ((a = malloc(coremap, p->p_size)) == NULL)
        return(0);
    if (xp = p->p_textp) {
        xlock(xp);
        if (!xmlink(xp) && xp->x_ccount==0) {
            if ((x = malloc(coremap, xp->x_size)) == NULL) {
                mfree(coremap, p->p_size, a);
                if ((x = malloc(coremap, xp->x_size)) == NULL) {
                    xunlock(xp);
                    return(0);
                }
            }
            if ((a = malloc(coremap, p->p_size)) == NULL) {
                mfree(coremap, xp->x_size, x);
                xunlock(xp);
                return(0);
            }
        }
        xp->x_caddr = x;
        if ((xp->x_flag&XLOAD)==0)
            swap(xp->x_daddr, x, xp->x_size, B_READ);
    }
    xp->x_ccount++;
    xunlock(xp);
}
if (p->p_xaddr[0]) {
    ta = a;
    for (x=0; x < NSCATSWAP; x++) {
        if (p->p_xaddr[x] == 0)
            continue;
        swap(p->p_xaddr[x], a, p->p_xsize[x], B_READ);
        mfree(swampmap, ctod(p->p_xsize[x]), (int)p->p_xaddr[x]);
        a += p->p_xsize[x];
        p->p_xaddr[x] = 0;
    }
    p->p_addr = ta;
} else {
    swap((daddr_t)p->p_dkaddr, a, p->p_size, B_READ);
    mfree(swampmap, ctod(p->p_size), (int)p->p_dkaddr);
    p->p_addr = a;
}
cxrelse(p->p_context);
p->p_flag |= SLOAD;
p->p_time = 0;
return(1);
}
#else
swapin(p)
register struct proc *p;
{

```

```

    register struct text *xp;
    register int a, x;
    int ta;

    if (p->p_flag&SCONTIG) {
        if ((a = cmemalloc(p->p_size)) == NULL)
            return(0);
    } else if ((a = memalloc(p->p_size)) == NULL)
        return(0);
    if (xp = p->p_textp) {
        xlock(xp);
        if (!xmlink(xp) && xp->x_ccount==0) {
            if ((x = memalloc(xp->x_size)) == NULL) {
                memfree(a);
                xunlock(xp);
                return(0);
            }
        }
        xp->x_scatter = x;
        if ((xp->x_flag&XLOAD)==0)
            (void) swap(xp->x_daddr, x, xp->x_size, B_READ);
    }
    xp->x_ccount++;
    xunlock(xp);
}
p->p_flag |= SNOMMU; /* swapping in, do not set mmu registers */
if (p->p_xaddr[0]) {
    ta = a;
    for (x=0; x < NSCATSWAP; x++) {
        if (p->p_xaddr[x] == 0)
            continue;
        a = swap(p->p_xaddr[x], a, p->p_xsize[x], B_READ);
        mfree(swampmap, ctod(p->p_xsize[x]), (int)p->p_xaddr[x]);
        p->p_xaddr[x] = 0;
    }
    p->p_scatter = ta;
} else {
    (void) swap((daddr_t)p->p_dkaddr, a, p->p_size, B_READ);
    mfree(swampmap, ctod(p->p_size), (int)p->p_dkaddr);
    p->p_scatter = a;
}
p->p_flag &= ~SNOMMU;
p->p_addr = itoc(p->p_scatter);
cxrelse(p->p_context);
p->p_flag |= SLOAD;
p->p_time = 0;
return(1);
}
#endif

/*
 * put the current process on
 * the Q of running processes and
 * call the scheduler.
 */
qswtch()
{
    setrq(u.u_proc);
    swtch();
}

/*
 * This routine is called to reschedule the CPU.
 * if the calling process is not in RUN state,
 * arrangements for it to restart must have
 * been made elsewhere, usually by calling via sleep.
 * There is a race here. A process may become
 * ready after it has been examined.
 * In this case, idle() will be called and
 * will return in at most 1HZ time.
 * i.e. its not worth putting an spl() in.
 */
swtch()
{
    register n;
    register struct proc *p, *q, *pp, *pq;
#ifdef mc68881 /* MC68881 floating-point coprocessor */

```

```

extern short fp881;          /* is there an MC68881? */
#endif mc68881

/*
 * If not the idle process, resume the idle process.
 */
sysinfo.pswitch++;
if (u.u_procp != &proc[0]) {
    if (save(u.u_rsav)) {
        sureq();
        return;
    }
}
#ifndef FLOAT
    /* sky floating point board */
    if (u.u_fpinuse && u.u_fpsaved==0) {
        savfp();
        u.u_fpsaved = 1;
    }
#endif
#endif
#ifdef mc68881
    /* MC68881 floating-point coprocessor */
    if (fp881)
        fpsave();
#endif mc68881
#ifdef NONSCATLOAD
    resume(proc[0].p_addr, u.u_qsav);
#else
    resume(ixtoc(proc[0].p_scatter), u.u_qsav);
#endif
}
/*
 * The first save returns nonzero when proc 0 is resumed
 * by another process (above); then the second is not done
 * and the process-search loop is entered.
 *
 * The first save returns 0 when switch is called in proc 0
 * from sched(). The second save returns 0 immediately, so
 * in this case too the process-search loop is entered.
 * Thus when proc 0 is awakened by being made runnable, it will
 * find itself and resume itself at rsav, and return to sched().
 */
if (save(u.u_qsav) == 0 && save(u.u_rsav))
    return;
loop:
    SPL6();
    runrun = 0;
    /*
     * Search for highest-priority runnable process
     */
    if (p = runq) {
        q = NULL;
        pp = NULL;
        n = 128;
        do {
            if ((p->p_flag&SLOAD) && p->p_pri <= n) {
                pp = p;
                pq = q;
                n = p->p_pri;
            }
            q = p;
        } while (p = p->p_link);
    } else
        goto cont;
    /*
     * If no process is runnable, idle.
     */
    if (pp == NULL) {
        curpri = PIDLE;
        curproc = &proc[0];
        idle();
        goto loop;
    }
    p = pp;
    q = pq;
    if (q == NULL)
        runq = p->p_link;
    else
        q->p_link = p->p_link;

```

```

curpri = n;
curproc = p;
SPL0();
/*
 * The rsav (ssav) contents are interpreted in the new address space
 */
n = p->p_flag&SSWAP;
p->p_flag &= ~SSWAP;
#endif NONSCATLOAD
    resume(p->p_addr, n? u.u_ssav: u.u_rsav);
#else
    resume(ixtoc(p->p_scatter), n? u.u_ssav: u.u_rsav);
#endif
}

int mpid;          /* external for sunix so it can be reinitialized */
/*
 * Create a new process-- the internal version of
 * sys fork.
 * It returns 1 in the new process, 0 in the old.
 */
newproc(i)
{
    register struct proc *rpp, *rip;
    register struct user *up;
    register n;
    register a;
    struct proc *pend;
    /* static mpid; */

    /*
     * First, just locate a slot for a process
     * and copy the useful info from this process into it.
     * The panic "cannot happen" because fork has already
     * checked for the existence of a slot.
     */
    up = &u;
    rpp = NULL;
retry:
    mpid++;
    if (mpid >= MAXPID) {
        mpid = 0;
        goto retry;
    }
    rip = &proc[0];
    n = (struct proc *)v.ve_proc - rip;
    a = 0;
    do {
        if (rip->p_stat == NULL) {
            if (rpp == NULL)
                rpp = rip;
            continue;
        }
        if (rip->p_pid==mpid)
            goto retry;
        if (rip->p_uid == up->u_ruid)
            a++;
        pend = rip;
    } while(rip++, --n);
    if (rpp==NULL) {
        if ((struct proc *)v.ve_proc >= &proc[(short)v.v_proc]) {
            if (i) {
                syserr.procovf++;
                up->u_error = EAGAIN;
                return(-1);
            } else
                panic("no procs");
        }
        rpp = (struct proc *)v.ve_proc;
    }
    if (rpp > pend)
        pend = rpp;
    pend++;
#ifdef lint
    v.ve_proc = pend;
#else

```



```

v.ve_proc = (char *)pend;
#endif
if (up->u_uid && up->u_ruid) {
    if (rpp == &proc[(short)(v.v_proc-1)] || a > v.v_maxup) {
        up->u_error = EAGAIN;
        return(-1);
    }
}
/*
 * make proc entry for new proc
 */

rip = up->u_procp;
rpp->p_stat = SRUN;
rpp->p_clktim = 0;
rpp->p_flag = SLOAD;
rpp->p_uid = rip->p_uid;
rpp->p_suid = rip->p_suid;
rpp->p_pgrp = rip->p_pgrp;
rpp->p_nice = rip->p_nice;
rpp->p_textp = rip->p_textp;
rpp->p_pid = mpid;
rpp->p_ppid = rip->p_pid;
rpp->p_time = 0;
rpp->p_cpu = rip->p_cpu;
rpp->p_sigign = rip->p_sigign;
rpp->p_pri = PUSER + rip->p_nice - NZERO;
#endif
NONSCATLOAD
rpp->p_scat = rip->p_scat;
#endif
rpp->p_addr = rip->p_addr;
rpp->p_size = rip->p_size;

/*
 * make duplicate entries
 * where needed
 */

for(n=0; n<NOFILE; n++)
    if (up->u_ofile[n] != NULL)
        up->u_ofile[n]->f_count++;
if (rpp->p_textp != NULL) {
    rpp->p_textp->x_count++;
    rpp->p_textp->x_ccount++;
}
up->u_cdir->i_count++;
if (up->u_rdir)
    up->u_rdir->i_count++;

shmfork(rpp, rip);

/*
 * Partially simulate the environment
 * of the new process so that when it is actually
 * created (by copying) it will look right.
 */
up->u_procp = rpp;
curproc = rpp;
/*
 * When the resume is executed for the new process,
 * here's where it will resume.
 */
if (save(up->u_ssav)) {
    sureg();
    return(1);
}
/*
 * If there is not enough memory for the
 * new process, swap out the current process to generate the
 * copy.
 */
if (procdup(rpp) == NULL) {
    rip->p_stat = SIDL;
    xswap(rpp, 0, 0);
    rip->p_stat = SRUN;
}

up->u_procp = rip;
curproc = rip;
if (rip != &proc[0]) /* only do if not scheduler */
    sureg();
setrg(rpp);
rpp->p_flag |= SSWAP;
up->u_rvall = rpp->p_pid; /* parent returns pid of child */
return(0);
}

/*
 * Change the size of the data+stack regions of the process.
 * If the size is shrinking, it's easy-- just release the extra core.
 * If it's growing, and there is core, just allocate it
 * and copy the image, taking care to reset registers to account
 * for the fact that the system's stack has moved.
 * If there is no memory, arrange for the process to be swapped
 * out after adjusting the size requirement-- when it comes
 * in, enough memory will be allocated.
 */
/*
 * After the expansion, the caller will take care of copying
 * the user's stack towards or away from the data area.
 */
#ifdef NONSCATLOAD
expand(newsize)
register newsize;
{
    register struct proc *p;
    register a1, a2;
    register i, n;

    p = u.u_procp;
    n = p->p_size;
    p->p_size = newsize;
    if (n == newsize)
        return;
    a1 = p->p_addr;
    if (n >= newsize) {
#ifdef EXPANDTRACE
        printf("expand:shrinking process by %d clicks\n", n-newsize);
#endif
        mfree(coremap, (mem_t)(n-newsize), (mem_t)(a1+newsize));
        return;
    }
    if (save(u.u_ssav)) {
        sureg();
        return;
    }
    /*
     * See if can just expand in place
     */
loop:
    a2 = malloc(coremap, newsize-n, (mem_t)(a1+n));
    if (a2 != NULL) {
#ifdef EXPANDTRACE
        printf("expanding in place by %d clicks at click %d\n",
            newsize-n, a1+n);
#endif
        cxrrelse(p->p_context);
        sureg();
        return;
    }
    /*
     * Will we be releasing shared text space anyway.
     * If so, then release it now and try in place
     * expansion again.
     */
    if ((a2 = domall(coremap, (mem_t)newsize)) == NULL)
        if (xmrelse())
            goto loop;
    if (a2 == NULL && (a2 = malloc(coremap, (mem_t)newsize)) == NULL) {
#ifdef EXPANDTRACE
        printf("expand:calling xswap\n");
#endif
        xswap(p, 1, n);
        p->p_flag |= SSWAP;
    }
}

```

```

        qswtch();
        /* no return */
    }
    p->p_addr = a2;
    for(i=0; i<n; i++)
        copyseg(a1+i, a2+i);
#ifdef EXPANDTRACE
    printf("expand:copyseg %d from 0x%x to 0x%x\n", n, a1, a2);
#endif
    mfree(coremap, (mem_t)n, (mem_t)a1);
    cxrelse(p->p_context);
    resume((mem_t)a2, u.u_ssav);
}
#else
expand(newsize)
register newsize;
{
    register struct scatter *s;
    register struct proc *p;
    register al, a2;
    register i, n;
    int t;

    p = u.u_procp;
    n = p->p_size;
    p->p_size = newsize;
    if (n == newsize)
        return;
    s = scatmap;
    al = p->p_scat;
    if (n >= newsize) {
        /*
         * shrink memory
         */
        for (i=1; i<newsize; i++)
            al = s[al].sc_index;
        t = scatfreelist.sc_index;
        scatfreelist.sc_index = s[al].sc_index;
        i = al;
        while ((a2 = s[al].sc_index) != SCATEND)
            al = a2;
        s[i].sc_index = SCATEND;
        s[al].sc_index = t;
        nscatfree += n-newsize;
        /*
         * Wake scheduler when freeing memory
         */
        if (runin) {
            runin = 0;
            wakeup((caddr_t)&runin);
        }
        return;
    }
    if (save(u.u_ssav)) {
        sureq();
        return;
    }
    /*
     * expand memory
     */
    if (a2 = memalloc(newsize-n)) {
        /* printf("expanding from %d clicks to %d clicks\n",
         * n, newsize); */
        for (i=1; i<n; i++)
            al = s[al].sc_index;
        if (s[al].sc_index != SCATEND)
            printf("expand:SCATEND expected\n");
        s[al].sc_index = a2;
        return;
    }
    xswap(p, 1, n);
    p->p_flag |= SSWAP;
    qswtch();
    /* no return */
}
#endif

```

```

#ifdef NONSCATLOAD
checkscat(s)
char *s;
{
    register struct proc *p;
    register struct text *xp;
    register i;

    i = countscat(scatfreelist.sc_index);
    printf("%s nscatfree=%d actual=%d\n", s, nscatfree, i);
    if (nscatfree < 30) {
        printf("freelist chain is ");
        dumpscat(scatfreelist.sc_index);
    }
    for (p = &proc[0]; p < (struct proc *)v.ve_proc; p++) {
        if (p->p_stat==0)
            continue;
        xp = p->p_textp;
        printf("pid=%d %d used (%d text)\n",
            p->p_pid, countscat(p->p_scat),
            xp ? countscat(xp->x_scat) : 0);
        dumpscat(p->p_scat);
        if (xp) {
            dumpscat(xp->x_scat);
        }
    }
    dumpscat(al)
    register al;
    {
        register struct scatter *s;

        s = scatmap;
        printf(" ");
        while (al != SCATEND) {
            printf(" %d,%x", al, ixtoc(al));
            al = s[al].sc_index;
        }
        printf("\n");
    }
    countscat(al)
    register al;
    {
        register struct scatter *s;
        register i;

        i = 0;
        s = scatmap;
        while (al != SCATEND) {
            al = s[al].sc_index;
            i++;
        }
        return(i);
    }
}
#endif

```

```

/*
 * SCC device driver
 *
 * Copyright 1982 UniSoft Corporation
 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/file.h"
#include "sys/tty.h"
#include "sys/termio.h"
#include "sys/conf.h"
#include "sys/sysinfo.h"
#include "sys/var.h"
#include "sys/reg.h"
#include <sys/scc.h>
#include "sys/proc.h"
#include "setjmp.h"

int scproc();
extern int sc_cnt;
extern struct tty sc_tty[];
extern struct ttyptr sc_ttptr[];
extern char sc_modem[];
extern struct scline sc_line[];

#define MODEM 0x80 /* modem control on bit */
#define scdev(d) ((d)&0x7f) /* from unix device number to device */

#define DELAY() asm("\tnop");

#define SCTIME (v.v_hz*5) /* scscan interval */

/*
 * Note: to select baud rate
 * k = chip_input_frequency/(2 * baud * factor) - 2
 * put factor in Register 9 and k in registers D & C
 * NOTE:
 * normally, factor = 16
 * for this driver, chip_input_frequency = 2400000 Hz
 * scspeeds is a table of these numbers by UNIX baud rate
 */
int scspeeds[] = {
    1, 50, 75, 110, 134, 150, 200, 300,
    600, 1200, 1800, 2400, 4800, 9600, 19200, 38400,
};

/*
 * table to initialize a port to 9600 baud
 */
char scitable[] = {
    W0NULL,
    9, 0, /* set according to sc_line reset value */
#define SCCIRESET scitable[2]
    4, W4CLK16 | W41STOP,
    10, 0,
    11, W11RBR | W11TBR,
    12, 0, /* 12/13 are baud rate, from sc_line speed value */
#define SCCISPLO scitable[10]
    13, 0,
#define SCCISPHI scitable[12]
    14, W14BRGE | W14BRINT,
    3, W38BIT | W3RXENABLE,
    5, W58BIT | W5TXENABLE,
    1, W1RXIAL | W1TXIEN /*| W1EXTIEN*/,
    2, 0, /* auto vector */
    0, W0RXINT,
    15, 0,
    9, W9MIE | W9DLC,

```

```

);
/*
 * we call this in startup() to preinitialize all the ports
 */
scinit()
{
    register struct device *addr;
    unsigned short nsc;

    for(nsc=0;nsc<500;nsc++);
    for (nsc = 0; nsc < sc_cnt; nsc++) {
        addr = (struct device *)sc_ttptr[nsc].tt_addr;
        /* do proper reset */
        scinil(addr, W9NV|sc_line[nsc].reset, (int)sc_line[nsc].speed);
    }
    printf("%d serial ports\n", nsc);
    /* scscan(); /* start modem scanning */
}

scinil(addr, reset, speed)
struct device *addr;
int reset, speed;
{
    register int i;
    int s;

    s = spl6();
    SCCIRESET = reset;
    speed = (speed/(9600<<1)) - 2;
    SCCISPHI = (speed >> 8) & 0xFF;
    SCCISPLO = speed & 0xFF;
    for (i = 0; i < sizeof(scitable); i++) {
        DELAY();
        addr->csr = scitable[i];
    }
    DELAY();
    splx(s);
}

/* ARGSUSED */
scopen(dev, flag)
register dev;
{
    register struct tty *tp;
    register struct device *addr;
    register d;
#ifdef SINGLEUSER
    register struct proc *p;
#endif SINGLEUSER

    d = scdev(dev);
    if (d >= sc_cnt) {
        u.u_error = ENXIO;
        return;
    }
    tp = sc_ttptr[d].tt_tty;
#ifdef SINGLEUSER
    p = u.u_proc;
    if ((p->p_pid == p->p_pgrp)
        && (u.u_ttyp == NULL)
        && (tp->t_pgrp == 0)) {
        u.u_error = ENOTTY;
        return;
    }
#endif SINGLEUSER
    tp->t_index = d;
    SPL6();
    if ((tp->t_state & (ISOPEN|WOPEN)) == 0) {
        tp->t_proc = scproc;
        ttinit(tp);
        sc_modem[d] = dev & MODEM;
        addr = (struct device *)sc_ttptr[d].tt_addr;
        if (dev & MODEM && (addr->csr & RODCD) == 0)
            tp->t_state = WOPEN;
        else

```

```

        tp->t_state = WOPEN | CARR_ON;
#ifdef SCC_CONSOLE
        if (d == CONSOLE) {
            tp->t_iflag = ICRNL | ISTRIP;
            tp->t_oflag = OPOST | ONLCR | TAB3;
            tp->t_lflag = ISIG | ICANON | ECHO | ECHOK;
            tp->t_cflag = sspeed | CS8 | CREAD | HUPCL;
        }
#endif
        scparam(dev);
    }
    if (!(flag & FNDELAY))
        while ((tp->t_state & CARR_ON) == 0)
            (void) sleep((caddr_t)&tp->t_rawq, TTOPRI);
    SPL0();
    (*linesw[tp->t_line].l_open)(tp);
}

/* ARGUSED */
scclose(dev, flag)
{
    register struct tty *tp;
    register int d;

    d = scdev(dev);
    tp = sc_ttptr[d].tt_tty;
    (*linesw[tp->t_line].l_close)(tp);
    if (tp->t_cflag & HUPCL)
        schup(dev);          /* hang up */
}

scread(dev)
{
    register struct tty *tp;

    tp = sc_ttptr[scdev(dev)].tt_tty;
    (*linesw[tp->t_line].l_read)(tp);
}

scwrite(dev)
{
    register struct tty *tp;

    tp = sc_ttptr[scdev(dev)].tt_tty;
    (*linesw[tp->t_line].l_write)(tp);
}

scproc(tp, cmd)
register struct tty *tp;
{
    register struct cblock *tbuf;
    register struct device *addr;
    register dev_t dev;
    extern ttrstrt();
    int s;

    s = spl6();
    dev = tp->t_index;
    addr = (struct device *)sc_ttptr[dev].tt_addr;
    switch (cmd) {

    case T_TIME:
        scw5(tp, addr, 0);      /* clear break */
        tp->t_state &= ~TIMEOUT;
        goto start;

    case T_WFLUSH:
        tbuf = &tp->t_tbuf;
        tbuf->c_size == tbuf->c_count;
        tbuf->c_count = 0;
        /* fall through */

    case T_RESUME:
        tp->t_state &= ~TTSTOP;
        goto start;

    case T_OUTPUT:

```

```

start:
    if (tp->t_state & (TTSTOP|TIMEOUT|BUSY))
        break;
    if (tp->t_state & TTXOFF) {
        tp->t_state &= ~TTXOFF;
        tp->t_state |= BUSY;
        addr->data = CSTOP;
        break;
    }
    if (tp->t_state & TTXON) {
        tp->t_state &= ~TTXON;
        tp->t_state |= BUSY;
        addr->data = CSTART;
        break;
    }
    tbuf = &tp->t_tbuf;
    if ((tbuf->c_ptr == 0) || (tbuf->c_count == 0)) {
        if (tbuf->c_ptr)
            tbuf->c_ptr -= tbuf->c_size;
        if (!(CPRES & (*linesw[tp->t_line].l_output)(tp)))
            break;
    }
    tp->t_state |= BUSY;
    addr->data = *tbuf->c_ptr++;
    tbuf->c_count--;
    break;

case T_SUSPEND:
    tp->t_state |= TTSTOP;
    break;

case T_BLOCK:
    tp->t_state &= ~TTXON;
    tp->t_state |= TBLOCK;
    tp->t_state |= TTXOFF;
    goto start;

case T_RFLUSH:
    if (!(tp->t_state & TBLOCK))
        break;
    /* fall through */

case T_UNBLOCK:
    tp->t_state &= ~(TTXOFF|TBLOCK);
    tp->t_state |= TTXON;
    goto start;

case T_BREAK:
    scw5(tp, addr, W5BREAK);
    tp->t_state |= TIMEOUT;
    timeout(ttrstrt, (caddr_t)tp, v.v_hz>>2);
    break;
}
splx(s);

scw5(tp, addr, d)
struct tty *tp;
struct device *addr;
int d;
{
    register int w5;
    int s;

    w5 = W5TXENABLE | d;
    switch(tp->t_cflag & CSIZE) {
        case CS5:
            w5 |= W55BIT; break;
        case CS6:
            w5 |= W56BIT; break;
        case CS7:
            w5 |= W57BIT; break;
        case CS8:
            w5 |= W58BIT; break;
    }
    s = spl5();

```

```

addr->csr = 5;
addr->csr = w5;
splx(s);
}

sciocctl(dev, cmd, arg, mode)
{
    if (ttiocom(sc_ttptr[scdev(dev)].tt_tty, cmd, arg, mode))
        scparam(dev);
}

scparam(dev)
{
    register flag;
    register struct tty *tp;
    register struct device *addr;
    int s;
    register int w4, w5, speed;

    tp = sc_ttptr[scdev(dev)].tt_tty;
    flag = tp->t_cflag;
    if (((flag&CBAUD) == B0) && (dev&MODEM)) { /* hang up line */
        schup(dev);
        return;
    }
    addr = (struct device *)sc_ttptr[scdev(dev)].tt_addr;
    w4 = W4CLK16;
    if (flag & CSTOPB)
        w4 |= W42STOP;
    else
        w4 |= W41STOP;
    w5 = W5TXENABLE;
    switch(flag & CSIZE) {
        case CS5:
            w5 |= W55BIT; break;
        case CS6:
            w5 |= W56BIT; break;
        case CS7:
            w5 |= W57BIT; break;
        case CS8:
            w5 |= W58BIT; break;
    }
    if (flag & PARENB)
        if (flag & PARODD)
            w4 |= W4PARENABLE;
        else
            w4 |= W4PARENABLE | W4PAREVEN;
    speed = sc_line[scdev(dev)].speed;
    speed = (speed/(scspeeds[flag&CBAUD]<<1)) - 2;
    s = spl6();
    addr->csr = 4;
    DELAY();
    addr->csr = w4;
    DELAY();
    addr->csr = 12;
    DELAY();
    addr->csr = speed;
    DELAY();
    addr->csr = 13;
    DELAY();
    addr->csr = speed >> 8;
    DELAY();
    addr->csr = 5;
    DELAY();
    addr->csr = w5;
    splx(s);
}

schup(dev)
{
    register struct device *addr;
    int s;

    dev = scdev(dev);
    addr = (struct device *)sc_ttptr[dev].tt_addr;
    s = spl6();

```

```

addr->csr = 5;
DELAY();
addr->csr = W5TXENABLE | W58BIT; /* turn off DTR/RTS */
splx(s);
}

scintr(ap)
register struct args *ap;
{
    register struct device *addr;

    for (ap->a_dev = 0; ap->a_dev < sc_cnt; ap->a_dev++) {
        addr = (struct device *)sc_ttptr[ap->a_dev].tt_addr;
        while (addr->csr & R0RXRDY) {
            addr->csr = 1;
            DELAY();
            if (addr->csr & (R1PARERR|R1OVRERR|R1FMERR))
                scsintr(ap);
            else
                scriintr(ap);
        }
        if (addr->csr & R0TXRDY)
            scxintr(ap);
    }
}

scintr(ap)
register struct args *ap;
{
    register struct device *addr;
    register struct ccblock *cbp;
    register int c, lcnt, flg;
    struct tty *tp;
    register char ctmp;
    char lbuf[3];

    addr = (struct device *)sc_ttptr[ap->a_dev].tt_addr;
    addr->csr = WORXINT; /* reinable receiver interrupt */
    addr->csr = W0RIUS; /* reset interrupt */
    c = addr->data & 0xFF;
    sysinfo.rcvint++;
    tp = sc_ttptr[ap->a_dev].tt_tty;
    if (tp->t_rbuf.c_ptr == NULL)
        return;
    if (tp->t_iflag & IXON) {
        ctmp = c & 0177;
        if (tp->t_state & TTSTOP) {
            if (ctmp == CSTART || tp->t_iflag & IXANY)
                (*tp->t_proc)(tp, T_RESUME);
        } else {
            if (ctmp == CSTOP)
                (*tp->t_proc)(tp, T_SUSPEND);
        }
        if (ctmp == CSTART || ctmp == CSTOP)
            return;
    }
    lcnt = 1;
    flg = tp->t_iflag;
    if (flg&ISTRIP)
        c &= 0177;
    else {
        /* c &= 0377; not needed */
        if (c == 0377 && flg&PARMRK) {
            lbuf[1] = 0377;
            lcnt = 2;
        }
    }
}
/*
 * Stash character in r_buf
 */
cbp = &tp->t_rbuf;
if (lcnt != 1) {
    lbuf[0] = c;
    while (lcnt) {
        *cbp->c_ptr++ = lbuf[--lcnt];
        if (--cbp->c_count == 0) {

```

```

        cbp->c_ptr -= cbp->c_size;
        (*linesw[tp->t_line].l_input)(tp);
    }
}
if (cbp->c_size != cbp->c_count) {
    cbp->c_ptr -= cbp->c_size - cbp->c_count;
    (*linesw[tp->t_line].l_input)(tp);
} else {
    *cbp->c_ptr = c;
    cbp->c_count--;
    (*linesw[tp->t_line].l_input)(tp);
}
}

```

```

scxintr(ap)
register struct args *ap;
{
    register short dev;
    struct tty *tp;
    register struct device *addr;

    sysinfo.xmtint++;
    dev = ap->a_dev;
    addr = (struct device *)sc_ttptr[dev].tt_addr;
    addr->csr = WORTXPND; /* reset transmitter interrupt */
    addr->csr = WORIOUS; /* reset interrupt */
    tp = sc_ttptr[dev].tt_tty;
    tp->t_state &= ~BUSY;
    scproc(tp, T_OUTPUT);
}

```

```

scsintr(ap)
register struct args *ap;
{
    register struct cblock *cbp;
    register int c, lcnt, flg;
    struct tty *tp;
    register char ctmp;
    char lbuf[3];
    register struct device *addr;
    unsigned char stat;

    sysinfo.rcvint++;
    addr = (struct device *)sc_ttptr[ap->a_dev].tt_addr;
    c = addr->data & 0xFF; /* read data BEFORE reset error */
    addr->csr = 0x1; /* cmd to read register 1 */
    stat = addr->csr; /* read the status */
    addr->csr = WORERROR; /* reset error condition */
    addr->csr = WORXINT; /* reinable receiver interrupt */
    addr->csr = WORIOUS; /* reset interrupt under service */
    tp = sc_ttptr[ap->a_dev].tt_tty;
    if (tp->t_rbuf.c_ptr == NULL)
        return;
    if (tp->t_iflag & IXON) {
        ctmp = c & 0177;
        if (tp->t_state & TTSTOP) {
            if (ctmp == CSTART || tp->t_iflag & IXANY)
                (*tp->t_proc)(tp, T_RESUME);
        } else {
            if (ctmp == CSTOP)
                (*tp->t_proc)(tp, T_SUSPEND);
        }
        if (ctmp == CSTART || ctmp == CSTOP)
            return;
    }
}
/*
 * Check for errors
 */
lcnt = 1;
flg = tp->t_iflag;
if (stat & (RIPARERR |RLOVRERR|RIFRMERR)) {
    if ((stat & RIPARERR) && (flg & INPCK))
        c |= PERROR;
    if (stat & RLOVRERR)
        c |= OVERRUN;
}

```

```

if (stat & RIFRMERR)
    c |= FRERROR;
}
if (c & (FRERROR|PERROR|OVERRUN)) {
    if ((c & 0377) == 0) {
        if (flg & IGNBRK)
            return;
        if (flg & BRKINT) {
            signal(tp->t_pgrp, SIGINT);
            ttyflush(tp, (FREAD|FWRITE));
            return;
        }
    } else {
        if (flg & IGNPAR)
            return;
    }
    if (flg & PARMRK) {
        lbuf[2] = 0377;
        lbuf[1] = 0;
        lcnt = 3;
        sysinfo.rawch += 2;
    } else
        c = 0;
} else {
    if (flg & ISTRIP)
        c &= 0177;
    else {
        /* c &= 0377; not needed */
        if (c == 0377 && flg & PARMRK) {
            lbuf[1] = 0377;
            lcnt = 2;
        }
    }
}
/*
 * Stash character in r_buf
 */
cbp = &tp->t_rbuf;
if (lcnt != 1) {
    while (lcnt) {
        *cbp->c_ptr++ = lbuf[--lcnt];
        if (--cbp->c_count == 0) {
            cbp->c_ptr -= cbp->c_size;
            (*linesw[tp->t_line].l_input)(tp);
        }
    }
    if (cbp->c_size != cbp->c_count) {
        cbp->c_ptr -= cbp->c_size - cbp->c_count;
        (*linesw[tp->t_line].l_input)(tp);
    }
} else {
    *cbp->c_ptr = c;
    cbp->c_count--;
    (*linesw[tp->t_line].l_input)(tp);
}
}
}
scscan()
{
    register int i;
    register struct tty *tp;
    register struct device *addr;

    timeout(scscan, (caddr_t)0, SCTIME);
    for (i = 0; i < sc_cnt; i++) {
        addr = (struct device *)sc_ttptr[i].tt_addr;
        addr->csr = WOREXT; /* update DCD */
        tp = sc_ttptr[i].tt_tty;
        if (addr->csr & R0DCD) {
            if ((tp->t_state & CARR_ON) == 0) {
                tp->t_state |= CARR_ON;
                if (tp->t_state & WOPEN)
                    wakeup((caddr_t)&tp->t_rawq);
            }
        }
    } else {
}

```

```
        if (tp->t_state&CARR_ON && sc_modem[i]) {
            tp->t_state &= ~CARR_ON;
            if (tp->t_state&ISOPEN) {
                ttyflush(tp, FREAD|FWRITE);
                signal(tp->t_pgrp, SIGHUP);
            }
        }
    }
}

#ifdef SCC_CONSOLE
sputc(char(c))
{
    register struct device *addr ;
    int s, i;

    addr = (struct device *)sc_ttptr[CONSOLE].tt_addr;
    s = spl6();
    if (c == '\n')
        sputc(char('\r'));
    i = 10000;
    while ((addr->csr & ROTXRDY) == 0 && --i) ;
    addr->data = c;
    splx(s);
}
#endif
#endif
```

```

/*
 * SCC device driver
 *
 * Copyright 1982 UniSoft Corporation
 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/file.h"
#include "sys/tty.h"
#include "sys/termio.h"
#include "sys/conf.h"
#include "sys/sysinfo.h"
#include "sys/var.h"
#include "sys/reg.h"
#include <sys/scc.h>
#include "sys/proc.h"
#include "setjmp.h"

int scproc();
extern int sc_cnt;
extern struct tty sc_tty[];
extern struct ttyptr sc_ttptr[];
extern char sc_modem[];
extern struct scline sc_line[];

#define MODEM 0x80 /* modem control on bit */
#define scdev(d) ((d)&0x7f) /* from unix device number to device */

#define DELAY() asm("\tnop");

#define SCTIME (v.v_hz*5) /* scscan interval */

/*
 * Note: to select baud rate
 * k = chip_input_frequency/(2 * baud * factor) - 2
 * put factor in Register 9 and k in registers D & C
 * NOTE:
 * normally, factor = 16
 * for this driver, chip_input_frequency = 2400000 Hz
 * scspeeds is a table of these numbers by UNIX baud rate
 */
int scspeeds[] = {
    1, 50, 75, 110, 134, 150, 200, 300,
    600, 1200, 1800, 2400, 4800, 9600, 19200, 38400,
};

/*
 * table to initialize a port to 9600 baud
 */
char scitable[] = {
    WONULL,
    9, 0, /* set according to sc_line reset value */
#define SCCIRESET scitable[2]
    4, W4CLK16 | W41STOP,
    10, 0,
    11, W11RBR | W11TBR,
    12, 0, /* 12/13 are baud rate, from sc_line speed value */
#define SCCISPLO scitable[10]
    13, 0,
#define SCCISPHI scitable[12]
    14, W14BRGE | W14BRINT,
    3, W38BIT | W38XENABLE,
    5, W58BIT | W58XENABLE,
    1, W1RXIAL | W1TXIEN /*| W1EXTIEN*/,
    2, 0, /* auto vector */
    0, WORXINT,
    15, 0,
    9, W9MIE | W9DLC,

```

```

};

/*
 * we call this in startup() to preinitialize all the ports
 */
scinit()
{
    register struct device *addr;
    unsigned short nsc;

    for(nsc=0;nsc<500;nsc++);
    for (nsc = 0; nsc < sc_cnt; nsc++) {
        addr = (struct device *)sc_ttptr[nsc].tt_addr;
        /* do proper reset */
        scinil(addr, W9NV|sc_line[nsc].reset, (int)sc_line[nsc].speed);
    }
    printf("%d serial ports\n", nsc);
    /* scscan(); /* start modem scanning */
}

scinil(addr, reset, speed)
struct device *addr;
int reset, speed;
{
    register int i;
    int s;

    s = spl6();
    SCCIRESET = reset;
    speed = (speed/(9600<<1)) - 2;
    SCCISPHI = (speed >> 8) & 0xFF;
    SCCISPLO = speed & 0xFF;
    for (i = 0; i < sizeof(scitable); i++) {
        DELAY();
        addr->csr = scitable[i];
    }
    DELAY();
    splx(s);
}

/* ARGSUSED */
scopen(dev, flag)
register dev;
{
    register struct tty *tp;
    register struct device *addr;
    register d;
#ifdef SINGLEUSER
    register struct proc *p;
#endif SINGLEUSER

    d = scdev(dev);
    if (d >= sc_cnt) {
        u.u_error = ENXIO;
        return;
    }
    tp = sc_ttptr[d].tt_tty;
#ifdef SINGLEUSER
    p = u.u_proc;
    if ((p->p_pid == p->p_pgrp)
        && (u.u_ttyp == NULL)
        && (tp->t_pgrp == 0)) {
        u.u_error = ENOTTY;
        return;
    }
#endif SINGLEUSER

    tp->t_index = d;
    SPL6();
    if ((tp->t_state&(ISOPEN|WOPEN)) == 0) {
        tp->t_proc = scproc;
        ttinit(tp);
        sc_modem[d] = dev & MODEM;
        addr = (struct device *)sc_ttptr[d].tt_addr;
        if (dev & MODEM && (addr->csr & R0DCD) == 0)
            tp->t_state = WOPEN;
        else

```



```

        tp->t_state = WOPEN | CARR_ON;
#ifdef SCC_CONSOLE
        if (d == CONSOLE) {
            tp->t_iflag = ICRNL | ISTRIP;
            tp->t_oflag = OPOST | ONLCR | TAB3;
            tp->t_lflag = ISIG | ICANON | ECHO | ECHOK;
            tp->t_cflag = sspeed | CS8 | CREAD | HUPCL;
        }
#endif
        scparam(dev);
    }
    if (!(flag & FNDELAY))
        while ((tp->t_state & CARR_ON) == 0)
            (void) sleep((caddr_t)&tp->t_rawq, TTOPRI);
    SPL0();
    (*linesw[tp->t_line].l_open)(tp);
}

/* ARGSUSED */
scclose(dev, flag)
{
    register struct tty *tp;
    register int d;

    d = scdev(dev);
    tp = sc_ttptr[d].tt_tty;
    (*linesw[tp->t_line].l_close)(tp);
    if (tp->t_cflag & HUPCL)
        schup(dev);          /* hang up */
}

scread(dev)
{
    register struct tty *tp;

    tp = sc_ttptr[scdev(dev)].tt_tty;
    (*linesw[tp->t_line].l_read)(tp);
}

scwrite(dev)
{
    register struct tty *tp;

    tp = sc_ttptr[scdev(dev)].tt_tty;
    (*linesw[tp->t_line].l_write)(tp);
}

scproc(tp, cmd)
register struct tty *tp;
{
    register struct cblock *tbuf;
    register struct device *addr;
    register dev_t dev;
    extern ttrstrt();
    int s;

    s = spl6();
    dev = tp->t_index;
    addr = (struct device *)sc_ttptr[dev].tt_addr;
    switch (cmd) {
        case T_TIME:
            scw5(tp, addr, 0);          /* clear break */
            tp->t_state &= ~TIMEOUT;
            goto start;
        case T_WFLUSH:
            tbuf = &tp->t_tbuf;
            tbuf->c_size -= tbuf->c_count;
            tbuf->c_count = 0;
            /* fall through */
        case T_RESUME:
            tp->t_state &= ~TTSTOP;
            goto start;
        case T_OUTPUT:

```

```

start:
    if (tp->t_state & (TTSTOP|TIMEOUT|BUSY))
        break;
    if (tp->t_state & TTXOFF) {
        tp->t_state &= ~TTXOFF;
        tp->t_state |= BUSY;
        addr->data = CSTOP;
        break;
    }
    if (tp->t_state & TTXON) {
        tp->t_state &= ~TTXON;
        tp->t_state |= BUSY;
        addr->data = CSTART;
        break;
    }
    tbuf = &tp->t_tbuf;
    if ((tbuf->c_ptr == 0) || (tbuf->c_count == 0)) {
        if (tbuf->c_ptr)
            tbuf->c_ptr -= tbuf->c_size;
        if (!(CPRES & (*linesw[tp->t_line].l_output)(tp)))
            break;
    }
    tp->t_state |= BUSY;
    addr->data = *tbuf->c_ptr++;
    tbuf->c_count--;
    break;
case T_SUSPEND:
    tp->t_state |= TTSTOP;
    break;
case T_BLOCK:
    tp->t_state &= ~TTXON;
    tp->t_state |= TBLOCK;
    tp->t_state |= TTXOFF;
    goto start;
case T_RFLUSH:
    if (!(tp->t_state & TBLOCK))
        break;
    /* fall through */
case T_UNBLOCK:
    tp->t_state &= ~(TTXOFF|TBLOCK);
    tp->t_state |= TTXON;
    goto start;
case T_BREAK:
    scw5(tp, addr, W5BREAK);
    tp->t_state |= TIMEOUT;
    timeout(ttrstrt, (caddr_t)tp, v.v_hz>>2);
    break;
}
splx(s);

scw5(tp, addr, d)
struct tty *tp;
struct device *addr;
int d;
{
    register int w5;
    int s;

    w5 = W5TXENABLE | d;
    switch(tp->t_cflag & CSIZE) {
        case CS5:
            w5 |= W55BIT; break;
        case CS6:
            w5 |= W56BIT; break;
        case CS7:
            w5 |= W57BIT; break;
        case CS8:
            w5 |= W58BIT; break;
    }
    s = spl5();

```

```

addr->csr = 5;
addr->csr = w5;
splx(s);
}

sioctl(dev, cmd, arg, mode)
{
    if (ttiocom(sc_ttptr[scdev(dev)].tt_tty, cmd, arg, mode))
        scparam(dev);
}

scparam(dev)
{
    register flag;
    register struct tty *tp;
    register struct device *addr;
    int s;
    register int w4, w5, speed;

    tp = sc_ttptr[scdev(dev)].tt_tty;
    flag = tp->t_cflag;
    if (((flag & CBAUD) == B0) && (dev & MODEM)) { /* hang up line */
        schup(dev);
        return;
    }
    addr = (struct device *)sc_ttptr[scdev(dev)].tt_addr;
    w4 = W4CLK16;
    if (flag & CSTOPB)
        w4 |= W42STOP;
    else
        w4 |= W41STOP;
    w5 = W5TXENABLE;
    switch(flag & CSIZE) {
        case CS5:
            w5 |= W55BIT; break;
        case CS6:
            w5 |= W56BIT; break;
        case CS7:
            w5 |= W57BIT; break;
        case CS8:
            w5 |= W58BIT; break;
    }
    if (flag & PARENB)
        if (flag & PARODD)
            w4 |= W4PARENABLE;
        else
            w4 |= W4PARENABLE | W4PAREVEN;
    speed = sc_line[scdev(dev)].speed;
    speed = (speed / (scspeeds[flag & CBAUD] << 1)) - 2;
    s = spl6();
    addr->csr = 4;
    DELAY();
    addr->csr = w4;
    DELAY();
    addr->csr = 12;
    DELAY();
    addr->csr = speed;
    DELAY();
    addr->csr = 13;
    DELAY();
    addr->csr = speed >> 8;
    DELAY();
    addr->csr = 5;
    DELAY();
    addr->csr = w5;
    splx(s);
}

schup(dev)
{
    register struct device *addr;
    int s;

    dev = scdev(dev);
    addr = (struct device *)sc_ttptr[dev].tt_addr;
    s = spl6();

```

```

addr->csr = 5;
DELAY();
addr->csr = W5TXENABLE | W58BIT; /* turn off DTR/RTS */
splx(s);
}

scintr(ap)
register struct args *ap;
{
    register struct device *addr;

    for (ap->a_dev = 0; ap->a_dev < sc_cnt; ap->a_dev++) {
        addr = (struct device *)sc_ttptr[ap->a_dev].tt_addr;
        while (addr->csr & RORXRDY) {
            addr->csr = 1;
            DELAY();
            if (addr->csr & (R1PARERR|R1OVRERR|R1FRMERR))
                scsintr(ap);
            else
                scintr(ap);
        }
        if (addr->csr & ROTXRDY)
            scxintr(ap);
    }
}

scintr(ap)
register struct args *ap;
{
    register struct device *addr;
    register struct ccblock *cbp;
    register int c, lcnt, flg;
    struct tty *tp;
    register char ctmp;
    char lbuf[3];

    addr = (struct device *)sc_ttptr[ap->a_dev].tt_addr;
    addr->csr = WORXINT; /* reinable receiver interrupt */
    addr->csr = WORJUS; /* reset interrupt */
    c = addr->data & 0xFF;
    sysinfo.rcvint++;
    tp = sc_ttptr[ap->a_dev].tt_tty;
    if (tp->t_rbuf.c_ptr == NULL)
        return;
    if (tp->t_iflag & IXON) {
        ctmp = c & 0177;
        if (tp->t_state & TTSTOP) {
            if (ctmp == CSTART || tp->t_iflag & IXANY)
                (*tp->t_proc)(tp, T_RESUME);
        } else {
            if (ctmp == CSTOP)
                (*tp->t_proc)(tp, T_SUSPEND);
        }
        if (ctmp == CSTART || ctmp == CSTOP)
            return;
    }
    lcnt = 1;
    flg = tp->t_iflag;
    if (flg & ISTRIP)
        c &= 0177;
    else {
        /* c &= 0377; not needed */
        if (c == 0377 && flg & PARMRK) {
            lbuf[1] = 0377;
            lcnt = 2;
        }
    }
}
/*
 * stash character in r_buf
 */
cbp = &tp->t_rbuf;
if (lcnt != 1) {
    lbuf[0] = c;
    while (lcnt) {
        *cbp->c_ptr++ = lbuf[--lcnt];
        if (--cbp->c_count == 0) {

```

```

        cbp->c_ptr -= cbp->c_size;
        (*linesw[tp->t_line].l_input)(tp);
    }
}
if (cbp->c_size != cbp->c_count) {
    cbp->c_ptr -= cbp->c_size - cbp->c_count;
    (*linesw[tp->t_line].l_input)(tp);
}
} else {
    *cbp->c_ptr = c;
    cbp->c_count--;
    (*linesw[tp->t_line].l_input)(tp);
}
}

```

```
scxintr(ap)
```

```
register struct args *ap;
```

```

{
    register short dev;
    struct tty *tp;
    register struct device *addr;

    sysinfo.xmtint++;
    dev = ap->a_dev;
    addr = (struct device *)sc_ttptr[dev].tt_addr;
    addr->csr = WORTXPNDR; /* reset transmitter interrupt */
    addr->csr = WORIOUS; /* reset interrupt */
    tp = sc_ttptr[dev].tt_tty;
    tp->t_state &= ~BUSY;
    scproc(tp, T_OUTPUT);
}

```

```
scsintr(ap)
```

```
register struct args *ap;
```

```

{
    register struct cblock *cbp;
    register int c, lcnt, flg;
    struct tty *tp;
    register char ctmp;
    char lbuf[3];
    register struct device *addr;
    unsigned char stat;

    sysinfo.rcvint++;
    addr = (struct device *)sc_ttptr[ap->a_dev].tt_addr;
    c = addr->data & 0xFF; /* read data BEFORE reset error */
    addr->csr = 0x1; /* cmd to read register 1 */
    stat = addr->csr; /* read the status */
    addr->csr = WORERROR; /* reset error condition */
    addr->csr = WORXINT; /* reinable receiver interrupt */
    addr->csr = WORIOUS; /* reset interrupt under service */
    tp = sc_ttptr[ap->a_dev].tt_tty;
    if (tp->t_rbuf.c_ptr == NULL)
        return;
    if (tp->t_iflag & IXON) {
        ctmp = c & 0177;
        if (tp->t_state & TTSTOP) {
            if (ctmp == CSTART || tp->t_iflag & IXANY)
                (*tp->t_proc)(tp, T_RESUME);
        } else {
            if (ctmp == CSTOP)
                (*tp->t_proc)(tp, T_SUSPEND);
        }
        if (ctmp == CSTART || ctmp == CSTOP)
            return;
    }
}
/*
 * Check for errors
 */
lcnt = 1;
flg = tp->t_iflag;
if (stat & (R1PARERR | R1OVRERR | R1FRMERR)) {
    if ((stat & R1PARERR) && (flg & INPCK))
        c |= PERROR;
    if (stat & R1OVRERR)
        c |= OVERRUN;
}

```

```

    if (stat & R1FRMERR)
        c |= FRERROR;
}
if (c & (FRERROR | PERROR | OVERRUN)) {
    if ((c & 0377) == 0) {
        if (flg & IGBRK)
            return;
        if (flg & BRKINT) {
            signal(tp->t_pgrp, SIGINT);
            ttyflush(tp, (FREAD | FWRITE));
            return;
        }
    } else {
        if (flg & IGNPAR)
            return;
    }
    if (flg & PARMRK) {
        lbuf[2] = 0377;
        lbuf[1] = 0;
        lcnt = 3;
        sysinfo.rawch += 2;
    } else
        c = 0;
} else {
    if (flg & ISTRIP)
        c &= 0177;
    else {
        /* c &= 0377; not needed */
        if (c == 0377 && flg & PARMRK) {
            lbuf[1] = 0377;
            lcnt = 2;
        }
    }
}
/*
 * Stash character in r_buf
 */
cbp = &tp->t_rbuf;
if (lcnt != 1) {
    lbuf[0] = c;
    while (lcnt) {
        *cbp->c_ptr++ = lbuf[--lcnt];
        if (--cbp->c_count == 0) {
            cbp->c_ptr -= cbp->c_size;
            (*linesw[tp->t_line].l_input)(tp);
        }
    }
    if (cbp->c_size != cbp->c_count) {
        cbp->c_ptr -= cbp->c_size - cbp->c_count;
        (*linesw[tp->t_line].l_input)(tp);
    }
} else {
    *cbp->c_ptr = c;
    cbp->c_count--;
    (*linesw[tp->t_line].l_input)(tp);
}
}

scscan()
{
    register int i;
    register struct tty *tp;
    register struct device *addr;

    timeout(scscan, (caddr_t)0, SCTIME);
    for (i = 0; i < sc_cnt; i++) {
        addr = (struct device *)sc_ttptr[i].tt_addr;
        addr->csr = WOREXT; /* update DCD */
        tp = sc_ttptr[i].tt_tty;
        if (addr->csr & RODCD) {
            if ((tp->t_state & CARR_ON) == 0) {
                tp->t_state |= CARR_ON;
                if (tp->t_state & WOPEN)
                    wakeup((caddr_t)&tp->t_rawq);
            }
        } else {

```

```
        if (tp->t_state&CARR_ON && sc_modem[i]) {
            tp->t_state &= ~CARR_ON;
            if (tp->t_state&ISOPEN) {
                ttyflush(tp, FREAD|FWRITE);
                signal(tp->t_pgrp, SIGHUP);
            }
        }
    }
}

#ifdef SCC_CONSOLE
sputc(c)
{
    register struct device *addr;
    int s, i;

    addr = (struct device *)sc_ttptr[CONSOLE].tt_addr;
    s = spl6();
    if (c == '\n')
        sputc('\r');
    i = 100000;
    while ((addr->csr & R0TXRDY) == 0 && --i);
    addr->data = c;
    splx(s);
}
#endif SCC_CONSOLE
```

```

/* 0(#)sem.c 1.5 */
/*
**      Inter-Process Communication Semaphore Facility.
**
#include "sys/types.h"
#include "sys/param.h"
#include "sys/dir.h"
#ifdef u3b
#include "sys/istk.h"
#endif
#include "sys/map.h"
#include "sys/errno.h"
#include "sys/signal.h"
#include "sys/ipc.h"
#include "sys/sem.h"
#include "sys/user.h"
#include "sys/seg.h"
#include "sys/proc.h"
#include "sys/buf.h"

#ifdef pdp11
#define MOVE      sempmove
#else
#define MOVE      iomove
#endif

extern struct semid_ds  sema[];      /* semaphore data structures */
extern struct sem       sem[];      /* semaphores */
extern struct map       semmap[];    /* sem allocation map */
extern struct sem_undo  *sem_undo[]; /* undo table pointers */
extern struct sem_undo  semu[];     /* operation adjust on exit table */
extern struct seminfo  seminfo;     /* param information structure */
extern union {
    short          semvals[1]; /* set semaphore values */
    struct semid_ds ds;        /* set permission values */
    struct sembuf  semops[1];  /* operation holding area */
} semtmp;
struct sem_undo *semunp; /* ptr to head of undo chain */
struct sem_undo *semfup; /* ptr to head of free undo chain */

extern time_t  time; /* system idea of date */

struct ipc_perm *ipcget();
struct semid_ds *semconv();

/*
**      semaoe - Create or update adjust on exit entry.
**
semaoe(val, id, num)
short  val, /* operation value to be adjusted on exit */
num; /* semaphore # */
int    id; /* semid */
{
    register struct undo      *uup, /* ptr to entry to update */
                          *uup2; /* ptr to move entry */
    register struct sem_undo  *up, /* ptr to process undo struct */
                          *up2; /* ptr to undo list */
    register int              i, /* loop control */
                          found; /* matching entry found flag */

    if(val == 0)
        return(0);
    if(val > seminfo.semaem || val < -seminfo.semaem) {
        u.u_error = ERANGE;
        return(1);
    }
    if((up = sem_undo[u.u_procp - procp]) == NULL)
        if (up = semfup) {
            semfup = up->un_np;
            up->un_np = NULL;
            sem_undo[u.u_procp - procp] = up;
        } else {
            u.u_error = ENOSPC;
            return(1);
        }
}

for(uup = up->un_ent, found = i = 0; i < up->un_cnt; i++) {
    if(uup->un_id < id || (uup->un_id == id && uup->un_num < num)) {
        uup++;
        continue;
    }
    if(uup->un_id == id && uup->un_num == num)
        found = 1;
    break;
}
if(!found) {
    if(up->un_cnt >= seminfo.semume) {
        u.u_error = EINVAL;
        return(1);
    }
    if(up->un_cnt == 0) {
        up->un_np = semunp;
        semunp = up;
    }
    uup2 = &up->un_ent[up->un_cnt++];
    while(uup2-- > uup)
        *(uup2 + 1) = *uup2;
    uup->un_id = id;
    uup->un_num = num;
    uup->un_aoe = -val;
    return(0);
}
uup->un_aoe -= val;
if(uup->un_aoe > seminfo.semaem || uup->un_aoe < -seminfo.semaem) {
    u.u_error = ERANGE;
    uup->un_aoe += val;
    return(1);
}
if(uup->un_aoe == 0) {
    uup2 = &up->un_ent[--(up->un_cnt)];
    while(uup2+ < uup2)
        *(uup - 1) = *uup;
    if(up->un_cnt == 0) {
        /* Remove process from undo list. */
        if(semunp == up)
            semunp = up->un_np;
        else
            for(up2 = semunp; up2 != NULL; up2 = up2->un_np)
                if(up2->un_np == up) {
                    up2->un_np = up->un_np;
                    break;
                }
        up->un_np = NULL;
    }
}
return(0);
}

/*
**      semconv - Convert user supplied semid into a ptr to the associated
**                semaphore header.
*/

struct semid_ds *
semconv(s)
register int    s; /* semid */
{
    register struct semid_ds *sp; /* ptr to associated header */

    sp = &sema[s % seminfo.semnm1];
    if((sp->sem_perm.mode & IPC_ALLOC) == 0 ||
        s / seminfo.semnm1 != sp->sem_perm.seq) {
        u.u_error = EINVAL;
        return(NULL);
    }
    return(sp);
}

/*
**      semctl - Semctl system call.

```

```

*/
semctl()
{
    register struct a {
        int    semid;
        uint   semnum;
        int    cmd;
        int    arg;
    } *uap = (struct a *)u.u_ap;
    register struct semid_ds *sp; /* ptr to semaphore header */
    register struct sem *p; /* ptr to semaphore */
    register int i; /* loop control */
    register struct user *up;

    if((sp = semconv(uap->semid)) == NULL)
        return;

    up = &u;
    up->u_rvall = 0;
    switch(uap->cmd) {

/* Remove semaphore set. */
    case IPC_RMID:
        if(up->u_uid != sp->sem_perm.uid && up->u_uid != sp->sem_perm.cuid
            && !suser())
            return;
        semunrm(uap->semid, 0, sp->sem_nsems);
        for(i = sp->sem_nsems, p = sp->sem_base; i-->0; p++) {
            p->semval = p->sempid = 0;
            if(p->semncnt) {
                wakeup((caddr_t)&p->semncnt);
                p->semncnt = 0;
            }
            if(p->semzcnt) {
                wakeup((caddr_t)&p->semzcnt);
                p->semzcnt = 0;
            }
        }
        mfree(semmap, (int)sp->sem_nsems, (sp->sem_base - sem) + 1);
        if(uap->semid + seminfo.semnm1 < 0)
            sp->sem_perm.seq = 0;
        else
            sp->sem_perm.seq++;
        sp->sem_perm.mode = 0;
        return;

/* Set ownership and permissions. */
    case IPC_SET:
        if(up->u_uid != sp->sem_perm.uid && up->u_uid != sp->sem_perm.cuid
            && !suser())
            return;
        if(copyin((caddr_t)uap->arg, (caddr_t)&semtmp.ds, sizeof(semtmp.ds))) {
            up->u_error = EFAULT;
            return;
        }
        sp->sem_perm.uid = semtmp.ds.sem_perm.uid;
        sp->sem_perm.gid = semtmp.ds.sem_perm.gid;
        sp->sem_perm.mode = semtmp.ds.sem_perm.mode & 0777 | IPC_ALLOC;
        sp->sem_ctime = time;
        return;

/* Get semaphore data structure. */
    case IPC_STAT:
        if(ipcaccess(&sp->sem_perm, SEM_R))
            return;
        if(copyout((caddr_t)sp, (caddr_t)uap->arg, sizeof(*sp))) {
            up->u_error = EFAULT;
            return;
        }
        return;

/* Get # of processes sleeping for greater semval. */
    case GETNCNT:
        if(ipcaccess(&sp->sem_perm, SEM_R))
            return;
        if(uap->semnum >= sp->sem_nsems) {
            up->u_error = EINVAL;
            return;
        }
        up->u_rvall = (sp->sem_base + uap->semnum)->semncnt;
        return;

/* Get pid of last process to operate on semaphore. */
    case GETPID:
        if(ipcaccess(&sp->sem_perm, SEM_R))
            return;
        if(uap->semnum >= sp->sem_nsems) {
            up->u_error = EINVAL;
            return;
        }
        up->u_rvall = (sp->sem_base + uap->semnum)->sempid;
        return;

/* Get semval of one semaphore. */
    case GETVAL:
        if(ipcaccess(&sp->sem_perm, SEM_R))
            return;
        if(uap->semnum >= sp->sem_nsems) {
            up->u_error = EINVAL;
            return;
        }
        up->u_rvall = (sp->sem_base + uap->semnum)->semval;
        return;

/* Get all semvals in set. */
    case GETALL:
        if(ipcaccess(&sp->sem_perm, SEM_R))
            return;
        up->u_base = (caddr_t)uap->arg;
        up->u_offset = 0;
        up->u_segflg = 0;
        for(i = sp->sem_nsems, p = sp->sem_base; i-->0; p++) {
            MOVE((caddr_t)&p->semval, sizeof(p->semval), B_READ);
            if(up->u_error)
                return;
        }
        return;

/* Get # of processes sleeping for semval to become zero. */
    case GETZCNT:
        if(ipcaccess(&sp->sem_perm, SEM_R))
            return;
        if(uap->semnum >= sp->sem_nsems) {
            up->u_error = EINVAL;
            return;
        }
        up->u_rvall = (sp->sem_base + uap->semnum)->semzcnt;
        return;

/* Set semval of one semaphore. */
    case SETVAL:
        if(ipcaccess(&sp->sem_perm, SEM_A))
            return;
        if(uap->semnum >= sp->sem_nsems) {
            up->u_error = EINVAL;
            return;
        }
        if((unsigned)uap->arg > seminfo.semvmx) {
            up->u_error = ERANGE;
            return;
        }
        if((p = sp->sem_base + uap->semnum)->semval == uap->arg) {
            if(p->semncnt) {
                p->semncnt = 0;
                wakeup((caddr_t)&p->semncnt);
            }
        } else
            if(p->semzcnt) {
                p->semzcnt = 0;
                wakeup((caddr_t)&p->semzcnt);
            }
        p->semval = uap->arg;
        p->sempid = up->u_procp->p_pid;
    }
}

```

```

semunrm(uap->semid, uap->semnum, uap->semnum);
return;

/* Set semvals of all semaphores in set. */
case SETALL:
    if(ipcaccess(&sp->sem_perm, SEM_A))
        return;
    up->u_base = (caddr_t)uap->arg;
    up->u_offset = 0;
    up->u_segflg = 0;
    MOVE((caddr_t)semtmp.semvals,
        (int)(sizeof(semtmp.semvals[0]) * sp->sem_nsems),
        B_WRITE);
    if(up->u_error)
        return;
    for(i = 0; i < sp->sem_nsems; i++)
        if(semtmp.semvals[i] > seminfo.semvmx) {
            up->u_error = ERANGE;
            return;
        }
    semunrm(uap->semid, 0, sp->sem_nsems);
    for(i = 0, p = sp->sem_base; i < sp->sem_nsems; i++)
        if(p->semid == up->u_proc->p_pid) {
            if(p->semval == semtmp.semvals[i]) {
                if(p->semcnt) {
                    p->semcnt = 0;
                    wakeup((caddr_t)&p->semcnt);
                }
            } else
                if(p->semzcnt) {
                    p->semzcnt = 0;
                    wakeup((caddr_t)&p->semzcnt);
                }
        }
    return;
default:
    up->u_error = EINVAL;
    return;
}

/*
** semexit - Called by exit(sys1.c) to clean up on process exit.
*/

semexit()
{
    register struct sem_undo      *up, /* process undo struct ptr */
                                *p; /* undo struct ptr */
    register struct semid_ds     *sp; /* semid being undone ptr */
    register int                 i; /* loop control */
    register long                v; /* adjusted value */
    register struct sem          *semp; /* semaphore ptr */

    if((up = sem_undo[u.u_proc - proc]) == NULL)
        return;
    if(up->un_cnt == 0)
        goto cleanup;
    for(i = up->un_cnt; i--;) {
        if((sp = semconv(up->un_ent[i].un_id)) == NULL)
            continue;
        v = (long)(semp = sp->sem_base + up->un_ent[i].un_num)->semval +
            up->un_ent[i].un_aoe;
        if(v < 0 || v > seminfo.semvmx)
            continue;
        semp->semval = v;
        if(v == 0 && semp->semzcnt) {
            semp->semzcnt = 0;
            wakeup((caddr_t)&semp->semzcnt);
        }
        if(up->un_ent[i].un_aoe > 0 && semp->semcnt) {
            semp->semcnt = 0;
            wakeup((caddr_t)&semp->semcnt);
        }
    }
    up->un_cnt = 0;
}

if(semunp == up)
    semunp = up->un_np;
else
    for(p = semunp; p != NULL; p = p->un_np)
        if(p->un_np == up) {
            p->un_np = up->un_np;
            break;
        }

cleanup:
    up->un_np = semfup;
    semfup = up;
    sem_undo[u.u_proc - proc] = NULL;
}

/*
** semget - Semget system call.
*/

semget()
{
    register struct a {
        key_t   key;
        int     nsems;
        int     semflg;
    } *uap = (struct a *)u.u_ap;
    register struct semid_ds *sp; /* semaphore header ptr */
    register int i; /* temp */
    int s; /* ipcget status return */

    if((sp = (struct semid_ds *)
        ipcget(uap->key, uap->semflg, (struct ipc_perm *)sema,
        seminfo.semnum, sizeof(*sp), &s)) == NULL)
        return;
    if(s) {
        /* This is a new semaphore set. Finish initialization. */
        if(uap->nsems <= 0 || uap->nsems > seminfo.semmsl) {
            u.u_error = EINVAL;
            sp->sem_perm.mode = 0;
            return;
        }
        if((i = malloc(semmap, uap->nsems)) == NULL) {
            u.u_error = ENOSPC;
            sp->sem_perm.mode = 0;
            return;
        }
        sp->sem_base = sem + (i - 1);
        sp->sem_nsems = uap->nsems;
        sp->sem_ctime = time;
    } else
        if(uap->nsems && sp->sem_nsems < uap->nsems) {
            u.u_error = EINVAL;
            return;
        }
    u.u_rvall = sp->sem_perm.seq * seminfo.semnum + (sp - sema);
}

/*
** seminit - Called by main(main.c) to initialize the semaphore map.
*/

seminit()
{
    register i;

    mapinit(semmap, seminfo.semmap);
    mfree(semmap, seminfo.semms, 1);

    semfup = semu;
    for (i = 0; i < seminfo.semnum - 1; i++) {
        semfup->un_np = (struct sem_undo *)((uint)semfup+seminfo.semusz);
        semfup = semfup->un_np;
    }
    semfup->un_np = NULL;
    semfup = semu;
}

```

```

/*
** semop - Semop system call.
**
semop()
{
    register struct a {
        int          semid;
        struct sembuf *sops;
        uint         nsops;
    } *uap = (struct a *)u.u_ap;
    register struct sembuf *op; /* ptr to operation */
    register int i; /* loop control */
    register struct semid_ds *sp; /* ptr to associated header */
    register struct sem *semp; /* ptr to semaphore */
    int again;

    if((sp = semconv(uap->semid)) == NULL)
        return;
    if(uap->nsops > seminfo.semopm) {
        u.u_error = E2BIG;
        return;
    }
    u.u_base = (caddr_t)uap->sops;
    u.u_offset = 0;
    u.u_segflg = 0;
    MOVE((caddr_t)semtmp.semops, (int)(uap->nsops * sizeof(*op)), B_WRITE);
    if(u.u_error)
        return;

    /* Verify that sem $s are in range and permissions are granted. */
    for(i = 0, op = semtmp.semops; i++ < uap->nsops; op++) {
        if(ipcaccess(&sp->sem_perm, (ushort)(op->sem_op?SEM_A:SEM_R)))
            return;
        if(op->sem_num >= sp->sem_nsems) {
            u.u_error = EFBIG;
            return;
        }
    }
    again = 0;

check:
    /* Loop waiting for the operations to be satisfied atomically. */
    /* Actually, do the operations and undo them if a wait is needed
    or an error is detected. */
    if (again) {
        /* Verify that the semaphores haven't been removed. */
        if(semconv(uap->semid) == NULL) {
            u.u_error = EIDRM;
            return;
        }
        /* copy in user operation list after sleep */
        u.u_base = (caddr_t)uap->sops;
        u.u_offset = 0;
        u.u_segflg = 0;
        MOVE((caddr_t)semtmp.semops,
            (int)(uap->nsops * sizeof(*op)), B_WRITE);
        if(u.u_error)
            return;
    }
    again = 1;

    for(i = 0, op = semtmp.semops; i < uap->nsops; i++, op++) {
        semp = sp->sem_base + op->sem_num;
        if(op->sem_op > 0) {
            if(op->sem_op + (long)semp->semval > seminfo.semvmx ||
                (op->sem_flg & SEM_UNDO &&
                 semaoe(op->sem_op, uap->semid, (short)op->sem_num)) {
                if(u.u_error == 0)
                    u.u_error = ERANGE;
                if(!i)
                    semundo(semtmp.semops, i, uap->semid, sp);
                return;
            }
            semp->semval += op->sem_op;
            if(semp->semcnt) {

```

```

                semp->semcnt = 0;
                wakeup((caddr_t)&semp->semcnt);
            }
            continue;
        }
        if(op->sem_op < 0) {
            if(semp->semval >= -op->sem_op) {
                if(op->sem_flg & SEM_UNDO &&
                    semaoe(op->sem_op, uap->semid, (short)op->sem_num)) {
                    if(!i)
                        semundo(semtmp.semops, i, uap->semid, sp);
                    return;
                }
                semp->semval += op->sem_op;
                if(semp->semval == 0 && semp->semzcnt) {
                    semp->semzcnt = 0;
                    wakeup((caddr_t)&semp->semzcnt);
                }
                continue;
            }
            if(!i)
                semundo(semtmp.semops, i, uap->semid, sp);
            if(op->sem_flg & IPC_NOWAIT) {
                u.u_error = EAGAIN;
                return;
            }
            semp->semcnt++;
            if(sleep((caddr_t)&semp->semcnt, PCATCH | PSEMN)) {
                if((semp->semcnt)-- <= 1) {
                    semp->semcnt = 0;
                    wakeup((caddr_t)&semp->semcnt);
                }
                u.u_error = EINTR;
                return;
            }
            goto check;
        }
        if(semp->semval) {
            if(!i)
                semundo(semtmp.semops, i, uap->semid, sp);
            if(op->sem_flg & IPC_NOWAIT) {
                u.u_error = EAGAIN;
                return;
            }
            semp->semzcnt++;
            if(sleep((caddr_t)&semp->semzcnt, PCATCH | PSEMZ)) {
                if((semp->semzcnt)-- <= 1) {
                    semp->semzcnt = 0;
                    wakeup((caddr_t)&semp->semzcnt);
                }
                u.u_error = EINTR;
                return;
            }
            goto check;
        }
    }

    /* All operations succeeded. Update semid for accessed semaphores. */
    for(i = 0, op = semtmp.semops; i++ < uap->nsops;
        (sp->sem_base + (op++)->sem_num)->semid = u.u_procp->p_pid;
        sp->sem_otime = time;
        u.u_rvall = 0;
    )
}

#ifdef pdp11
/*
** sempimove - PDP 11 pimove interface for possibly large copies.
**
sempimove(base, count, mode)
paddr_t base; /* base address */
register unsigned count; /* byte count */
int mode; /* transfer mode */
{
    register unsigned tcount; /* current transfer count */

```



```

while(u.u_error == 0 && count) {
    tcount = count > 8064 ? 8064 : count;
    pmove(base, tcount, mode);
    base += tcount;
    count -= tcount;
}
#endif

/*
** semsys - System entry point for semctl, semget, and semop system calls.
*/

semsys()
{
    int    semctl(),
          semget(),
          semop();

    static int (*calls[])() = {semctl, semget, semop};
    register struct a {
        uint    id; /* function code id */
    } *uap = (struct a *)u.u_ap;

    if(uap->id > 2) {
        u.u_error = EINVAL;
        return;
    }
    u.u_ap = &u.u_arg[1];
    (*calls[uap->id]) ();
}

/*
** semundo - Undo work done up to finding an operation that can't be done.
*/

semundo(op, n, id, sp)
register struct sembuf *op; /* first operation that was done ptr */
register int n, /* # of operations that were done */
register struct semid_ds *sp; /* semaphore data structure ptr */
{
    register struct sem *semp; /* semaphore ptr */

    for(op += n - 1; n--; op--) {
        if(op->sem_op == 0)
            continue;
        semp = sp->sem_base + op->sem_num;
        semp->semval -= op->sem_op;
        if(op->sem_flg & SEM_UNDO)
            (void) semaop(-op->sem_op, id, (short)op->sem_num);
    }
}

/*
** semunrm - Undo entry remover.
**
** This routine is called to clear all undo entries for a set of semaphores
** that are being removed from the system or are being reset by SETVAL or
** SETVALS commands to semctl.
*/

semunrm(id, low, high)
int id; /* semid */
ushort low, /* lowest semaphore being changed */
high; /* highest semaphore being changed */
{
    register struct sem_undo *pp, /* ptr to predecessor to p */
    *p; /* ptr to current entry */
    register struct undo *up; /* ptr to undo entry */
    register int i, /* loop control */
    j; /* loop control */

    pp = NULL;
    p = semunp;
    while(p != NULL) {

```

```

/* Search through current structure for matching entries. */
for(up = p->un_ent, i = 0; i < p->un_cnt; i++) {
    if(id < up->un_id)
        break;
    if(id > up->un_id || low > up->un_num) {
        up++;
        i++;
        continue;
    }
    if(high < up->un_num)
        break;
    for(j = i; j < p->un_cnt; j++)
        p->un_ent[j - 1] = p->un_ent[j];
    p->un_cnt--;
}

/* Reset pointers for next round. */
if(p->un_cnt == 0)

/* Remove from linked list. */
if(pp == NULL) {
    semunp = p->un_np;
    p->un_np = NULL;
    p = semunp;
} else {
    pp->un_np = p->un_np;
    p->un_np = NULL;
    p = pp->un_np;
}
} else {
    pp = p;
    p = p->un_np;
}
}
}

```

```

/* @(#)shm.c 1.7 */
#include "sys/types.h"
#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/errno.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/seg.h"
#include "sys/ipc.h"
#include "sys/shm.h"
#include "sys/proc.h"
#include "sys/context.h"
#include "sys/system.h"
#include "sys/map.h"
#include "sys/var.h"
#include "sys/scat.h"

extern struct shmid_ds shmем[]; /* shared memory headers */
extern struct shmid_ds *shm_shmem[]; /* ptrs to attached segments */
extern struct shmpt_ds shm_pte[]; /* segment attach points */
extern struct shminfo shminfo; /* shared memory info structure */
int shm_tot; /* total shared memory currently used */

extern time_t time; /* system idea of date */

struct ipc_perm *ipcget();
struct shmid_ds *shmconv();

/*
** shmat - Shmat system call.
*/

shmat()
{
    register struct a {
        int shmid;
        char *addr;
        int flag;
    } *uap = (struct a *)u.u_ap;
    register struct shmid_ds *sp; /* shared memory header ptr */
    register struct user *up;
    register struct proc *p;
    register int shmn;
    register int segbeg;
    register struct phys *ph;
    int i, aa, ix;
    int as,bs,ap,bp;

    up = &u;
    p = up->u_proc;
    if ((sp = shmconv(uap->shmid, SHM_DEST)) == NULL)
        return;
    if (ipcaccess(&sp->shm_perm, SHM_R))
        return;
    if ((uap->flag & SHM_RDONLY) == 0)
        if (ipcaccess(&sp->shm_perm, SHM_W))
            return;
    for(shmn = 0; shmn < shminfo.shmseg; shmn++)
        if (shm_shmem[(p - proc)*shminfo.shmseg+shmn] == NULL)
            break;
    if (shmn >= shminfo.shmseg) {
        up->u_error = EMFILE;
        return;
    }
    if (uap->flag & SHM_RND)
        uap->addr = (char *)((int)uap->addr & ~(SHMLBA - 1));

/*
* Check for page alignment and containment within data space
*/
    if ((int)uap->addr & (SHMLBA - 1) ||
        sp->shm_segsize <= 0 ||
        ((int)uap->addr & 0x80000000) ||

```

```

        (((int)uap->addr != 0) &&
        (((int)uap->addr + ctob(stoc(ctos(btoc(sp->shm_segsize)))) >
        ctob(stoc(ctos(btoc(v.v_uend)-up->u_ssize)))))) {
            up->u_error = EINVAL;
            return;
        }
    }

/*
* An address of 0 places the shared memory into a first fit location
*/
    if (uap->addr == NULL) {
        if (shmn > 0) { /* there was a previous attach */
            /* try the virtual address just beyond the
            last one attached */
            segbeg = (short)(p - proc) * (short)shminfo.shmseg +
                shmn - 1; /* index of last shmem */
            segbeg = shm_pte[segbeg].shm_segbeg +
                ctob(stoc(ctos(btoc(
                    shm_shmem[segbeg]->shm_segsize)))));
        } else { /* this is the 1st attach */
            segbeg = v.v_ustart +
                ctob(stoc(ctos(up->u_tsize))) +
                ctob(stoc(ctos(up->u_dsize))) +
                ctob(stoc(ctos(shminfo.shmbrk))));
        }
        /* need to avoid any phys areas */
        for (ph = &u.u_phys[0]; ph < &u.u_phys[v.v_phys]; ph++) {
            if (ph->u_phsize) {
                as = segbeg;
                bs = segbeg +
                    ctob(stoc(ctos(btoc(sp->shm_segsize)))));
                ap = ph->u_phladdr;
                bp = ph->u_phladdr +
                    ctob(stoc(ctos(ph->u_phsize)));
                if ((as < ap) && (bs <= ap))
                    /* shmat all before phys - ok */
                    continue;
                if ((as >= bp) && (bs > bp))
                    /* shmat all after phys - ok */
                    continue;
                /* allocation would conflict with phys */
                /* choose another location... where? */
                up->u_error = ENOMEM;
                return;
            }
        }
    } else {
/*
* Check to make sure segment does not overlay any valid segments
*/
        segbeg = vtoseg((int)uap->addr);
        for (i = btoc(sp->shm_segsize); i > 0; i -- aa) {
            aa = min(NPAGEPERSEG, (unsigned)i);
            if ((getmmu((short *) (segbeg|ACCLIM)) & PROTMASK) != ASINVAL) {
                up->u_error = ENOMEM;
                return;
            }
            segbeg += (1<<SEGSIZE);
        }
        segbeg = (int)uap->addr;
    }
    if (chksize(up->u_tsize, up->u_dsize, up->u_ssize)) {
        up->u_error = ENOMEM;
        return;
    }
    i = (short)(p - proc) * (short)shminfo.shmseg + shmn;
    shm_shmem[i] = sp;
    shm_pte[i].shm_segbeg = segbeg;
    shm_pte[i].shm_sflg = ((uap->flag & SHM_RDONLY) ? RO : RW);
    shm_pte[i].shm_seg = 0;
    cxrlease(p->p_context); /*
    sureg();

/*
* Clear segment on first attach
*/
    if (sp->shm_perm.mode & SHM_CLEAR) {

```

```

        i = btoc(sp->shm_segsz);
#ifdef NONSCATLOAD
        ix = btoc(segbeg);
        while (--i >= 0)
            clearseg(ix++);
#else
        ix = sp->shm_scat;
        while (--i >= 0) {
            clearseg(ixtoc(ix));
            ix = scatmap[ix].sc_index;
        }
#endif
    sp->shm_perm.mode &= ~SHM_CLEAR;
}
if (p->p_smbeg == 0 || p->p_smbeg > segbeg)
    p->p_smbeg = segbeg;
sp->shm_nattch++;
sp->shm_cnattch++;
up->u_rvall = segbeg;
sp->shm_atime = time;
sp->shm_lpid = p->p_pid;
}

/*
** shmconv - Convert user supplied shmid into a ptr to the associated
** shared memory header.
*/

struct shmid_ds *
shmconv(s, flg)
int s; /* shmid */
int flg; /* error if matching bits are set in mode */
{
    register struct shmid_ds *sp; /* ptr to associated header */

    sp = &shmem[(short)(s % shminfo.shmnm1)];
    if (!(sp->shm_perm.mode & IPC_ALLOC) || sp->shm_perm.mode & flg ||
        s / shminfo.shmnm1 != sp->shm_perm.seq) {
        u.u_error = EINVAL;
        return(NULL);
    }
    return(sp);
}

/*
** shmctl - Shmctl system call.
*/

shmctl()
{
    register struct a {
        int shmid,
            cmd;
        struct shmid_ds *arg;
        *uap = (struct a *)u.u_ap;
    }
    register struct shmid_ds *sp; /* shared memory header ptr */
    struct shmid_ds ds; /* hold area for IPC_SET */
    register struct user *up;

    if ((sp = shmconv(uap->shmid, (uap->cmd == IPC_STAT) ? 0 : SHM_DEST)) ==
        NULL)
        return;
    up = &u;
    up->u_rvall = 0;
    switch(uap->cmd) {

/* Remove shared memory identifier. */
    case IPC_RMID:
        if (up->u_uid != sp->shm_perm.uid && up->u_uid != sp->shm_perm.cuid
            && !suser())
            return;
        sp->shm_ctime = time;
        sp->shm_perm.mode |= SHM_DEST;

/* Change key to private so old key can be reused without
        waiting for last detach. Only allowed accesses to

```

```

        this segment now are shmdt() and shmctl(IPC_STAT).
        All others will give bad shmid. */
    sp->shm_perm.key = IPC_PRIVATE;

/* Adjust counts to counter shmfree decrements. */
    sp->shm_nattch++;
    sp->shm_cnattch++;
    shmfree(sp);
    break;

/* Set ownership and permissions. */
    case IPC_SET:
        if (up->u_uid != sp->shm_perm.uid && up->u_uid != sp->shm_perm.cuid
            && !suser())
            return;
        if (copyin((caddr_t)uap->arg, (caddr_t)&ds, sizeof(ds))) {
            up->u_error = EFAULT;
            return;
        }
        sp->shm_perm.uid = ds.shm_perm.uid;
        sp->shm_perm.gid = ds.shm_perm.gid;
        sp->shm_perm.mode = (ds.shm_perm.mode & 0777) |
            (sp->shm_perm.mode & ~0777);
        sp->shm_ctime = time;
        break;

/* Get shared memory data structure. */
    case IPC_STAT:
        if (ipcaccess(&sp->shm_perm, SHM_R))
            return;
        if (copyout((caddr_t)sp, (caddr_t)uap->arg, sizeof(*sp)))
            up->u_error = EFAULT;
        break;

    default:
        up->u_error = EINVAL;
        break;
    }
}

/*
** shmdt - Shmdt system call.
*/

shmdt()
{
    struct a {
        char *addr;
    }
    *uap = (struct a *)u.u_ap;
    register struct shmid_ds *sp, **spp;
    int segbeg;
    register struct shmpt_ds *pt;
    register i, j;
    register struct proc *p;

/*
** Check for page alignment
*/
    if ((int)uap->addr & (ctob(1) - 1) ||
        (segbeg = (int)uap->addr) == 0) {
        u.u_error = EINVAL;
        return;
    }

/*
** find segment
*/
    spp = &shm_shmem[i=(p-u.u_procp)-proc]*shminfo.shmseg;
    pt = &shm_pte[i];
    for (i=0; i < shminfo.shmseg; i++, pt++, spp++) {
        if ((*spp) != NULL && pt->shm_segbeg == segbeg)
            break;
        sp = NULL;
    }
    if (sp == NULL) {
        u.u_error = EINVAL;
        return;
    }

```

```

}
shmfree(sp);
sp->shm_dtime = time;
sp->shm_lpid = p->p_pid;
*spp = NULL;
pt->shm_segbeg = 0;
p->p_smbeg = 0;
pt = &shm_pte[(p-proc)*shminfo.shmseg];
for (j=0; j<shminfo.shmseg; j++, pt++) {
    if (i = pt->shm_segbeg) {
        if (p->p_smbeg) {
            if (p->p_smbeg > i)
                p->p_smbeg = i;
        } else {
            p->p_smbeg = i;
        }
    }
}
/*
cxrelse(p->p_context); */
sureg();
u.u_rvall = 0;
}

/*
** shmexec - Called by setregs(os/sys1.c) to handle shared memory exec
** processing.
*/

shmexec()
{
    register struct shmid_ds    **spp; /* ptr to ptr to header */
    register struct shmpt_ds    *sppp; /* ptr to pte data */
    register int                i;      /* loop control */

    if (u.u_procp->p_smbeg == 0)
        return;
    /* Detach any attached segments. */
    sPPP = &shm_pte[i = (u.u_procp - proc)*shminfo.shmseg];
    u.u_procp->p_smbeg = 0;
    spp = &shm_shmem[i];
    for(i = 0; i < shminfo.shmseg; i++, spp++, sPPP++) {
        if (*spp == NULL)
            continue;
        shmfree(*spp);
        *spp = NULL;
        sPPP->shm_segbeg = 0;
    }
}

/*
** shmexit - Called by exit(os/sys1.c) to clean up on process exit.
*/

shmexit()
{
    /* Same processing as for exec. */
    shmexec();
}

/*
** shmfork - Called by newproc(os/slp.c) to handle shared memory fork
** processing.
*/

shmfork(cp, pp)
struct proc    *cp, /* ptr to child proc table entry */
              *pp; /* ptr to parent proc table entry */
{
    register struct shmid_ds    **cpp, /* ptr to child shm mem ptrs */
    **ppp; /* ptr to parent shm mem ptrs */
    register struct shmpt_ds    *cPPP,
    *pPPP;
    register int                i;      /* loop control */

    if (pp->p_smbeg == 0)
        return;
    /* Copy ptrs and update counts on any attached segments. */
    cpp = &shm_shmem[(cp - proc)*shminfo.shmseg];
    ppp = &shm_shmem[(pp - proc)*shminfo.shmseg];
    cPPP = &shm_pte[(cp - proc)*shminfo.shmseg];
    pPPP = &shm_pte[(pp - proc)*shminfo.shmseg];
    cp->p_smbeg = pp->p_smbeg;
    for(i = 0; i < shminfo.shmseg; i++, cpp++, ppp++, cPPP++, pPPP++) {
        if (*cpp == *ppp) {
            (*cpp)->shm_nattach++;
            (*cpp)->shm_cnattach++;
            cPPP->shm_segbeg = pPPP->shm_segbeg;
            cPPP->shm_sflg = pPPP->shm_sflg;
            cPPP->shm_seg = 0;
        }
    }
}

/*
** shmfree - Decrement counts. Free segment and page tables if
** indicated.
*/

shmfree(sp)
register struct shmid_ds    *sp; /* shared memory header ptr */
{
    register int size;

    if (sp == NULL)
        return;
    sp->shm_nattach--;
    if (--(sp->shm_cnattach) == 0 && sp->shm_perm.mode & SHM_DEST) {
        size = btoc(sp->shm_segsz);
#ifdef NONSCATLOAD
        mfree(coremap, size, (int)sp->shm_scat);
#else
        memfree((int)sp->shm_scat);
#endif
        /* adjust maxmem for amount freed */
        maxmem -= size;
        shmtot -= size;

        sp->shm_perm.mode = 0;
        if (((int)(++(sp->shm_perm.seq)*shminfo.shmmni + (sp - shm mem)) < 0)
            sp->shm_perm.seq = 0;
        }
}

/*
** shmget - Shmget system call.
*/

shmget()
{
    register struct a {
        key_t    key;
        int      size,
        shmflg;
    } *uap = (struct a *)u.u_ap;
    register struct shmid_ds    *sp; /* shared memory header ptr */
    int                          s; /* ipcget status */
    register int                size;

    /*if (uap->size == 0) /* Bug #675 ... Paul */
    /* return; */
    if ((sp = (struct shmid_ds *)ipcget(uap->key, uap->shmflg,
        (struct ipc_perm *)shm mem, shminfo.shmmni, sizeof(*sp), &s)) == NULL)
        return;
    if (s) {
        /* This is a new shared memory segment. Allocate memory and
        finish initialization. */
        if (uap->size < shminfo.shmmni || uap->size > shminfo.shmmax) {
            u.u_error = EINVAL;
            sp->shm_perm.mode = 0;
            return;
        }
        size = btoc(uap->size);
    }
}

```

```

    if (shmtot + size > shminfo.shmall) {
        u.u_error = ENOMEM;
        sp->shm_perm.mode = 0;
        return;
    }
    sp->shm_segsz = uap->size;
#ifdef NONSCATLOAD
    if ((sp->shm_scat = malloc(coremap, size)) == 0) {
        u.u_error = ENOMEM;
        sp->shm_perm.mode = 0;
        return;
    }
#else
    if ((sp->shm_scat = memalloc(size)) == 0) {
        u.u_error = ENOMEM;
        sp->shm_perm.mode = 0;
        return;
    }
#endif

    /* adjust maxmem for the segment */
    maxmem -= size;
    shmtot += size;

    sp->shm_perm.mode |= SHM_CLEAR;
    sp->shm_nattch = 0;
    sp->shm_cnattch = 0;
    sp->shm_atime = 0;
    sp->shm_dtime = 0;
    sp->shm_ctime = time;
    sp->shm_lpid = 0;
    sp->shm_cpuid = u.u_procp->p_pid;
} else
    if (uap->size && uap->size > sp->shm_segsz) {
        u.u_error = EINVAL;
        return;
    }
u.u_rvall = sp->shm_perm.seq * shminfo.shmni + (sp - shmем);
}

/*
** shmsys - System entry point for shmat, shmctl, shmdt, and shmget
** system calls.
*/

shmsys()
{
    register struct a {
        uint id;
    } *uap = (struct a *)u.u_ap;
    int
        shmat(),
        shmctl(),
        shmdt(),
        shmget();

    static int (*calls[])() = {shmat, shmctl, shmdt, shmget};

    if (uap->id > 3) {
        u.u_error = EINVAL;
        return;
    }
    u.u_ap = &u.u_arg[1];
    (*calls[uap->id])();
}

#ifdef notdef
shmreset(p, ub, p0br, p0lr)
struct proc *p;
struct user *ub;
int *p0br, p0lr;
{
    register struct shm_id_ds **sp;
    register struct shmpt_ds *pt;
    register i, j;
    register int *seg, shm, *pte;

```

```

    /*
    if (p->p_smbeg == 0)
        return;
    /*
    * clear unused pte's
    */
    seg = p0br + ub->u_tsize + ub->u_dsize;
    for (i = ub->u_tsize + ub->u_dsize; i < p0lr; i++)
        *seg++ = 0;
    /*
    * move in the shared memory segments
    */
    sp = &shm_shmem[i = (p - proc)*shminfo.shmseg];
    pt = &shm_pte[i];
    for (i = 0; i < shminfo.shmseg; i++, sp++, pt++) {
        if (shm = pt->shm_segbeg) {
            seg = p0br + shm;
            pte = (int *)((*sp)->shm_ptbl);
            for (j = 0; j < btoc((*sp)->shm_segsz); j++)
                *seg++ = *pte++ | PG_V | pt->shm_sflg;
        }
    }
}
#endif

#ifdef notdef
dumppte(p0br, p0lr, p1lr, p1br)
int *p0br, p0lr, p1lr, *p1br;
{
    register i;

    printf("tsize %d, dsize %d\n", u.u_tsize, u.u_dsize);
    printf("p0br %x p1br %d\n", p1br %x p1lr %d\n", p0br, p0lr, p1br, p1lr);
    for (i=0; i<p0lr; i++) {
        if ((i%8) == 0)
            printf("\n");
        printf("%x ", *p0br++);
    }
    printf("\n\n");
}
#endif

```

```

/*
* do only if there is shared memory attached

```

```

/* @(#)sig.c 1.3 */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/proc.h"
#include "sys/inode.h"
#include "sys/file.h"
#include "sys/reg.h"
#include "sys/text.h"
#include "sys/seg.h"
#include "sys/var.h"
#include "sys/psl.h"
#include "sys/scat.h"

/*
 * Priority for tracing
 */
#define IPCPRI PZERO

/*
 * Tracing variables.
 * Used to pass trace command from
 * parent to child being traced.
 * This data base cannot be
 * shared and is locked
 * per user.
 */
struct ipctrace
{
    int    ip_data;
    int    ip_lock;
    int    ip_req;
    int    *ip_addr;
} ipc;

/*
 * Send the specified signal to
 * all processes with 'pgrp' as
 * process group.
 * Called by tty.c for quits and
 * interrupts.
 */
signal(pgrp, sig)
register pgrp;
{
    register struct proc *p;

    if (pgrp == 0)
        return;
    for (p = &proc[1]; p < (struct proc *)v.ve_proc; p++)
        if (p->p_pgrp == pgrp)
            psignal(p, sig);
}

/*
 * Send the specified signal to
 * the specified process.
 */
psignal(p, sig)
register struct proc *p;
register sig;
{
    sig--;
    if (sig < 0 || sig >= NSIG)
        return;
    if (((p->p_sigign >> sig) & 1) && sig != (SIGCLD - 1))
        return;
    p->p_sig := 1L<<sig;

```

```

    if (p->p_stat == SSLEEP && p->p_pri > PZERO) {
        if (p->p_pri > PUSER)
            p->p_pri = PUSER;
        setrun(p);
    }
}

/*
 * Returns true if the current
 * process has a signal to process.
 * This is asked at least once
 * each time a process enters the
 * system.
 * A signal does not do anything
 * directly to a process; it sets
 * a flag that asks the process to
 * do something to itself.
 */
issig()
{
    register n;
    register struct proc *p, *q;

    p = u.u_procp;
    while (p->p_sig) {
        n = fsig(p);
        if (n == SIGCLD) {
            if (u.u_signal[SIGCLD-1]&01) {
                for (q = &proc[1];
                    q < (struct proc *)v.ve_proc; q++)
                    if (p->p_pid == q->p_ppid &&
                        q->p_stat == SZOMB)
                        freeproc(q, 0);
            } else if (u.u_signal[SIGCLD-1])
                return(n);
        } else if (n == SIGPWR) {
            if (u.u_signal[SIGPWR-1] && (u.u_signal[SIGPWR-1]&1)==0)
                return(n);
        } else if ((u.u_signal[n-1]&1) == 0 || (p->p_flag&STRC))
            return(n);
        p->p_sig &= ~(1L<<(n-1));
    }
    return(0);
}

/*
 * Enter the tracing STOP state.
 * In this state, the parent is
 * informed and the process is able to
 * receive commands from the parent.
 */
stop()
{
    register struct proc *pp, *cp;

loop:
    cp = u.u_procp;
    if (cp->p_ppid != 1)
        for (pp = &proc[0]; pp < (struct proc *)v.ve_proc; pp++)
            if (pp->p_pid == cp->p_ppid) {
                wakeup((caddr_t)pp);
                cp->p_stat = SSTOP;
                swtch();
                if ((cp->p_flag&STRC)==0 || procxmt())
                    return;
                goto loop;
            }
    exit(fsig(u.u_procp));
}

/*
 * Perform the action specified by
 * the current signal.
 * The usual sequence is:
 *     if (issig())
 *         psig();

```

```

*/
psig()
{
    register n, p;
    register struct proc *rp;
#ifdef mc68881 /* MC68881 floating-point coprocessor */
    extern short fp881; /* is there an MC68881? */
#endif mc68881

    rp = u.u_procp;
#ifdef FLOAT /* sky floating point board */
    if (u.u_fpinuse && u.u_fpsaved==0) {
        savfp();
        u.u_fpsaved = 1;
    }
#endif
#ifdef mc68881 /* MC68881 floating-point coprocessor */
    if (fp881)
        fpsave();
#endif mc68881
    if (rp->p_flag&STRC)
        stop();
    n = fsig(rp);
    if (n==0)
        return;
    rp->p_sig &= ~(1L<<(n-1));
    if ((p=u.u_signal[n-1]) != 0) {
        if (p & 1)
            return;
        u.u_error = 0;
        if (n != SIGILL && n != SIGTRAP)
            u.u_signal[n-1] = 0;
        sendsig((caddr_t)p, n);
        return;
    }
    switch(n) {

    case SIGQUIT:
    case SIGILL:
    case SIGTRAP:
    case SIGIOT:
    case SIGEMT:
    case SIGFPE:
    case SIGBUS:
    case SIGSEGV:
    case SIGSYS:
        if (core())
            n += 0200;
    }
    exit(n);
}

/*
 * find the signal in bit-position
 * representation in p_sig.
 */
fsig(p)
struct proc *p;
{
    register short i;
    register long n;

    n = p->p_sig;
    i = NSIG - 1;
    do {
        if (n & 1L)
            return(NSIG - i);
        n >>= 1;
    } while (--i != -1);
    return(0);
}

/*
 * Create a core image on the file "core"
 *
 * It writes USIZE (v.v_usize) block of the

```

```

 * user.h area followed by the entire
 * data+stack segments.
 */
core()
{
    register struct inode *ip;
    register struct user *up;
    register s;
    extern schar();

    up = &u;
    if (up->u_uid != up->u_ruid)
        return(0);
    up->u_error = 0;
    up->u_dirp = "core";
    ip = namei(schar, 1);
    if (ip == NULL) {
        if (up->u_error)
            return(0);
        ip = maknode(0666);
        if (ip==NULL)
            return(0);
    }
    if (!access(ip, IWRITE) && (ip->i_mode&IFMT) == IFREG) {
        itrunc(ip);
        up->u_offset = 0;
        up->u_base = (caddr_t)up;
        up->u_count = ctob(v.v_usize);
        up->u_segflg = 1;
        up->u_limit = (daddr_t)ctod(MAXMEM);
        up->u_fmode = FWRITE;
        /* make register pointer relative for adb */
        up->u_ar0 = (int *)((int)up->u_ar0 - (int)up);
        up->u_usrtop = btoc(v.v_uend);
        writei(ip);
        up->u_ar0 = (int *)((int)up->u_ar0 + (int)up);
        s = up->u_procp->p_size - v.v_usize;
        (void) estabur((unsigned)0, (unsigned)s, (unsigned)0, 0, RO);
        up->u_base = (caddr_t)v.v_ustart;
        up->u_count = ctob(s);
        up->u_segflg = 0;
        writei(ip);
    } else
        up->u_error = EACCES;
    input(ip);
    return(up->u_error==0);
}

/*
 * grow the stack to include the SP
 * true return if successful.
 */
#ifdef NONSCATLOAD
grow(sp)
unsigned sp;
{
    register struct user *up;
    register struct proc *p;
    register si, i, al;

    up = &u;
    if ((v.v_uend-sp) < ctob(up->u_ssize))
        return(0);
    si = btoc(v.v_uend-sp) - up->u_ssize + SINCR;
    if (si <= 0)
        return(0);
    if (estabur(up->u_tsize, up->u_dsize, up->u_ssize+si, 0, RO))
        return(0);
    p = up->u_procp;
    expand((int)(p->p_size+si));
    al = p->p_addr + p->p_size;
    for(i=up->u_ssize; i; i--) {
        al--;
        copyseg(al-si, al);
    }
    for(i=si; i; i--)

```

```

        clearseg(--al);
        up->u_ssize += si;
        return(1);
    }
    #else
    grow(sp)
    unsigned sp;
    {
        register struct user *up;
        register struct proc *p;
        register si, i, al;
        register struct scatter *s;
        register a2, n;
        short t;

        up = &u;
        if ((v.v_uend-sp) < ctob(up->u_ssize))
            return(0);
        si = btoc(v.v_uend-sp) - up->u_ssize + SINCR;
        if (si <= 0)
            return(0);
        if (chksize(up->u_tsize, up->u_dsize, up->u_ssize+si))
            return(0);
        p = up->u_procp;
        expand((int) (p->p_size+si));
        /*
         * locate last click of old data size
         */
        s = scatmap;
        al = p->p_scat;
        n = v.v_usize + up->u_dsize;
        for (i=1; i<n; i++)
            al = s[al].sc_index;
        /*
         * locate last click of old stack
         */
        a2 = s[al].sc_index;
        n = up->u_ssize;
        if (n == 0)
            printf("grow:ssize not expected to be zero\n");
        for (i=1; i<n; i++)
            a2 = s[a2].sc_index;
        /*
         * chain new clicks into stack space following data space
         */
        t = s[al].sc_index;
        s[al].sc_index = s[a2].sc_index;
        n = si;
        for (i=0; i<n; i++)
            clearseg(ixtoc(al = s[al].sc_index));
        if (s[al].sc_index != SCATEND)
            printf("grow failure\n");
        s[al].sc_index = t;
        s[a2].sc_index = SCATEND;
        p->p_flag ^= ~SCONTIG;
        up->u_ssize += si;
        (void) estabur(up->u_tsize, up->u_dsize, up->u_ssize, 0, RO);
        return(1);
    }
    #endif

    /*
     * sys-trace system call.
     */
    ptrace()
    {
        register struct ipctrace *ipcp;
        register struct user *up;
        register struct proc *p;
        register struct a {
            int    req;
            int    pid;
            int    *addr;
            int    data;
        } *uap;

```

```

        up = &u;
        uap = (struct a *)up->u_ap;
        if (uap->req <= 0) {
            up->u_procp->p_flag |= STRC;
            return;
        }
        for (p=proc; p < (struct proc *)v.ve_proc; p++)
            if (p->p_stat==SSTOP
                && p->p_pid==uap->pid
                && p->p_ppid==up->u_procp->p_pid)
                goto found;
        up->u_error = ESRCH;
        return;

    found:
        ipcp = &ipc;
        while (ipcp->ip_lock)
            (void) sleep((caddr_t)ipcp, IPCPRI);
        ipcp->ip_lock = p->p_pid;
        ipcp->ip_data = uap->data;
        ipcp->ip_addr = uap->addr;
        ipcp->ip_req = uap->req;
        p->p_flag ^= ~SWTCD;
        setrun(p);
        while (ipcp->ip_req > 0)
            (void) sleep((caddr_t)ipcp, IPCPRI);
        up->u_rvall = ipcp->ip_data;
        if (ipcp->ip_req < 0)
            up->u_error = EIO;
        ipcp->ip_lock = 0;
        wakeup((caddr_t)ipcp);
    }

    /*
     * Code that the child process
     * executes to implement the command
     * of the parent process in tracing.
     */
    procxmt()
    {
        register struct ipctrace *ipcp;
        register struct user *up;
        register int i;
        register *p;
        register struct text *xp;

        up = &u;
        ipcp = &ipc;
        if (ipcp->ip_lock != up->u_procp->p_pid)
            return(0);
        i = ipcp->ip_req;
        ipcp->ip_req = 0;
        wakeup((caddr_t)ipcp);
        switch (i) {

            /* read user I */
            case 1:
                ipcp->ip_data = fuword((caddr_t)ipcp->ip_addr);
                break;

            /* read user D */
            case 2:
                ipcp->ip_data = fuword((caddr_t)ipcp->ip_addr);
                break;

            /* read u */
            case 3:
                i = (int)ipcp->ip_addr;
                if (i<0 || i >= ctob(v.v_usize))
                    goto error;
                ipcp->ip_data = *((int *) ((char *)up + (i & ~1)));
                break;

            /* write user I */
            /* Must set up to allow writing */
            case 4:

```



```

/*
 * If text, must assure exclusive use
 */
if (xp = up->u_procp->p_textp) {
    if (xp->x_count!=1 || xp->x_iptr->i_mode&ISVTX)
        goto error;
    xp->x_iptr->i_flag &= ~ITEXT;
}
(void) estabur(up->u_tsize, up->u_dsize, up->u_ssize, 0, RW);
i = suword((caddr_t)ipcp->ip_addr, 0);
(void) suword((caddr_t)ipcp->ip_addr, ipcp->ip_data);
(void) estabur(up->u_tsize, up->u_dsize, up->u_ssize, 0, RO);
if (i<0)
    goto error;
if (xp)
    xp->x_flag |= XWRIT;
break;

/* write user D */
case 5:
    if (suword((caddr_t)ipcp->ip_addr, 0) < 0)
        goto error;
    (void) suword((caddr_t)ipcp->ip_addr, ipcp->ip_data);
    break;

/* write u */
case 6:
    i = (int)ipcp->ip_addr;
    p = (int *)((char *)up + (i & ~1));
    for (i=0; i<17; i++)
        if (p == &up->u_ar0[regloc[i]])
            goto ok;
    if (p == &up->u_ar0[RPS]) {
        /* assure user space and priority 0 */
        ipcp->ip_data &= ~0x2700;
        goto ok;
    }
    goto error;

ok:
    *p = ipcp->ip_data;
    break;

/* set signal and continue */
/* one version causes a trace-trap */
case 9:
    up->u_ar0[RPS] |= PS_T;

case 7:
    if ((int)ipcp->ip_addr != 1)
        up->u_ar0[PC] = (int)ipcp->ip_addr;
    up->u_procp->p_sig = 0L;
    if (ipcp->ip_data)
        psignal(up->u_procp, ipcp->ip_data);
    return(1);

/* force exit */
case 8:
    exit(fsig(up->u_procp));

/* read u registers */
case 10:
    if ((i = (int)ipcp->ip_addr) < 0 || i > 17)
        goto error;
    if (i == 17)
        ipcp->ip_data = up->u_ar0[regloc[17]] & 0xFFFF;
    else
        ipcp->ip_data = up->u_ar0[regloc[i]];
    break;

/* write u registers */
case 11:
    if ((i = (int)ipcp->ip_addr) < 0 || i > 17)
        goto error;
    if (i == 17) {
        ipcp->ip_data &= ~0x2700;        /* user only */

```

```

        up->u_ar0[regloc[17]] =
            (up->u_ar0[regloc[17]] & ~0xFFFF) |
            (ipcp->ip_data & 0xFFFF);
    } else
        up->u_ar0[regloc[i]] = ipcp->ip_data;
    break;

default:
error:
    ipcp->ip_req = -1;
}
return(0);
}

```

```

/*
 * Copyright 1982 UniSoft Corporation
 *
 * Speaker Driver
 * Used to operate the lisa speaker.
 */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/file.h"
#include "sys/tty.h"
#include "sys/termio.h"
#include "sys/conf.h"
#include "sys/sysinfo.h"
#include "sys/var.h"
#include "sys/reg.h"
#include "sys/callo.h"
#include "sys/ttold.h"
#include "setjmp.h"
#include "sys/mmu.h"
#include "sys/cops.h"
#include "sys/local.h"
#include "sys/speaker.h"
#include "sys/l2.h"

typedef unsigned long u_long;
u_long sktrap;          /* flag and slot to calculate spkr delay */

int sk_open;           /* active flag */

#define SKPRI (PZERO+8)

skopen(dev, flag)
dev_t dev;
{
    if (dev != 0) {      /* minor device number is wrong */
        u.u_error = ENXIO;
        return;
    }
    if (flag == 1) {     /* open for reading ?? */
        u.u_error = EINVAL;
        return;
    }
    if (sk_open++ > 0) { /* already opened */
        u.u_error = EBUSY;
        return;
    }
}

/* ARGSUSED */
skclose(dev, flag)
{
    if (sk_open <= 0)
        u.u_error = EINVAL;
    else
        sk_open = 0;
}

/* ARGSUSED */
skwrite(dev)
{
    struct speaker spkr;

    while (u.u_count >= sizeof(spkr)) {
        if (copyin(u.u_base, (caddr_t)&spkr, sizeof(spkr))) {
            u.u_error = EFAULT;
            return;
        }
        u.u_base += sizeof(spkr);
        u.u_count -= sizeof(spkr);
        SPL2();
    }
}

while (sktrap)
    (void) sleep((caddr_t)&sktrap, SKPRI);
COPSADDR->e_irk = (COPSADDR->e_irk & 0xF1) | ((spkr.sk_volume&7) << 1);
sksound(spkr.sk_wavlen&MAXWLEN, spkr.sk_duration);
SPL0();
}

/* Produce a sound on the speaker at wavelength w microseconds
 * for duration d clock ticks
 */
sksound(w, d)
register unsigned w, d;
{
    extern int skquiet();

    w &= MAXWLEN;          /* max wavelength */
    if (w < MINWLEN)
        w = MINWLEN;
    sktrap = d;            /* call skquiet at now + d */
    if (sktrap <= 0) sktrap = 1; /* in case < MILLIRATE */
    timeout(skquiet, (caddr_t)0, (int)sktrap);
    sktone (w);
}

/* Start a tone at wavelength w microseconds. Sound is produced by rapidly
 * turning on and off the speaker. The 6522 cops chip has an output to the
 * speaker connected to a shift register. A built in timer controls the rate
 * at which the shift register bits are output to the speaker.
 */
sktone(w)
register unsigned w;
{
    register struct device_e *p = COPSADDR;
    register int cmd = 0x55;          /* sk shift reg */

    w = w >> 3;          /* wavelength resolution is 8 microseconds */
    if (w > 0xFF) {
        cmd = 0x33;
        w = w >> 1;
        if (w > 0xFF) {
            cmd = 0xF;
            w = w >> 1;
        }
    }

    p->e_acr |= 0x10;      /* enable */
    p->e_t2cl = w;        /* set timer */
    p->e_sr = cmd;        /* set output pattern */
}

skquiet()
{
    struct device_e *p = COPSADDR;
    p->e_acr &= 0xE3;
    sktrap = 0;
    wakeup((caddr_t)&sktrap);
}

beep()
{
    int sp;
    if (sktrap) return;
    sp = spl2();
    COPSADDR->e_irk = (COPSADDR->e_irk & 0xF1) | (12_bvol << 1);
    sksound((unsigned)12_bpitch, (unsigned)12_btime);
    splx(sp);
}

#ifdef notdef
/*
 * Produce a short click to implement keyboard clicking
 */
click()
{
    register struct device_e *p = COPSADDR;

```

```
int i = 20;
int sp;

if (sktrap) return;
sp = spl2();
p->e_acr |= 0x10;
p->e_t2c1 = 0x10;
p->e_sr = 0x55;
while (--i != -1) ;
p->e_acr &= 0xE3;
splx(sp);
}
#endif
```

```

/* @(#)slp.c 1.8 */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/proc.h"
#include "sys/context.h"
#include "sys/text.h"
#include "sys/system.h"
#include "sys/sysinfo.h"
#include "sys/map.h"
#include "sys/file.h"
#include "sys/inode.h"
#include "sys/buf.h"
#include "sys/var.h"
#include "sys/ipc.h"
#include "sys/shm.h"
#include "sys/errno.h"
#include "sys/scat.h"

typedef int mem_t;

#define NHSQUE 64 /* must be power of 2 */
#define sqhash(X) ((int)(X) >> 3) & (NHSQUE-1))
struct proc *hsque[NHSQUE];
char runin, runout, runrun, curpri;
struct proc *curproc, *runq;

/*
 * sleep according to a cpu adjusted priority
 */
asleep(chan, pri)
caddr_t chan;
{
    return(sleep(chan, pri + ((u.u_procp->p_cpu&0xFF) >> 5)));
}

/*
 * Give up the processor till a wakeup occurs
 * on chan, at which time the process
 * enters the scheduling queue at priority pri.
 * The most important effect of pri is that when
 * pri<=PZERO a signal cannot disturb the sleep;
 * if pri>PZERO signals will be processed.
 * Callers of this routine must be prepared for
 * premature return, and check that the reason for
 * sleeping has gone away.
 */
#define TZERO 10
sleep(chan, disp)
caddr_t chan;
{
    register struct proc *rp = u.u_procp;
    register struct proc **q = sqhash(chan);
    register s;

    s = splhi();
    if (panicstr) {
        SPL0();
        splx(s);
        return(0);
    }
    rp->p_stat = SSLEEP;
    rp->p_wchan = chan;
    rp->p_link = *q;
    *q = rp;
    if (rp->p_time > TZERO)
        rp->p_time = TZERO;
    if ((rp->p_pri = (disp&PMASK)) > PZERO) {
        if (rp->p_sig && issig()) {
            rp->p_wchan = 0;
            rp->p_stat = SRUN;

```

```

s = splhi();
for(q=runq; q!=NULL; q=q->p_link)
    if (q == p) {
        printf("proc on q\n");
        goto out;
    }
p->p_link = runq;
runq = p;
out:
    splx(s);
}
/*
 * Set the process running;
 * arrange for it to be swapped in if necessary.
 */
setrun(p)
register struct proc *p;
{
    register struct proc **q;
    register s;

    s = splhi();
    if (p->p_stat == SSLEEP) {
        /* take off sleep queue */
        for (q = sqhash(p->p_wchan); *q != p; q = &(*q)->p_link);
        *q = p->p_link;
        p->p_wchan = 0;
    } else if (p->p_stat == SRUN) {
        /* already on run queue - just return */
        splx(s);
        return;
    }
    /* put on run queue */
    p->p_stat = SRUN;
    p->p_link = runq;
    runq = p;
    if (!(p->p_flag&SLOAD)) {
        p->p_time = 0;
        if (runout > 0) {
            runout = 0;
            setrun(&proc[0]);
        }
    } else if (p->p_pri < curpri)
        runrun++;
    splx(s);
}
/*
 * The main loop of the scheduling (swapping) process.
 * The basic idea is:
 * see if anyone wants to be swapped in;
 * swap out processes until there is room;
 * swap him in;
 * repeat.
 * The runout flag is set whenever someone is swapped out.
 * Sched sleeps on it awaiting work.
 *
 * Sched sleeps on runin whenever it cannot find enough
 * memory (by swapping out or otherwise) to fit the
 * selected swapped process. It is awakened when the
 * memory situation changes and in any case once per second.
 */
sched()
{
    register struct proc *rp, *p;
    register outage, inage;
    int maxbad;
    int tmp;

    /*
     * find user to swap in;
     * of users ready, select one out longest
     */

```

```

loop:
    SPL6();
    outage = -20000;
#ifdef NONSCATLOAD
    for (rp = &proc[0]; rp < (struct proc *)v.ve_proc; rp++)
        if (rp->p_stat==SRUN && (rp->p_flag&SLOAD) == 0 &&
            rp->p_time > outage) {
                p = rp;
                outage = rp->p_time;
            }
#else
    for (rp = &proc[0]; rp < (struct proc *)v.ve_proc; rp++)
        if ((rp->p_flag&(SSWAPIT|SLOAD)) == (SSWAPIT|SLOAD)) {
                p = rp;
                SPL0();
                goto swapit;
            } else if (rp->p_stat==SRUN && (rp->p_flag&SLOAD) == 0 &&
                rp->p_time > outage) {
                p = rp;
                outage = rp->p_time;
            }
#endif
    /*
     * If there is no one there, wait.
     */
    if (outage == -20000) {
        runout++;
        (void) sleep((caddr_t)&runout, PSWP);
        goto loop;
    }
    SPL0();

    /*
     * See if there is memory for that process;
     * if so, swap it in.
     */
    if (swapin(p))
        goto loop;

    /*
     * none found.
     * look around for memory.
     * Select the largest of those sleeping
     * at bad priority; if none, select the oldest.
     */
    SPL6();
    p = NULL;
    maxbad = 0;
    inage = 0;
    for (rp = &proc[0]; rp < (struct proc *)v.ve_proc; rp++) {
        if (rp->p_stat==SZOMB ||
            (rp->p_flag&(SSYS|SLOCK|SLOAD))!=SLOAD)
            continue;
        if (rp->p_textp && rp->p_textp->x_flag&XLOCK)
            continue;
        if (rp->p_stat==SSLEEP || rp->p_stat==SSTOP) {
            tmp = rp->p_pri - PZERO + rp->p_time;
            if (maxbad < tmp) {
                p = rp;
                maxbad = tmp;
            }
        } else if (maxbad<=0 && rp->p_stat==SRUN) {
            tmp = rp->p_time + rp->p_nice - NZERO;
            if (tmp > inage) {
                p = rp;
                inage = tmp;
            }
        }
    }
    SPL0();
    /*
     * Swap found user out if sleeping at bad pri,
     * or if he has spent at least 2 seconds in memory and
     * the swapped-out process has spent at least 2 seconds out.
     */

```

```

    * Otherwise wait a bit and try again.
    */
    if (maxbad>0 || (outage>=2 && inage>=2)) {
#ifdef NONSCATLOAD
        swapit:
#endif
        p->p_flag &= ~SLOAD;
        xswap(p, 1, 0);
        goto loop;
    }
    SPL6();
    runin++;
    (void) sleep((caddr_t)&runin, PSWP);
    goto loop;
}

/*
 * Swap a process in.
 */
#ifdef NONSCATLOAD
swapin(p)
register struct proc *p;
{
    register struct text *xp;
    register int a, x;
    int ta;

    if ((a = malloc(coremap, p->p_size)) == NULL)
        return(0);
    if (xp = p->p_textp) {
        xlock(xp);
        if (!xmlink(xp) && xp->x_ccount==0) {
            if ((x = malloc(coremap, xp->x_size)) == NULL) {
                mfree(coremap, p->p_size, a);
                if ((x = malloc(coremap, xp->x_size)) == NULL) {
                    xunlock(xp);
                    return(0);
                }
            }
            if ((a = malloc(coremap, p->p_size)) == NULL) {
                mfree(coremap, xp->x_size, x);
                xunlock(xp);
                return(0);
            }
        }
        xp->x_caddr = x;
        if ((xp->x_flag&XLOAD)==0)
            swap(xp->x_daddr, x, xp->x_size, B_READ);
    }
    xp->x_ccount++;
    xunlock(xp);
}
if (p->p_xaddr[0]) {
    ta = a;
    for (x=0; x < NSCATSWAP; x++) {
        if (p->p_xaddr[x] == 0)
            continue;
        swap(p->p_xaddr[x], a, p->p_xsize[x], B_READ);
        mfree(swapmap, ctod(p->p_xsize[x]), (int)p->p_xaddr[x]);
        a += p->p_xsize[x];
        p->p_xaddr[x] = 0;
    }
    p->p_addr = ta;
} else {
    swap((daddr_t)p->p_dkaddr, a, p->p_size, B_READ);
    mfree(swapmap, ctod(p->p_size), (int)p->p_dkaddr);
    p->p_addr = a;
}
cxrelse(p->p_context);
p->p_flag |= SLOAD;
p->p_time = 0;
return(1);
}
#else
swapin(p)
register struct proc *p;
{

```

```

    register struct text *xp;
    register int a, x;
    int ta;

    if (p->p_flag&SCONTIG) {
        if ((a = cmalloc(p->p_size)) == NULL)
            return(0);
    } else if ((a = memalloc(p->p_size)) == NULL)
        return(0);
    if (xp = p->p_textp) {
        xlock(xp);
        if (!xmlink(xp) && xp->x_ccount==0) {
            if ((x = memalloc(xp->x_size)) == NULL) {
                memfree(a);
                xunlock(xp);
                return(0);
            }
        }
        xp->x_scatter = x;
        if ((xp->x_flag&XLOAD)==0)
            (void) swap(xp->x_daddr, x, xp->x_size, B_READ);
    }
    xp->x_ccount++;
    xunlock(xp);
}
p->p_flag |= SNOMMU; /* swapping in, do not set mmu registers */
if (p->p_xaddr[0]) {
    ta = a;
    for (x=0; x < NSCATSWAP; x++) {
        if (p->p_xaddr[x] == 0)
            continue;
        a = swap(p->p_xaddr[x], a, p->p_xsize[x], B_READ);
        mfree(swapmap, ctod(p->p_xsize[x]), (int)p->p_xaddr[x]);
        p->p_xaddr[x] = 0;
    }
    p->p_scatter = ta;
} else {
    (void) swap((daddr_t)p->p_dkaddr, a, p->p_size, B_READ);
    mfree(swapmap, ctod(p->p_size), (int)p->p_dkaddr);
    p->p_scatter = a;
}
p->p_flag &= ~SNOMMU;
p->p_addr = ixtoc(p->p_scatter);
cxrelse(p->p_context);
p->p_flag |= SLOAD;
p->p_time = 0;
return(1);
}
#endif

/*
 * put the current process on
 * the Q of running processes and
 * call the scheduler.
 */
qswtch()
{
    setrq(u.u_proc);
    swtch();
}

/*
 * This routine is called to reschedule the CPU.
 * if the calling process is not in RUN state,
 * arrangements for it to restart must have
 * been made elsewhere, usually by calling via sleep.
 * There is a race here. A process may become
 * ready after it has been examined.
 * In this case, idle() will be called and
 * will return in at most 1HZ time.
 * i.e. its not worth putting an spl() in.
 */
swtch()
{
    register n;
    register struct proc *p, *q, *pp, *pq;
#ifdef mc68881 /* MC68881 floating-point coprocessor */

```

```

extern short fp881;          /* is there an MC68881? */
#endif mc68881

/*
 * If not the idle process, resume the idle process.
 */
sysinfo.pswitch++;
if (u.u_procp != &proc[0]) {
    if (save(u.u_rsav)) {
        sureg();
        return;
    }
}
#ifdef FLOAT
/* sky floating point board */
if (u.u_fpinuse && u.u_fpsaved==0) {
    savfp();
    u.u_fpsaved = 1;
}
#endif
#ifdef mc68881
/* MC68881 floating-point coprocessor */
if (fp881)
    fpsave();
#endif mc68881
#ifdef NONSCATLOAD
resume(proc[0].p_addr, u.u_qsav);
#else
resume(ixtoc(proc[0].p_scatter), u.u_qsav);
#endif
}
/*
 * The first save returns nonzero when proc 0 is resumed
 * by another process (above); then the second is not done
 * and the process-search loop is entered.
 *
 * The first save returns 0 when swtch is called in proc 0
 * from sched(). The second save returns 0 immediately, so
 * in this case too the process-search loop is entered.
 * Thus when proc 0 is awakened by being made runnable, it will
 * find itself and resume itself at rsav, and return to sched().
 */
if (save(u.u_qsav) == 0 && save(u.u_rsav))
    return;
loop:
SPL6();
runrun = 0;
/*
 * Search for highest-priority runnable process
 */
if (p = runq) {
    q = NULL;
    pp = NULL;
    n = 128;
    do {
        if ((p->p_flag&SLOAD) && p->p_pri <= n) {
            pp = p;
            pq = q;
            n = p->p_pri;
        }
        q = p;
    } while (p = p->p_link);
} else
    goto cont;
/*
 * If no process is runnable, idle.
 */
if (pp == NULL) {
cont:
    curpri = PIDLE;
    curproc = &proc[0];
    idle();
    goto loop;
}
p = pp;
q = pq;
if (q == NULL)
    runq = p->p_link;
else
    q->p_link = p->p_link;

```

```

curpri = n;
curproc = p;
SPL0();
/*
 * The rsav (ssav) contents are interpreted in the new address space
 */
n = p->p_flag&SSWAP;
p->p_flag &= ~SSWAP;
#ifdef NONSCATLOAD
resume(p->p_addr, n? u.u_ssav: u.u_rsav);
#else
resume(ixtoc(p->p_scatter), n? u.u_ssav: u.u_rsav);
#endif
}
/*
 * Create a new process-- the internal version of
 * sys fork.
 * It returns 1 in the new process, 0 in the old.
 */
newproc(i)
{
    register struct proc *rpp, *rip;
    register struct user *up;
    register n;
    register a;
    struct proc *pend;
    static mpid;

/*
 * First, just locate a slot for a process
 * and copy the useful info from this process into it.
 * The panic "cannot happen" because fork has already
 * checked for the existence of a slot.
 */
up = &u;
rpp = NULL;
retry:
mpid++;
if (mpid >= MAXPID) {
    mpid = 0;
    goto retry;
}
rip = &proc[0];
n = (struct proc *)v.ve_proc - rip;
a = 0;
do {
    if (rip->p_stat == NULL) {
        if (rpp == NULL)
            rpp = rip;
        continue;
    }
    if (rip->p_pid==mpid)
        goto retry;
    if (rip->p_uid == up->u_ruid)
        a++;
    pend = rip;
} while(rip++, --n);
if (rpp==NULL) {
    if ((struct proc *)v.ve_proc >= &proc[(short)v.v_proc]) {
        if (i) {
            syserr.procovf++;
            up->u_error = EAGAIN;
            return(-1);
        } else
            panic("no procs");
    }
    rpp = (struct proc *)v.ve_proc;
}
if (rpp > pend)
    pend = rpp;
pend++;
#ifdef lint
v.ve_proc = pend;
#else
v.ve_proc = (char *)pend;

```

```

#endif
if (up->u_uid && up->u_ruid) {
    if (rpp == &proc[short](v.v_proc-1) || a > v.v_maxup) {
        up->u_error = EAGAIN;
        return(-1);
    }
}
/*
 * make proc entry for new proc
 */

rip = up->u_procp;
rpp->p_stat = SRUN;
rpp->p_clktim = 0;
rpp->p_flag = SLOAD;
rpp->p_uid = rip->p_uid;
rpp->p_suid = rip->p_suid;
rpp->p_pgrp = rip->p_pgrp;
rpp->p_nice = rip->p_nice;
rpp->p_textp = rip->p_textp;
rpp->p_pid = mpid;
rpp->p_ppid = rip->p_pid;
rpp->p_time = 0;
rpp->p_cpu = rip->p_cpu;
rpp->p_sigign = rip->p_sigign;
rpp->p_pri = PUSER + rip->p_nice - NZERO;
#ifdef NONSCATLOAD
rpp->p_scat = rip->p_scat;
#endif

rpp->p_addr = rip->p_addr;
rpp->p_size = rip->p_size;

/*
 * make duplicate entries
 * where needed
 */

for(n=0; n<NOFILE; n++)
    if (up->u_ofile[n] != NULL)
        up->u_ofile[n]->f_count++;
if (rpp->p_textp != NULL) {
    rpp->p_textp->x_count++;
    rpp->p_textp->x_ccount++;
}
up->u_cdir->i_count++;
if (up->u_rdir)
    up->u_rdir->i_count++;

shmfork(rpp, rip);

/*
 * Partially simulate the environment
 * of the new process so that when it is actually
 * created (by copying) it will look right.
 */
up->u_procp = rpp;
curproc = rpp;
/*
 * When the resume is executed for the new process,
 * here's where it will resume.
 */
if (save(up->u_ssav) {
    sureg();
    return(1);
}
/*
 * If there is not enough memory for the
 * new process, swap out the current process to generate the
 * copy.
 */
if (procdup(rpp) == NULL) {
    rip->p_stat = SIDL;
    xswap(rpp, 0, 0);
    rip->p_stat = SRUN;
}
up->u_procp = rip;

curproc = rip;
if (rip != &proc[0]) /* only do if not scheduler */
    sureg();
setrq(rpp);
rpp->p_flag |= SSWAP;
up->u_rvall = rpp->p_pid; /* parent returns pid of child */
return(0);
}

/*
 * Change the size of the data+stack regions of the process.
 * If the size is shrinking, it's easy-- just release the extra core.
 * If it's growing, and there is core, just allocate it
 * and copy the image, taking care to reset registers to account
 * for the fact that the system's stack has moved.
 * If there is no memory, arrange for the process to be swapped
 * out after adjusting the size requirement-- when it comes
 * in, enough memory will be allocated.
 *
 * After the expansion, the caller will take care of copying
 * the user's stack towards or away from the data area.
 */
#ifdef NONSCATLOAD
expand(newsize)
register newsize;
{
    register struct proc *p;
    register a1, a2;
    register i, n;

    p = u.u_procp;
    n = p->p_size;
    p->p_size = newsize;
    if (n == newsize)
        return;
    a1 = p->p_addr;
    if (n >= newsize) {
#ifdef EXPANDTRACE
        printf("expand:shrinking process by %d clicks\n", n-newsize);
#endif
        mfree(coremap, (mem_t)(n-newsize), (mem_t)(a1+newsize));
    }
    if (save(u.u_ssav) {
        sureg();
        return;
    }
    /*
     * See if can just expand in place
     */
loop:
    a2 = malloc(coremap, newsize-n, (mem_t)(a1+n));
    if (a2 != NULL) {
#ifdef EXPANDTRACE
        printf("expanding in place by %d clicks at click %d\n",
            newsize-n, a1+n);
#endif
        cxrrelse(p->p_context);
        sureg();
        return;
    }
    /*
     * Will we be releasing shared text space anyway.
     * If so, then release it now and try in place
     * expansion again.
     */
    if ((a2 = domall(coremap, (mem_t)newsize)) == NULL)
        if (xmrelse())
            goto loop;
    if (a2 == NULL && (a2 = malloc(coremap, (mem_t)newsize)) == NULL) {
#ifdef EXPANDTRACE
        printf("expand:calling xswap\n");
#endif
        xswap(p, 1, n);
        p->p_flag |= SSWAP;
        qswtch();
    }
}
}

```



```

        /* no return */
    }
    p->p_addr = a2;
    for(i=0; i<n; i++)
        copyseg(a1+i, a2+i);
#ifdef EXPANDTRACE
    printf("expand:copyseg %d from 0x%x to 0x%x\n", n, a1, a2);
#endif
    mfree(coremap, (mem_t)n, (mem_t)a1);
    cxrelse(p->p_context);
    resume((mem_t)a2, u.u_ssav);
}
#else
expand(newsize)
register newsize;
{
    register struct scatter *s;
    register struct proc *p;
    register al, a2;
    register i, n;
    int t;

    p = u.u_procp;
    n = p->p_size;
    p->p_size = newsize;
    if (n == newsize)
        return;
    s = scatmap;
    al = p->p_scat;
    if (n >= newsize) {
        /*
         * shrink memory
         */
        for (i=1; i<newsize; i++)
            al = s[al].sc_index;
        t = scatfreelist.sc_index;
        scatfreelist.sc_index = s[al].sc_index;
        i = al;
        while ((a2 = s[al].sc_index) != SCATEND)
            al = a2;
        s[i].sc_index = SCATEND;
        s[al].sc_index = t;
        nscatfree += n-newsize;
        /*
         * Wake scheduler when freeing memory
         */
        if (runin) {
            runin = 0;
            wakeup((caddr_t)&runin);
        }
        return;
    }
    if (save(u.u_ssav)) {
        sureg();
        return;
    }
    /*
     * expand memory
     */
    if (a2 = memalloc(newsize-n)) {
        /* printf("expanding from %d clicks to %d clicks\n",
            n, newsize); */
        for (i=1; i<n; i++)
            al = s[al].sc_index;
        if (s[al].sc_index != SCATEND)
            printf("expand:SCATEND expected\n");
        s[al].sc_index = a2;
        return;
    }
    xswap(p, 1, n);
    p->p_flag |= SSWAP;
    qswtch();
    /* no return */
}
#endif

```

```

#ifdef NONSCATLOAD
checkscat(s)
char *s;
{
    register struct proc *p;
    register struct text *xp;
    register i;

    i = countscat(scatfreelist.sc_index);
    printf("%s nscatfree=%d actual=%d\n", s, nscatfree, i);
    if (nscatfree < 30) {
        printf("    freelist chain is ");
        dumpscat(scatfreelist.sc_index);
    }
    for (p = &proc[0]; p < (struct proc *)v.ve_proc; p++) {
        if (p->p_stat==0)
            continue;
        xp = p->p_textp;
        printf("    pid=%d %d used (%d text)\n",
            p->p_pid, countscat(p->p_scat),
            xp ? countscat(xp->x_scat) : 0);
        dumpscat(p->p_scat);
        if (xp) {
            dumpscat(xp->x_scat);
        }
    }
    dumpscat(al)
    register al;
    {
        register struct scatter *s;

        s = scatmap;
        printf("    ");
        while (al != SCATEND) {
            printf(" %d,%x", al, ixtoc(al));
            al = s[al].sc_index;
        }
        printf("\n");
    }
    countscat(al)
    register al;
    {
        register struct scatter *s;
        register i;

        i = 0;
        s = scatmap;
        while (al != SCATEND) {
            al = s[al].sc_index;
            i++;
        }
        return(i);
    }
}
#endif

```

```

/*      socket.c      1.3      84/11/02      */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/termio.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/proc.h"
#include "sys/file.h"
#include "sys/inode.h"
#include "sys/buf.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "net/ubavar.h"
#include "sys/stat.h"
#include "sys/ioctl.h"
#include "net/in.h"
#include "net/in_system.h"
#include "net/route.h"

/*
 * Socket support routines.
 */
/*
 * DEAL WITH INTERRUPT NOTIFICATION.
 */
/*
 * Create a socket.
 */
screate(aso, type, asp, asa, options)
struct socket **aso;
int type;
struct sockproto *asp;
struct sockaddr *asa;
int options;
{
    register struct protosw *prp;
    register struct socket *so;
    int pf, proto, error;

    /*
     * Use process standard protocol/protocol family if none
     * specified by address argument.
     */
    if (asp == 0) {
        pf = PF_INET;          /* should be u.u_protobuf */
        proto = 0;
    } else {
        pf = asp->sp_family;
        proto = asp->sp_protocol;
    }

    /*
     * If protocol specified, look for it, otherwise
     * for a protocol of the correct type in the right family.
     */
    if (proto)
        prp = pffindproto(pf, proto);
    else
        prp = pffindtype(pf, type);
    if (prp == 0)
        return (EPROTONOSUPPORT);

    /*
     * Get a socket structure.
     */
    MSGET(so, struct socket, 1);
    if (so == 0)

        return (ENOBUFS);
    so->so_options = options;
    so->so_state = 0;
    if (u.u_uid == 0)
        so->so_state = SS_PRIV;

    /*
     * Attach protocol to socket, initializing
     * and reserving resources.
     */
    so->so_proto = prp;
    error = (*prp->pr_usrreq)(so, PRU_ATTACH, 0, asa);
    if (error) {
#ifdef NEVER
        /*
         * This commenting-out came from jsq@utexas-11; see the
         * iptcp11 messages.  Simply means temp lack of mbufs...
         */
        if (so->so_snd.sb_mbmax || so->so_rcv.sb_mbmax)
            panic("screate");
#endif
        so->so_state |= SS_USERGONE;
        sofree(so);
        return (error);
    }
    *aso = so;
    return (0);
}

sofree(so)
struct socket *so;
{
    if (so->so_pcb || (so->so_state & SS_USERGONE) == 0)
        return;
    sbrelease(&so->so_snd);
    sbrelease(&so->so_rcv);
    MSFREE(so);
}

/*
 * Close a socket on last file table reference removal.
 * Initiate disconnect if connected.
 * Free socket when disconnect complete.
 */
/* THIS IS REALLY A UNIX INTERFACE ROUTINE
 */
soclose(so, exiting)
register struct socket *so;
int exiting;
{
    int s = splnet();          /* conservative */

    if (so->so_pcb == 0)
        goto discard;
    if (exiting)
        so->so_options |= SO_KEEPAIVE;
    if (so->so_state & SS_ISCONNECTED) {
        if ((so->so_state & SS_ISDISCONNECTING) == 0) {
            u.u_error = sodisconnect(so, (struct sockaddr *)0);
            if (u.u_error) {
                if (exiting)
                    goto drop;
                splx(s);
                return;
            }
        }
    }
    if ((so->so_options & SO_DONTLINGER) == 0) {
        if ((so->so_state & SS_ISDISCONNECTING) &&
            (so->so_state & SS_NBIO) &&
            exiting == 0) {
            u.u_error = EINPROGRESS;
            /*
             * billn. This is a kludge rendered unnecessary
             * by a bug fix 12/27/83.
             */
            so->so_state |= SS_USERGONE;
            /*
             */
        }
    }
}

```

```

        splx(s);
        return;
    }
    /* should use tsleep here, for at most linger */
    while (so->so_state & SS_ISCONNECTED)
        (void) sleep((caddr_t)&so->so_timeo, PZERO+1);
}
drop:
    if (so->so_pcb) {
        u.u_error = (*so->so_proto->pr_usrreq)(so, PRU_DETACH, 0, 0);
        if (exiting == 0 && u.u_error) {
            splx(s);
            return;
        }
    }
discard:
    so->so_state |= SS_USERGONE;
    sofree(so);
    splx(s);
}
/*ARGSUSED*/
sostat(so, sb)
    struct socket *so;
    struct stat *sb;
{
    /* bug fix by JC Stewart of sri; this zeros the kernel, should zero
       user data (!)...
       bzero((caddr_t)sb, sizeof (*sb));          /* XXX */
       return (0);                               /* XXX */
    }
}
/*
 * Accept connection on a socket.
 */
soaccept(so, asa)
    struct socket *so;
    struct sockaddr *asa;
{
    int s = splnet();
    int error;

    if ((so->so_options & SO_ACCEPTCONN) == 0) {
        error = EINVAL;          /* XXX */
        goto bad;
    }
    if ((so->so_state & SS_CONNAWAITING) == 0) {
        error = ENOTCONN;
        goto bad;
    }
    so->so_state &= ~SS_CONNAWAITING;
    error = (*so->so_proto->pr_usrreq)(so, PRU_ACCEPT, 0, (caddr_t)asa);
bad:
    splx(s);
    return (error);
}
/*
 * Connect socket to a specified address.
 * If already connected or connecting, then avoid
 * the protocol entry, to keep its job simpler.
 */
soconnect(so, asa)
    struct socket *so;
    struct sockaddr *asa;
{
    int s = splnet();
    int error;

    if (so->so_state & (SS_ISCONNECTED|SS_ISCONNECTING)) {
        error = EISCONN;
        goto bad;
    }
    error = (*so->so_proto->pr_usrreq)(so, PRU_CONNECT, 0, (caddr_t)asa);
}

```

```

bad:
    splx(s);
    return (error);
}
/*
 * Disconnect from a socket.
 * Address parameter is from system call for later multicast
 * protocols. Check to make sure that connected and no disconnect
 * in progress (for protocol's sake), and then invoke protocol.
 */
sodisconnect(so, asa)
    struct socket *so;
    struct sockaddr *asa;
{
    int s = splnet();
    int error;

    if ((so->so_state & SS_ISCONNECTED) == 0) {
        error = ENOTCONN;
        goto bad;
    }
    if (so->so_state & SS_ISDISCONNECTING) {
        error = EALREADY;
        goto bad;
    }
    error = (*so->so_proto->pr_usrreq)(so, PRU_DISCONNECT, 0, asa);
bad:
    splx(s);
    return (error);
}
/*
 * Send on a socket.
 * If send must go all at once and message is larger than
 * send buffering, then hard error.
 * Lock against other senders.
 * If must go all at once and not enough room now, then
 * inform user that this would block and do nothing.
 */
sosend(so, asa)
    register struct socket *so;
    struct sockaddr *asa;
{
    struct mbuf *top = 0;
    register struct mbuf **mp = &top;
    register struct mbuf *m;
    register struct user *up = &u;
    register u_int len;
    register int space;
    int error = 0, s;

    if (sosendallatonce(so) && up->u_count > so->so_snd.sb_hiwat)
        return (EMSGSIZE);
#ifdef notdef
    /* NEED TO PREVENT BUSY WAITING IN SELECT FOR WRITING */
    if ((so->so_snd.sb_flags & SB_LOCK) && (so->so_state & SS_NBIO))
        return (EWOULDBLOCK);
#endif
restart:
    sblock(&so->so_snd);
#define snderr(errno) { error = errno; splx(s); goto release; }
again:
    s = splnet();
    if (so->so_state & SS_CANTSENDMORE) {
        psignal(up->u_procp, SIGPIPE);
        snderr(EPIPE);
    }
    if (so->so_error) {
        error = so->so_error;
        so->so_error = 0;
        splx(s);
        goto release;
    }
    if ((so->so_state & SS_ISCONNECTED) == 0) {
}

```

```

    if (so->so_proto->pr_flags & PR_CONNREQUIRED)
        snderr(ENOTCONN);
    if (asa == 0)
        snderr(EDESTADDRREQ);
}
if (top) {
    error = (*so->so_proto->pr_usrreq)(so, PRU_SEND, top, asa);
    top = 0;
    if (error) {
        splx(s);
        goto release;
    }
    mp = &top;
}
if (up->u_count == 0) {
    splx(s);
    goto release;
}
space = sbspace(&so->so_snd);
if (space <= 0 || sosendallatonce(so) && space < up->u_count) {
    if (so->so_state & SS_NBIO)
        snderr(EWOULDBLOCK);
    sbunlock(&so->so_snd);
    sbwait(&so->so_snd);
    splx(s);
    goto restart;
}
splx(s);
while (up->u_count && space > 0) {
    MGET(m, 1);
    if (m == NULL) {
        error = ENOBUFS;
        goto release;
    }
    m->m_off = MMINOFF;
    len = (u_int)MIN(((int)MLEN), ((int)up->u_count));
    iomove(mtod(m, caddr_t), (int)len, B_WRITE);
    if (error = up->u_error) goto release;
    m->m_len = len;
    *mp = m;
    mp = &m->m_next;
    space = sbspace(&so->so_snd);
}
goto again;

release:
sbunlock(&so->so_snd);
if (top)
    m_freem(top);
return (error);
}

soreceive(so, asa)
register struct socket *so;
struct sockaddr *asa;
{
    register struct mbuf *m, *n;
    u_int len;
    int eor, s, error = 0, cnt = u.u_count;
    caddr_t base = u.u_base;

restart:
sblock(&so->so_rcv);
s = splnet();

#define rcverr(errno) { error = errno; splx(s); goto release; }
if (so->so_rcv.sb_cc == 0) {
    if (so->so_error) {
        error = so->so_error;
        so->so_error = 0;
        splx(s);
        goto release;
    }
    if (so->so_state & SS_CANTRCVMORE) {
        splx(s);
        goto release;
    }
}

```

```

    }
    if ((so->so_state & SS_ISCONNECTED) == 0 &&
        (so->so_proto->pr_flags & PR_CONNREQUIRED))
        rcverr(ENOTCONN);
    if (so->so_state & SS_NBIO)
        rcverr(EWOULDBLOCK);
    sbunlock(&so->so_rcv);
    sbwait(&so->so_rcv);
    splx(s);
    goto restart;
}
m = so->so_rcv.sb_mb;
if (m == 0)
    panic("receive");
if (so->so_proto->pr_flags & PR_ADDR) {
    if (m->m_len != sizeof(struct sockaddr))
        panic("soreceive addr");
    if (asa)
        bcopy(mtod(m, caddr_t), (caddr_t)asa, sizeof(*asa));
    so->so_rcv.sb_cc -= m->m_len;
    so->so_rcv.sb_mbcnt -= MSIZE;
    m = m_free(m);
    if (m == 0)
        panic("receive 2");
    so->so_rcv.sb_mb = m;
}
so->so_state &= ~SS_RCVATMARK;
if (so->so_oobmark && cnt > so->so_oobmark)
    cnt = so->so_oobmark;
eor = 0;
do {
    len = MIN(m->m_len, cnt);
    splx(s);
    iomove(mtod(m, caddr_t), (int)len, B_READ);
    cnt -= len;
    s = splnet();
    if (len == m->m_len) {
        eor = (int)m->m_act;
        sbfree(&so->so_rcv, m);
        so->so_rcv.sb_mb = m->m_next;
        MFREE(m, n);
    } else {
        m->m_off += len;
        m->m_len -= len;
        so->so_rcv.sb_cc -= len;
    }
} while ((m = so->so_rcv.sb_mb) && cnt && !eor);
if ((so->so_proto->pr_flags & PR_ATOMIC) && eor == 0)
    do {
        if (m == 0)
            panic("receive 3");
        sbfree(&so->so_rcv, m);
        eor = (int)m->m_act;
        so->so_rcv.sb_mb = m->m_next;
        MFREE(m, n);
        m = n;
    } while (eor == 0);
if ((so->so_proto->pr_flags & PR_WANTRCVD) && so->so_pcb)
    (*so->so_proto->pr_usrreq)(so, PRU_RCVD, 0, 0);
if (so->so_oobmark) {
    so->so_oobmark -= u.u_base - base;
    if (so->so_oobmark == 0)
        so->so_state |= SS_RCVATMARK;
}
release:
sbunlock(&so->so_rcv);
splx(s);
return (error);
}

sohasoutofband(so)
struct socket *so;
{
    struct proc *pfind();
    if (so->so_pgrp == 0)

```

```

        return;
    if (so->so_pgrp > 0)
        /*
         * gsignal(so->so_pgrp, SIGURG);
         */
        signal(so->so_pgrp, SIGURG);
    else {
        struct proc *p = pfind(-so->so_pgrp);

        if (p)
            psignal(p, SIGURG);
    }
}

/*ARGSUSED*/
soioctl(so, cmd, cmdp)
register struct socket *so;
int cmd;
register caddr_t cmdp;
{
    extern struct uba_device ubdinit[];

    long *iaddrp = (long *)((int)ubdinit + 0xA);

    switch (cmd) {
    case SIOCCIADDR:
        /* set internet address ie write into uba flags */
        if (u.u_ruid != 0 && u.u_uid != 0) {
            u.u_error = EPERM;
            return;
        }
        if (copyin(cmdp, (caddr_t)iaddrp, sizeof (long))) {
            u.u_error = EFAULT;
            return;
        }
        return;
    case SIOCGIADDR:
        if (copyout((caddr_t)iaddrp, cmdp, sizeof (long)))
            u.u_error = EFAULT;
        return;
    case FIONBIO: {
        int nbio;
        if (copyin(cmdp, (caddr_t)&nbio, sizeof (nbio))) {
            u.u_error = EFAULT;
            return;
        }
        if (nbio)
            so->so_state |= SS_NBIO;
        else
            so->so_state &= ~SS_NBIO;
        return;
    }
    case FIOASYNC: {
        int async;
        if (copyin(cmdp, (caddr_t)&async, sizeof (async))) {
            u.u_error = EFAULT;
            return;
        }
        if (async)
            so->so_state |= SS_ASYNC;
        else
            so->so_state &= ~SS_ASYNC;
        return;
    }
    case FIONREAD: {
        long nread = so->so_rcv.sb_cc;
        if (copyout((caddr_t)&nread, cmdp, sizeof (nread)))
            u.u_error = EFAULT;
        return;
    }
}

```

```

    case SIOCSKEEP: {
        int keep;
        if (copyin(cmdp, (caddr_t)&keep, sizeof (keep))) {
            u.u_error = EFAULT;
            return;
        }
        if (keep)
            so->so_options |= SO_KEEPAKIVE;
        else
            so->so_options &= ~SO_KEEPAKIVE;
        return;
    }
    case SIOCGKEEP: {
        int keep = (so->so_options & SO_KEEPAKIVE) != 0;
        if (copyout((caddr_t)&keep, cmdp, sizeof (keep)))
            u.u_error = EFAULT;
        return;
    }
    case SIOCSLINGER: {
        int linger;
        if (copyin(cmdp, (caddr_t)&linger, sizeof (linger))) {
            u.u_error = EFAULT;
            return;
        }
        so->so_linger = linger;
        if (so->so_linger)
            so->so_options &= ~SO_DONTLINGER;
        else
            so->so_options |= SO_DONTLINGER;
        return;
    }
    case SIOCGLINGER: {
        int linger = so->so_linger;
        if (copyout((caddr_t)&linger, cmdp, sizeof (linger))) {
            u.u_error = EFAULT;
            return;
        }
    }
    case SIOCSPPGRP: {
        int pgrp;
        if (copyin(cmdp, (caddr_t)&pgrp, sizeof (pgrp))) {
            u.u_error = EFAULT;
            return;
        }
        so->so_pgrp = pgrp;
        return;
    }
    case SIOCGPPGRP: {
        int pgrp = so->so_pgrp;
        if (copyout((caddr_t)&pgrp, cmdp, sizeof (pgrp))) {
            u.u_error = EFAULT;
            return;
        }
    }
    case SIOCDDONE: {
        int flags;
        if (copyin(cmdp, (caddr_t)&flags, sizeof (flags))) {
            u.u_error = EFAULT;
            return;
        }
        flags++;
        if (flags & FREAD) {
            int s = splimp();
            socantrcvmore(so);
            sbflush(&so->so_rcv);
            splx(s);
        }
        if (flags & FWRITE)
            u.u_error = (*so->so_proto->pr_usrreq)(so, PRU_SHUTDOWN, (struct mbuf *)0, 0);
        return;
    }
}

```

```
case SIOCSENDOOb: {
    char oob;
    struct mbuf *m;
    if (copyin(cmdp, (caddr_t)&oob, sizeof (oob))) {
        u.u_error = EFAULT;
        return;
    }
    m = m_get(M_DONTWAIT);
    if (m == 0) {
        u.u_error = ENOBUFS;
        return;
    }
    m->m_off = MMINOFF;
    m->m_len = 1;
    *mtod(m, caddr_t) = oob;
    (*so->so_proto->pr_usrreq)(so, PRU_SENDOOB, m, 0);
    return;
}

case SIOCRVVOOB: {
    struct mbuf *m = m_get(M_DONTWAIT);
    if (m == 0) {
        u.u_error = ENOBUFS;
        return;
    }
    m->m_off = MMINOFF; *mtod(m, caddr_t) = 0;
    (*so->so_proto->pr_usrreq)(so, PRU_RCVVOOB, m, 0);
    if (copyout(mtod(m, caddr_t), cmdp, sizeof (char))) {
        u.u_error = EFAULT;
        return;
    }
    (void) m_free(m);
    return;
}

case SIOCATMARK: {
    int atmark = (so->so_state&SS_RCVATMARK) != 0;
    if (copyout((caddr_t)&atmark, cmdp, sizeof (atmark))) {
        u.u_error = EFAULT;
        return;
    }
    return;
}

/* routing table update calls */
case SIOCADDRt:
case SIOCDELRT:
case SIOCCHGRT: {
    struct rtentry route;
    if (!suser())
        return;
    if (copyin(cmdp, (caddr_t)&route, sizeof (route))) {
        u.u_error = EFAULT;
        return;
    }
    u.u_error = rtrequest(cmd, &route);
    return;
}

/* type/protocol specific ioctls */
}
u.u_error = EOPNOTSUPP;
}
```

```

/* socketsubr.c 4.23 82/06/14 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/mmu.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/errno.h"
#include "sys/user.h"
#include "sys/proc.h"
#include "sys/file.h"
#include "sys/inode.h"
#include "sys/buf.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "net/in.h"
#include "net/in_system.h"

/*
 * Primitive routines for operating on sockets and socket buffers
 */

/*
 * Procedures to manipulate state flags of socket
 * and do appropriate wakeups. Normal sequence is that
 * soisconnecting() is called during processing of connect() call,
 * resulting in an eventual call to soisconnected() if/when the
 * connection is established. When the connection is torn down
 * soisdisconnecting() is called during processing of disconnect() call,
 * and soisdisconnected() is called when the connection to the peer
 * is totally severed. The semantics of these routines are such that
 * connectionless protocols can call soisconnected() and soisdisconnected()
 * only, bypassing the in-progress calls when setting up a 'connection'
 * takes no time.
 *
 * When higher level protocols are implemented in
 * the kernel, the wakeups done here will sometimes
 * be implemented as software-interrupt process scheduling.
 */

soisconnecting(so)
    struct socket *so;
{
    so->so_state &= ~(SS_ISCONNECTED|SS_ISDISCONNECTING);
    so->so_state |= SS_ISCONNECTING;
    wakeup((caddr_t)&so->so_timeo);
}

soisconnected(so)
    struct socket *so;
{
    so->so_state &= ~(SS_ISCONNECTING|SS_ISDISCONNECTING);
    so->so_state |= SS_ISCONNECTED;
    wakeup((caddr_t)&so->so_timeo);
    sorwakeup(so);
    sowwakeup(so);
}

soisdisconnecting(so)
    struct socket *so;
{
    so->so_state &= ~SS_ISCONNECTING;
    so->so_state |= (SS_ISDISCONNECTING|SS_CANTRCVMORE|SS_CANTSENDMORE);
    wakeup((caddr_t)&so->so_timeo);
    sowwakeup(so);
    sorwakeup(so);
}

soisdisconnected(so)
    struct socket *so;
{
    so->so_state &= ~(SS_ISCONNECTING|SS_ISCONNECTED|SS_ISDISCONNECTING);
    so->so_state |= (SS_CANTRCVMORE|SS_CANTSENDMORE);
    wakeup((caddr_t)&so->so_timeo);
    sowwakeup(so);
}

/*
 * Socket select/wakeup routines.
 */

/*
 * Interface routine to select() system
 * call for sockets.
 */
soselect(so, rw)
    register struct socket *so;
    int rw;
{
    int s = splnet();

    switch (rw) {
    case FREAD:
        if (soreadable(so)) {
            splx(s);
            return (1);
        }
        sbselqueue(&so->so_rcv);
        break;
    case FWRITE:
        if (sowriteable(so)) {
            splx(s);
            return (1);
        }
        sbselqueue(&so->so_snd);
        break;
    }
    splx(s);
    return (0);
}

/*
 * Queue a process for a select on a socket buffer.
 */
socantsendmore(so)
    struct socket *so;
{
    so->so_state |= SS_CANTSENDMORE;
    sowwakeup(so);
}

socantrcvmore(so)
    struct socket *so;
{
    so->so_state |= SS_CANTRCVMORE;
    sorwakeup(so);
}

/*
 * Socantsendmore indicates that no more data will be sent on the
 * socket; it would normally be applied to a socket when the user
 * informs the system that no more data is to be sent, by the protocol
 * code (in case PRU_SHUTDOWN). Socantrcvmore indicates that no more data
 * will be received, and will normally be applied to the socket by a
 * protocol when it detects that the peer will send no more data.
 * Data queued for reading in the socket may yet be read.
 */
}

```

```

*/
sbselqueue(sb)
    struct sockbuf *sb;
{
    register struct proc *p;

    if ((p = sb->sb_sel) && p->p_wchan == (caddr_t)&selwait)
        sb->sb_flags |= SB_COLL;
    else
        sb->sb_sel = u.u_procp;
}

/*
 * Wait for data to arrive at/drain from a socket buffer.
 */
sbwait(sb)
    struct sockbuf *sb;
{
    sb->sb_flags |= SB_WAIT;
    (void) sleep((caddr_t)&sb->sb_cc, PZERO+1);
}

/*
 * Wakeup processes waiting on a socket buffer.
 */
sbwakeup(sb)
    struct sockbuf *sb;
{
    if (sb->sb_sel) {
        selwakeup(sb->sb_sel, sb->sb_flags & SB_COLL);
        sb->sb_sel = 0;
        sb->sb_flags &= ~SB_COLL;
    }
    if (sb->sb_flags & SB_WAIT) {
        sb->sb_flags &= ~SB_WAIT;
        wakeup((caddr_t)&sb->sb_cc);
    }
}

/*
 * Socket buffer (struct sockbuf) utility routines.
 *
 * Each socket contains two socket buffers: one for sending data and
 * one for receiving data. Each buffer contains a queue of mbufs,
 * information about the number of mbufs and amount of data in the
 * queue, and other fields allowing select() statements and notification
 * on data availability to be implemented.
 *
 * Before using a new socket structure it is first necessary to reserve
 * buffer space to the socket, by calling sbreserve. This commits
 * some of the available buffer space in the system buffer pool for the
 * socket. The space should be released by calling sbrelease when the
 * socket is destroyed.
 *
 * The routine sbappend() is normally called to append new mbufs
 * to a socket buffer, after checking that adequate space is available
 * comparing the function spspace() with the amount of data to be added.
 * Data is normally removed from a socket buffer in a protocol by
 * first calling m_copy on the socket buffer mbuf chain and sending this
 * to a peer, and then removing the data from the socket buffer with
 * sbdrop when the data is acknowledged by the peer (or immediately
 * in the case of unreliable protocols.)
 *
 * Protocols which do not require connections place both source address
 * and data information in socket buffer queues. The source addresses
 * are stored in single mbufs after each data item, and are easily found
 * as the data items are all marked with end of record markers. The
 * sbappendaddr() routine stores a datum and associated address in
 * a socket buffer. Note that, unlike sbappend(), this routine checks
 * for the caller that there will be enough space to store the data.
 * It fails if there is not enough space, or if it cannot find
 * a mbuf to store the address in.
 *
 * The higher-level routines sosend and soreceive (in socket.c)
    */
    /* also add data to, and remove data from socket buffers respectively.
    */

#ifdef notdef
/* billm -- add this routine from 4.1c for 4.1c udp */
soreserve(so, sndcc, rcvcc)
    struct socket *so;
    int sndcc, rcvcc;
{
    if (sbreserve(&so->so_snd, sndcc) == 0)
        goto bad;
    if (sbreserve(&so->so_rcv, rcvcc) == 0)
        goto bad2;
    return (0);
bad2:
    sbrelease(&so->so_snd);
bad:
    return (ENOBUFS);
}
#endif

/*
 * Allot mbufs to a sockbuf.
 */
sbreserve(sb, cc)
    struct sockbuf *sb;
{
    /* someday maybe this routine will fail... */
    sb->sb_hiwat = cc;
    sb->sb_mbmax = cc*2;
    return (1);
}

/*
 * Free mbufs held by a socket, and reserved mbuf space.
 */
sbrelease(sb)
    struct sockbuf *sb;
{
    sbflush(sb);
    sb->sb_hiwat = sb->sb_mbmax = 0;
}

/*
 * Routines to add (at the end) and remove (from the beginning)
 * data from a mbuf queue.
 */

/*
 * Append mbuf queue m to sockbuf sb.
 */
sbappend(sb, m)
    register struct mbuf *m;
    register struct sockbuf *sb;
{
    register struct mbuf *n;

    n = sb->sb_mb;
#ifdef SIGH
    mcheck(n, "sbappend");
#endif
    if (n)
        while (n->m_next)
            n = n->m_next;
    while (m) {
        if (m->m_len == 0 && (int)m->m_act == 0) {
            m = m_free(m);
            continue;
        }
        if (n && n->m_off <= MMAXOFF && m->m_off <= MMAXOFF &&
            (int)n->m_act == 0 && (int)m->m_act == 0 &&
            (n->m_off + n->m_len + m->m_len) <= MMAXOFF) {

```



```

        MBCOPY(m, 0, n, n->m_len, (u_int)m->m_len);
        n->m_len += m->m_len;
        sb->sb_cc += m->m_len;
        m = m_free(m);
        continue;
    }
    sballot(sb, m);
    if (n == 0) {
        sb->sb_mb = m;
#ifdef SIGH
        mcheck(n, "sbappend2");
#endif
    }
    else
        n->m_next = m;
    n = m;
    m = m->m_next;
    n->m_next = 0;
}

/*
 * Append data and address.
 * Return 0 if no space in sockbuf or if
 * can't get mbuf to stuff address in.
 */
sbappendaddr(sb, asa, m0)
    struct sockbuf *sb;
    struct sockaddr *asa;
    struct mbuf *m0;
{
    struct sockaddr *msa;
    register struct mbuf *m;
    register int len = sizeof (struct sockaddr);
    struct sockaddr sa;

    sa = *asa;
    m = m0;
    if (m == 0)
        panic("sbappendaddr");
    for (;;) {
        len += m->m_len;
        if (m->m_next == 0) {
            m->m_act = (struct mbuf *)1;
            break;
        }
        /* This else clause is an sri bug-fix by JC Stewart */
        else {
            m->m_act = (struct mbuf *) 0;
        }
        /* */
        m = m->m_next;
    }
    if (len > sb->space(sb))
        return (0);
    m = m_get(M_DONTWAIT);
    if (m == 0)
        return (0);
    m->m_off = MMINOFF;
    m->m_len = sizeof (struct sockaddr);
    MAPSAVE();
    msa = mtod(m, struct sockaddr *);
    *msa = sa;
    MAPREST();
    m->m_act = (struct mbuf *)1;
    sbappend(sb, m);
    sbappend(sb, m0);
    return (1);
}

/*
 * Free all mbufs on a sockbuf mbuf chain.
 * Check that resource allocations return to 0.
 */
sbflush(sb)
    struct sockbuf *sb;

```

```

{
    if (sb->sb_flags & SB_LOCK)
        panic("sbflush");
    if (sb->sb_cc)
        sbdrop(sb, sb->sb_cc);
#ifdef SIGH
    /*
     * else
     * return;
     */
#endif
    if (sb->sb_cc || sb->sb_mbcnt || sb->sb_mb) {
#ifdef SIGH
        extern struct mbuf *mfree;

        printf("sbflush: sb_cc=%x, sb_mbcnt=%x, sb_mb=%x, mfreep=%x\n",
            sb->sb_cc, sb->sb_mbcnt, sb->sb_mb, mfreep);
#endif
        panic("sbflush 2");
    }
}

/*
 * Drop data from (the front of) a sockbuf chain.
 */
sbdrop(sb, len)
    register struct sockbuf *sb;
    register int len;
{
    register struct mbuf *m = sb->sb_mb, *mn;

#ifdef SIGH
    mcheck (m, "sbdrop1");
#endif
    while (len > 0) {
        if (m == 0)
            panic("sbdrop");
        if (m->m_len > len) {
            m->m_len -= len;
            m->m_off += len;
            sb->sb_cc -= len;
            break;
        }
        len -= m->m_len;
        sbfree(sb, m);
        MFREE(m, mn);
        m = mn;
    }
    sb->sb_mb = m;
#ifdef SIGH
    mcheck (m, "sbdrop2");
#endif
}

```

```

/* #define HOWFAR */
/* #define WBBLK      /* allows writing block 0 */
/* #define UNISOFT    /* allows access to restricted blocks on boot disk */
#define KLUDGE /* kludge for format to work */
*/
* (C) Copyright 1983 UniSoft Systems of Berkeley CA
*
* Sony driver
* and eject driver
*
*/

#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/utsname.h"
#include "sys/buf.h"
#include "sys/elog.h"
#include "sys/erec.h"
#include "sys/iobuf.h"
#include "sys/system.h"
#include "sys/var.h"
#include "sys/uioct1.h"
#include "sys/al_ioc1.h"
#include "sys/diskformat.h"
#include "setjmp.h"
#include "sys/pport.h"
#include "sys/sony.h"
#include "sys/cops.h"

#define NRETRY 5

#define physical(d) ((minor(d)>>4)&0xF) /* physical unit number 0-15 */
#define logical(d) (minor(d)&0x7) /* logical unit number, 0-7 */
#define splsn spl1

#define GETBUF(bp) splsn(); \
while (bp->b_flags & B_BUSY) { \
    bp->b_flags |= B_WANTED; \
    (void)sleep((caddr_t)bp, PRIBIO+1); \
} \
bp->b_flags |= B_BUSY; \
spl0()

#define FREEBUF(bp) splsn(); \
if (bp->b_flags & B_WANTED) \
    wakeup((caddr_t)bp); \
bp->b_flags = 0; \
spl0()

struct buf snhbuf; /* header buffer (for reading block 0) */
struct buf sncbuf; /* command buffer */
struct buf snrbuf;
struct iostat snstat[NSN];
struct iobuf snTAB = tabinit(SN1,snstat); /* active buffer header */

#ifdef WBBLK
char sn_bblk[512];
#endif
char sn_bcount[NSN*2]; /* block opens */
char sn_ccount[NSN*2]; /* character opens */
#define count(d) (d<<1) /* first char is how many opens */
#define boot(d) ((d<<1)+1) /* 2nd char (sn_bcount only) is whether a boot disk */
char snbf; /* index into sn_fns array */
#ifdef KLUDGE
char noerror; /* read before format is acceptable error */
#endif

struct sn_sizes {
    caddr_t sn_offset, sn_size;

```

```

} sn_sizes[] = {
    0,          800,      /* a = filesystem */
    201,        599,      /* b = filesystem on a boot Sony */
    0,          201,      /* c = 1-100 for lisa (serialization),
                        101-200 for boot program */
    0,          0,        /* d = unused */
    0,          0,        /* e = unused */
    0,          0,        /* f = unused */
    0,          0,        /* g = unused */
    0,          800,      /* h = entire disk */
};

/*
 * called from oem7init at (at least) level 1, so won't be interrupted
 * by parallel port 0 or sony
 */
sninit()
{
    char c = 0x00; /* to avoid clear byte instructions */

    if (SNIOS->type) {
        printf("Microdiskette with %d head%s\n", SNIOS->type,
            (SNIOS->type==1)?"":"s");
    } else {
        printf("Unix sninit: not a sony drive\n");
        return;
    }
    SNIOS->drive=0x80; /* always lower drive */
    SNIOS->side = c; /* always first side */
    SNIOS->mask= 0xff; /* clear ints and enable */
    if (snwaitrdy()) {
        printf("Unix sninit: command to clear status failed\n");
        return;
    }
    if (SNIOS->drv_connect != 0xff) {
        printf("Unix sninit: no drive connected\n");
        return;
    }
    SNIOS->gobyte=SN_CLRST;
    SNIOS->mask =0x80; /* enable interrupts */
    if (snwaitrdy()) {
        printf("Unix sninit: command to clear status failed\n");
    }
    SNIOS->gobyte=SN_STMASK;
}

/*
 * check block 0 to see whether this is a boot disk
 */
snocz(dev)
register dev_t dev;
{
    static char hp[BSIZE];
    register struct buf *bp = &snhbuf;
    register rc = 0;

    GETBUF(bp);
    sn_bcount[boot(physical(dev))] = 0; /* for now set to non-boot */
    bp->b_bcount = BSIZE;
    bp->b_dev = dev | 7; /* set logical unit 7 */
    bp->b_blkno = 0;
    bp->b_un.b_addr = hp;
    bp->b_flags |= B_READ;
    snbf = 1;
    snstrategy(bp);
    iowait(bp);
    if (bp->b_flags & B_ERROR) {
        rc++;
        snbf = 0;
    }
    FREEBUF(bp);
    return(rc);
}

/*
 * don't really check whether it's a boot disk
 */

```

```

*/
/* ARGUSED */
snconf(dev)
dev_t dev;
{
    snbf = 1;
    return(0);
}

struct sn_fns {
    int (*fnc)();
} sn_fns[] = {
    snbf, /* the real routine to check block 0 */
    snconf /* fake routine to confuse */
};

snopen(dev)
register dev_t dev;
{
    dev = physical(dev);
    if (dev >= NSN) {
        u.u_error = ENXIO;
        return(-1);
    }
    return(0);
}

snbopen(dev)
register dev_t dev;
{
    if (snopen(dev) == 0)
        sn_bcount[count(physical(dev))]++;
}

snccopen(dev)
register dev_t dev;
{
    if (snopen(dev) == 0)
        sn_ccount[count(physical(dev))]++;
}

sncclose(dev)
register dev_t dev;
{
    if (physical(dev) >= NSN) {
        u.u_error = ENXIO;
        return(-1);
    }
    snbf = 0;
    return(0);
}

snbclose(dev)
register dev_t dev;
{
    if (sncclose(dev) == 0)
        sn_bcount[count(physical(dev))] = 0;
}

snccclose(dev)
register dev_t dev;
{
    if (sncclose(dev) == 0)
        sn_ccount[count(physical(dev))] = 0;
}

snstrategy(bp)
register struct buf *bp;
{
    register punit;

    punit = physical(bp->b_dev);
    if (bp == &snbuf) { /* if command */
        snstat[punit].io_misc++; /* errlog: */
        spln();
        if (sntab.b_actf == (struct buf *)NULL) /* set up links */
            sntab.b_actf = bp;
        else
            sntab.b_act1->av_forw = bp;
        sntab.b_act1 = bp;
        bp->av_forw = (struct buf *)NULL;
    } else {
        if ((*sn_fns[snbf].fnc)(bp->b_dev)) {
            bp->b_flags |= B_ERROR;
            iodone(bp); /* resets flags */
            return;
        }
        snstat[punit].io_ops++; /* errlog: */
        /* resid for disksort */
        bp->b_resid = bp->b_blkno + sn_sizes[logical(bp->b_dev)].sn_offset;
        splsn();
        disksort(&sntab, bp);
    }
    if (sntab.b_active == 0)
        snstart();

    spl0();
}

snstart()
{
    register struct buf *bp;
    register punit, lunit;
    register daddr_t bn;
    caddr_t addr;

loop:
    if ((bp = sntab.b_actf) == (struct buf *)NULL)
        return;
    if (sntab.b_active == 0) {
        sntab.b_active = 1;
        if (bp != &snbuf)
            bp->b_resid = bp->b_bcount;
    }
    punit = physical(bp->b_dev);
    lunit = logical(bp->b_dev);
    blkacty |= (1<<SN1);
    if (bp == &snbuf) {
#ifdef WBBLK
        if (bp->b_resid == UIOCWBBLK)
            snw0(punit);
        else
#ifdef WBBLK
            snccmd(bp->b_resid); /* b_resid holds the command */
            return;
        }
        bn = bp->b_blkno + ((bp->b_bcount - bp->b_resid) >> 9);
        if (bp->b_resid < 512 ||
            bn >= sn_sizes[lunit].sn_size ||
            ((bn += sn_sizes[lunit].sn_offset) >= SN_MAXBN) ||
            (bn <= 200 && sn_bcount[boot(punit)])) {
#ifdef HOWFAR
            if (bp->b_resid != 0)
                printf("Unix snstart: blkno=%d resid=%d bn=%d\n",
                    bp->b_blkno, bp->b_resid, bn);
#endif
            blkacty &= ~(1<<SN1);
            if (sntab.b_errcnt)
                logberr(&sntab, 0); /* errlog non-fatal errors */
            sntab.b_active = 0;
            sntab.b_errcnt = 0;
            sntab.b_actf = bp->av_forw;
            iodone(bp);
            goto loop;
        }
        addr = bp->b_un.b_addr + bp->b_bcount - bp->b_resid;
        snrw(punit, bn, bp->b_flags&B_READ, addr);
    }
}

snread(dev)
dev_t dev;
{
    physio(snstrategy, &snbuf, dev, B_READ);
}

```

```

}

snwrite(dev)
dev_t dev;
{
    physio(snstrategy, &snrbuf, dev, B_WRITE);
}

struct sn_blockmap {
    int maxblock, sectors;
} sn_blockmap[] = {
    192, 12,
    176, 11,
    160, 10,
    144, 9,
    128, 8
};

/* ARGSUSED */
snrw(unit, bn, rw, addr) /* always called at priority sn (see splsn above) */
int unit, rw;
register daddr_t bn;
register caddr_t addr;
{
    register char *pm = (char *)SN_DATABUF;
    register struct sn_blockmap *p = sn_blockmap;
    register int i;
    int track, sector;
    char c = 0x00;

    track = 0;
    while( bn >= p->maxblock ) {
        bn -= p->maxblock;
        track += 16;
        p++;
    }
    sector = bn % p->sectors;
    track += bn / p->sectors;
    if (snwaitrdy())
        return;
    SNIIOB->side=c;
    SNIIOB->drive=0x80;
    SNIIOB->track = track;
    SNIIOB->sector= sector;
    if (rw) {
        SNIIOB->cmd=SN_READ;
    } else {
        SNIIOB->cmd=SN_WRITE;
        i = 511;
        do {
            *pm++; /* cmos ram, every other byte */
            *pm++ = *addr++;
        } while (--i != -1);
    }
    if (snwaitrdy())
        return;
    SNIIOB->gobyte = SN_CMD;
}

/* ARGSUSED */
snioctl(dev, cmd, adr, flag)
dev_t dev;
caddr_t adr;
{
    int unit;
    register struct buf *bp;

    if ((unit = physical(dev)) >= NSN) {
        u.u_error = EFAULT;
        return;
    }
    bp = &snrbuf;
    switch (cmd) {
        case AL_EJECT:
            if ((sn_bcount[count(unit)] > 0)
                || (sn_ccount[count(unit)] > 1)) {

```

```

                u.u_error = EINVAL;
                return;
            }
            break;
        case UIOCFORMAT:
            if (!suser()) {
                u.u_error = EPERM;
                return;
            }
            noerror=1;
            (void)(snzc(0));
            noerror=0;
            break;
        case UIOCWBBLK:
            if (!suser()) {
                u.u_error = EPERM;
                return;
            }
            if (copyin(adr, (caddr_t)sn_bblk, sizeof(sn_bblk))) {
                u.u_error = EFAULT;
                return;
            }
            break;
        default:
            u.u_error = ENOTTY;
            return;
    }
    GETBUF(bp);
    bp->b_dev = dev;
    bp->b_resid = cmd; /* stash the command in resid */
    u.u_error = 0; /* any error on snzc is OK */
    snstrategy(bp);
    iowait(bp);
    if (bp->b_flags & B_ERROR)
        u.u_error = EIO;
    FREEBUF(bp);
}

snrcmd(cmd) /* always called at priority sn (see splsn above) */
unsigned int cmd;
{
    if (snwaitrdy())
        return;
    switch (cmd) {
        case UIOCFORMAT:
            SNIIOB->confirm=0xff;
            SNIIOB->cmd=SN_FORMAT;
            break;
        case AL_EJECT:
            SNIIOB->cmd=SN_EJECT;
            break;
        default:
            return;
    }
    SNIIOB->gobyte = SN_CMD;
}

#ifdef WBBLK
snw0(punit)
int punit;
{
    register char *pm;
    struct sn_hdr sn_hdr;
    register char *addr;
    register i;
    char c = 0x00;

    if (snwaitrdy())
        return;
    SNIIOB->side=c;
    SNIIOB->drive=0x80;
    SNIIOB->track = 0;

```

```

SNIOS->sector= 0;
SNIOS->cmd=SN_WRITE;
sn_hdr.version = 0;
sn_hdr.volume = 0;
sn_hdr.fileid = FILEID;
sn_hdr.relpq = 0;
sn_hdr.dum1 = 0;
sn_hdr.dum2 = 0;
i = sizeof(sn_hdr) - 1;
pm = (char *)SN_HDRBUF; /* write special 12-byte hdr */
addr = (char *)&sn_hdr;
do {
    *pm++; /* cmos ram, every other byte */
    *pm++ = (unsigned)*addr++;
} while (--i != -1);
i = 511;
pm = (char *)SN_DATABUF; /* write regular 512-byte buffer */
addr = (char *)sn_bblk;
do {
    *pm++; /* cmos ram, every other byte */
    *pm++ = *addr++;
} while (--i != -1);
if(snwaitrty())
    return;
SNIOS->gobyte = SN_CMD;
}
#endif WBLK

snintr() /* called at pl 1 */
{
    struct buf *bp;
    register short i;
    register char *addr;
    register char *pm = (char *)SN_DATABUF;
    int cnt;
    struct deverreg snreg[1]; /* errlog: */

    i = SNIOS->status;
#ifdef HOWFAR
    if (i)
        printf("Unix snintr: SNIOS->status = 0x%x\n",i);
#endif HOWFAR
    if (sntab.b_active == 0) {
#ifdef HOWFAR
        printf("Unix snintr: b_active==0\n");
#endif HOWFAR
        snbf = 0; /* force real check for boot disk */
        SNIOS->mask = SN_CLEARMSK;
        SNIOS->gobyte=SN_CLRST;
        (void) snwaitrty();
        return;
    }
    if ((bp = sntab.b_actf) == (struct buf *)NULL) {
#ifdef HOWFAR
        printf("Unix snintr: bp==0\n");
#endif HOWFAR
        snbf = 0; /* force real check for boot disk */
        SNIOS->mask = SN_CLEARMSK;
        SNIOS->gobyte=SN_CLRST;
        (void) snwaitrty();
        return;
    }
    cnt = 512;
    if (i) {
        cnt = 0;
        /* errlog: */
        sntab.io_stp = &snstat[physical(bp->b_dev)];
        snreg[0].drvalue = i;
        snreg[0].drname = "status";
        snreg[0].drbits = "status register";
        fmtberr(&sntab, (unsigned)physical(bp->b_dev),
            (unsigned)(SNIOS->track), /* trk */
            (unsigned)(SNIOS->side), /* head */
            (unsigned)(SNIOS->sector), /* sector */
            (long)(sizeof(snreg)/sizeof(snreg[0])), /* regcnt */
            &snreg[0]);
    }
    if (++sntab.b_errcnt > NRETRY || bp == &snbuf) {
        bp->b_flags |= B_ERROR;
        logberr(&sntab, 1); /* errlog: */
    }
}
/*
 * because a single buffer can take several io operations,
 * we leave it to snstart() to figure out when it's done
 */
if (bp->b_flags&B_ERROR || bp == &snbuf) {
#ifdef KLUDGE
    if (!noerror)
#endif KLUDGE
    if (bp->b_flags & B_ERROR) {
        printf("Unix: HARD I/O ERROR on /dev/s%d%c ",
            physical(bp->b_dev), logical(bp->b_dev)+'a');
        if (bp != &snbuf)
            printf("bn %d\n",
                bp->b_blkno + ((bp->b_bcount-bp->b_resid) >> 9)
                + sn_sizes[logical(bp->b_dev)].sn_offset);
        else printf("\n");
    }
    blkacty &= ~(1<<SN1);
    sntab.b_active = 0;
    sntab.b_errcnt = 0;
    sntab.b_actf = bp->av_forw;
    iodone(bp);
} else if (cnt && bp->b_flags & B_READ) {
    addr = bp->b_un.b_addr + bp->b_bcount - bp->b_resid;
    i = 511;
    do {
        *pm++; /* cmos ram, every other byte */
        *addr++ = *pm++;
    } while (--i != -1);
    if (!(bp->b_blkno + ((bp->b_bcount-bp->b_resid) >> 9)
        + sn_sizes[logical(bp->b_dev)].sn_offset)) { /* bn=0 */
        register caddr_t pm;
        register unsigned short fileid;

        pm = (char *)SN_HDRBUF;
        fileid = *(pm+9)<<8;
        fileid |= *(pm+11);
        if (fileid == FILEID) {
            sn_bcount[boot(physical(bp->b_dev))] = 1;
#ifdef UNISOFT
            sn_bcount[boot(physical(bp->b_dev))] = 0;
#endif UNISOFT
        }
    }
    bp->b_resid -= cnt;
    SNIOS->mask = SN_CLEARMSK;
    SNIOS->gobyte=SN_CLRST;
    snstart();
}

snwaitrty()
{
    register int i, j, k;

    k = 1024;
    do {
        i = 1000;
        while (((PPADDR)->d_ibr & DSKDIAG) == 0) { /* floppy is busy */
            for(j=0;j<1024;j++); /* don't access flpy */
            if (--i < 0) {
                printf("Unix snwaitrty: DSKDIAG not ready\n");
                return(1);
            }
        }
        if (SNIOS->gobyte == 0)
            return(0);
    } while (--k);
    printf("Unix snwaitrty: drive not ready\n");
    return (1);
}

```

```
}

snprint(dev, str)
char *str;
{
    printf("%s on sn drive %d, slice %d\n", str, (dev>>4)&0xF, dev&0x7);
}

/* This is the eject driver
 * which uses a different major device so
 * it can tell whether the sony is being used or not
 */

/* ARGSUSED */
ejioctl(dev, cmd, adr, flag)
dev_t dev;
caddr_t adr;
{
    int unit;
    register struct buf *bp;

    if ((unit = physical(dev)) >= NSN) {
        u.u_error = EFAULT;
        return;
    }
    bp = &snbuf;
    if ( cmd != AL_EJECT ) {
        u.u_error = ENOTTY;
        return;
    }
    if ((sn_bcount[count(unit)] != 0) || (sn_ccount[count(unit)] != 0) ) {
        u.u_error = EINVAL;
        return;
    }
    GETBUF(bp);
    bp->b_dev = dev;
    bp->b_resid = cmd;          /* stash the command in resid */
    snstrategy(bp);
    iowait((struct buf *)bp);
    if (bp->b_flags & B_ERROR)
        u.u_error = EIO;
    FREEBUF(bp);
    return;
}
```

```

/* (#)subr.c 1.4 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/inode.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/buf.h"
#include "sys/mount.h"
#include "sys/var.h"

/*
 * Bmap defines the structure of file system storage
 * by returning the physical block number on a device given the
 * inode and the logical block number in a file.
 * When convenient, it also leaves the physical
 * block number of the next block of the file in u.u_rablock
 * for use in read-ahead.
 */
daddr_t
bmap(ip, readflg)
register struct inode *ip;
{
    register struct user *up;
    register i;
    register dev;
    daddr_t bn;
    daddr_t nb, *bap;
    int raflag;

    up = &u;
    up->u_rablock = 0;
    raflag = 0;
    {
        register sz, rem, type;

        type = ip->i_mode&IFMT;
        if (type == IFBLK) {
            dev = (dev_t)ip->i_rdev;
            for (i=0; i<v.v_mount; i++)
                if ((mount[i].m_flags==MINUSE) &&
                    (brdev(mount[i].m_dev)==brdev(dev))) {
                    dev = mount[i].m_dev;
                    break;
                }
        } else
            dev = ip->i_dev;
        up->u_pbdev = dev;
        bn = FsBNO(dev, up->u_offset);
        if (bn < 0) {
            up->u_error = EFBIG;
            return((daddr_t)-1);
        }
        if ((ip->i_lastr + 1) == bn)
            raflag++;
        up->u_pboff = FsBOFF(dev, up->u_offset);
        sz = FsBSIZE(dev) - up->u_pboff;
        if (up->u_count < sz) {
            sz = up->u_count;
            raflag = 0;
        } else
            ip->i_lastr = bn;

        up->u_pbsize = sz;
        if (type == IFBLK) {
            if (raflag)
                up->u_rablock = bn + 1;
            return(bn);
        }
        if (readflg) {
            if (type == IFIFO) {
                raflag = 0;
                rem = ip->i_size;
            } else
                rem = ip->i_size - up->u_offset;
            if (rem < 0)
                rem = 0;
            if (rem < sz)
                sz = rem;
            if ((up->u_pbsize = sz) == 0)
                return((daddr_t)-1);
        } else {
            if (bn >= FsPTOL(dev, up->u_limit)) {
                up->u_error = EFBIG;
                return((daddr_t)-1);
            }
        }
    }
    register struct buf *bp;
    register j, sh;

    /*
     * blocks 0..NADDR-4 are direct blocks
     */
    if (bn < NADDR-3) {
        i = bn;
        nb = ip->i_addr[i];
        if (nb == 0) {
            if (readflg || (bp = alloc(dev))==NULL)
                return((daddr_t)-1);
            nb = FsPTOL(dev, bp->b_blkno);
            bwrite(bp);
            ip->i_addr[i] = nb;
            ip->i_flag |= IUPD|ICHG;
        }
        if ((i < NADDR-4) && raflag)
            up->u_rablock = ip->i_addr[i+1];
        return(nb);
    }

    /*
     * addresses NADDR-3, NADDR-2, and NADDR-1
     * have single, double, triple indirect blocks.
     * the first step is to determine
     * how many levels of indirection.
     */
    sh = 0;
    nb = 1;
    bn -= NADDR-3;
    for(j=3; j>0; j--) {
        sh += FsNSHIFT(dev);
        nb <<= FsNSHIFT(dev);
        if (bn < nb)
            break;
        bn -= nb;
    }
    if (j == 0) {
        up->u_error = EFBIG;
        return((daddr_t)-1);
    }

    /*
     * fetch the address from the inode
     */
    nb = ip->i_addr[NADDR-j];
    if (nb == 0) {
        if (readflg || (bp = alloc(dev))==NULL)
            return((daddr_t)-1);
        nb = FsPTOL(dev, bp->b_blkno);
        bwrite(bp);
        ip->i_addr[NADDR-j] = nb;
        ip->i_flag |= IUPD|ICHG;
    }

    /*
     * fetch through the indirect blocks
     */

```

```

for(; j<=3; j++) {
    bp = bread(dev, nb);
    if (up->u_error) {
        brelse(bp);
        return((daddr_t)-1);
    }
    bap = bp->b_un.b_daddr;
    sh = FsNSHIFT(dev);
    i = (bn>>sh) & FSNMASK(dev);
    nb = bap[i];
    if (nb == 0) {
        register struct buf *nbp;

        if (readflg || (nbp = alloc(dev))==NULL) {
            brelse(bp);
            return((daddr_t)-1);
        }
        nb = FsPTOL(dev, nbp->b_blkno);
        if (j < 3)
            bwrite(nbp);
        else
            bdwrite(nbp);
        bap[i] = nb;
        bdwrite(bp);
    } else
        brelse(bp);
}

/*
 * calculate read-ahead.
 */
if ((i < FsNINDIR(dev)-1) && raflag)
    up->u_rablock = bap[i+1];
return(nb);
}

/*
 * Pass back c to the user at his location u_base;
 * update u_base, u_count, and u_offset. Return -1
 * on the last character of the user's read.
 * u_base is in the user data space.
 */
passc(c)
register c;
{
    register struct user *up;

    up = &u;
    if (subyte(up->u_base, c) < 0) {
        up->u_error = EFAULT;
        return(-1);
    }
    up->u_count--;
    up->u_offset++;
    up->u_base++;
    return(up->u_count == 0? -1: 0);
}

/*
 * Pick up and return the next character from the user's
 * write call at location u_base;
 * update u_base, u_count, and u_offset. Return -1
 * when u_count is exhausted.
 * u_base is in the user data space.
 */
cpass()
{
    register struct user *up;
    register c;

    up = &u;
    if (up->u_count == 0)
        return(-1);
    if ((c = fubyte(up->u_base)) < 0) {
        up->u_error = EFAULT;

```

```

        return(-1);
    }
    up->u_count--;
    up->u_offset++;
    up->u_base++;
    return(c);
}

```

```

/*
 * Routine which sets a user error; placed in
 * illegal entries in the bdevsw and cdevsw tables.
 */

```

```

nodev()
{
    u.u_error = ENODEV;
}

```

```

/*
 * Null routine; placed in insignificant entries
 * in the bdevsw and cdevsw tables.
 */

```

```

nulldev()
{
}

```

```

/*
 * Max function
 */

```

```

max(a, b)
unsigned a, b;
{
    if (a >= b)
        return(a);
    return(b);
}

```

```

/*
 * Min function
 */

```

```

min(a, b)
unsigned a, b;
{
    if (a <= b)
        return(a);
    return(b);
}

```



```
/* l(#)sys.c 1.2 */
/*
 * indirect driver for controlling tty.
 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/conf.h"
#include "sys/proc.h"

/* ARGSUSED */
syopen(dev, flag)
{
    if (sycheck())
        (*cdevsw[(short)major(u.u_ttyd)].d_open)(minor(u.u_ttyd), flag);
}

/* ARGSUSED */
syread(dev)
{
    if (sycheck())
        (*cdevsw[(short)major(u.u_ttyd)].d_read)(minor(u.u_ttyd));
}

/* ARGSUSED */
sywrite(dev)
{
    if (sycheck())
        (*cdevsw[(short)major(u.u_ttyd)].d_write)(minor(u.u_ttyd));
}

/* ARGSUSED */
syioctl(dev, cmd, arg, mode)
{
    if (sycheck())
        (*cdevsw[(short)major(u.u_ttyd)].d_ioctl)(minor(u.u_ttyd), cmd, arg, mode);
}

sycheck()
{
    if (u.u_ttyp == NULL) {
        u.u_error = ENXIO;
        return(0);
    }
    if (*u.u_ttyp != u.u_procp->p_pgrp) {
        u.u_error = EIO;
        return(0);
    }
    return(1);
}
```

```

/* #define HOWFAR */

/* @(#)sys1.c 1.8 */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/map.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/proc.h"
#include "sys/context.h"
#include "sys/buf.h"
#include "sys/req.h"
#include "sys/file.h"
#include "sys/inode.h"
#include "sys/seg.h"
#include "sys/acct.h"
#include "sys/sysinfo.h"
#include "sys/var.h"
#include "sys/ipc.h"
#include "sys/shm.h"
#include "sys/scat.h"

/*
 * exec system call, with and without environments.
 */
struct execa {
    char *fname;
    char **argp;
    char **envp;
};

exec()
{
    ((struct execa *)u.u_ap)->envp = NULL;
    exece();
}

#define NCABLK (NCARGS+BSIZE-1)/BSIZE
exece()
{
    register unsigned nc;
    register char *cp;
    register struct buf *bp;
    register struct execa *uap;
    register struct user *up;
    int na, ne, ucp, ap, c, bno;
    struct inode *ip;
    extern struct inode *gethead();
    char *namep;
    extern int (*putchar)();

    sysinfo.sysexec++;
    if ((ip = gethead()) == NULL)
        return;
    up = &u;
    bp = 0; na = 0; nc = 0; ne = 0;
    uap = (struct execa *)up->u_ap;
    /* collect arglist */
    if ((bno = swapalloc(NCABLK, 0)) == 0) {
        printf("No swap space for exec args\n");
        iput(ip);
        up->u_error = ENOMEM;
        return;
    }
    if (uap->argp) for (;;) {
        ap = NULL;
        if (uap->argp) {
            ap = fuword((caddr_t)uap->argp);
            uap->argp++;
        }
        if (ap == NULL && uap->envp) {
            uap->argp = NULL;
            if ((ap = fuword((caddr_t)uap->envp)) == NULL)
                break;
            uap->envp++;
            ne++;
        }
        if (ap == NULL)
            break;
        na++;
        if (ap == -1)
            up->u_error = EFAULT;
        do {
            if (nc >= NCARGS-1)
                up->u_error = E2BIG;
            if ((c = fubyte((caddr_t)ap)) < 0)
                up->u_error = EFAULT;
            if (up->u_error)
                goto bad;
            if ((nc & BMASK) == 0) {
                if (bp)
                    bwrite(bp);
                bp = getblk(swapdev,
                    (daddr_t)(swplo+bno+(nc>>BSHIFT)));
                cp = bp->b_un.b_addr;
            }
            nc++;
            *cp++ = c;
        } while (c > 0);
        if (bp)
            bwrite(bp);
        bp = 0;
        nc = (nc + NBPW-1) & ~(NBPW-1);
        getxfile(ip, (int)(nc + sizeof(char *)*na));
        if (up->u_error) {
            printf("exec error: u_error %d u_dent.d_name ", up->u_error);
            for (namep = &up->u_dent.d_name[0]; namep && *namep && namep < &up->u_dent.d_name[DIRS];
                (*putchar)(*namep);
                (*putchar)('\n');
                psignal(up->u_proc, SIGKILL);
                goto bad;
            }
        }

        /* copy back arglist */
        ucp = v.v_uend - nc - NBPW;
        ap = ucp - na*NBPW - 3*NBPW;
        up->u_ar0[SP] = ap;
#ifdef HOWFAR
        printf("Setting new stack pointer to 0x%x\n", ap);
#endif
        (void) suword((caddr_t)ap, na-ne);
        nc = 0;
        for (;;) {
            ap += NBPW;
            if (na == ne) {
                ap += NBPW;
            }
            if (--na < 0)
                break;
            (void) suword((caddr_t)ap, ucp);
            do {
                if ((nc & BMASK) == 0) {
                    if (bp) {
                        bp->b_flags |= B_AGE;
                        bp->b_flags &= ~B_DELWRI;
                        brelse(bp);
                    }
                    bp = bread(swapdev,
                        (daddr_t)(swplo+bno+(nc>>BSHIFT)));
                    bp->b_flags |= B_AGE;
                    bp->b_flags &= ~B_DELWRI;
                    cp = bp->b_un.b_addr;
                }
            }
        }
    }
}

```

```

        (void) subyte((caddr_t)ucp++, (c = *cp++));
        nc++;
    } while (c&0377);
}
(void) suword((caddr_t)ap, 0);
if ((long)ucp & 1)
    (void) subyte((caddr_t)ucp++, 0);
(void) suword((caddr_t)ucp, 0);
setregs();
if (bp) {
    bp->b_flags |= B_AGE;
    bp->b_flags &= ~B_DELRWI;
    brelse(bp);
}
iput(ip);
mfree(swapmap, NCABLK, bno);
return;
bad:
if (bp)
    brelse(bp);
iput(ip);
for (nc = 0; nc < NCABLK; nc++) {
    bp = getblk(swapdev, (daddr_t)(swplo+bno+nc));
    bp->b_flags |= B_AGE;
    bp->b_flags &= ~B_DELRWI;
    brelse(bp);
}
mfree(swapmap, NCABLK, bno);
}

struct inode *
gethead()
{
    register struct exdata *ep;
    register struct inode *ip;
    register unsigned ds, ts;
    register struct user *up;
    struct exdata exdata;

    struct naout {
        short magic;
        short vstamp;
        long tsize;
        long dsize;
        long bsize;
        entry;
        ts;
        ds;
    };

    struct filhd {
        unsigned short magic;
        unsigned short nscns;
        long timdat;
        symptr;
        nsyms;
        unsigned short ophdr;
        flags;
    };

    struct scnhdr {
        char s_name[8];
        long s_paddr;
        long s_vaddr;
        long s_size;
        long s_scnptr;
        long s_relptr;
        long s_lnnoptr;
        unsigned short s_nreloc;
        long s_hinno;
        long s_flags;
    };

    if ((ip = namei(uchar, 0)) == NULL)
        return(NULL);
    up = &u;

```

```

if (access(ip, IEXEC) ||
    (ip->i_mode & IFMT) != IFREG ||
    (ip->i_mode & (IEXEC|(IEXEC>>3)|(IEXEC>>6))) == 0) {
    up->u_error = EACCES;
    goto bad;
}
/*
 * read in first few bytes of file for segment sizes
 * ux_mag = 407/410/411
 * 407 is plain executable
 * 410 is RO text
 * 411 is separated ID
 * 570 Common object
 * 575 "
 * set ux_tstart to start of text portion
 */
ep = &exdata;
up->u_base = (caddr_t)ep;
up->u_count = sizeof(*ep);
up->u_offset = 0;
up->u_segflg = 1;
readi(ip);
#ifdef notdef
if (ep->ux_mag == 0570 || ep->ux_mag == 0575) {
    up->u_base = (caddr_t)ep;
    up->u_count = sizeof(*ep);
    up->u_offset = sizeof(struct filhd);
    up->u_segflg = 1;
    readi(ip);
    ep->ux_tstart = sizeof(struct naout) +
        sizeof(struct filhd) + (3 * sizeof(struct scnhdr));
    ep->ux_entloc = ep->ux_ssize;
} else {
    ep->ux_tstart = sizeof(up->u_exdata);
}
#endif
up->u_segflg = 0;
if (up->u_count!=0)
    ep->ux_mag = 0;
if (ep->ux_mag == 0407) {
    ds = btoc((long)ep->ux_tsize +
        (long)ep->ux_dsize + (long)ep->ux_bsize);
    ts = 0;
    ep->ux_dsize += ep->ux_tsize;
    ep->ux_tsize = 0;
} else {
    ts = btoc(ep->ux_tsize);
    ds = btoc(ep->ux_dsize+ep->ux_bsize);
    if ((ip->i_flag&ITEXT)==0 && ip->i_count!=1) {
        register struct file *fp;

        for (fp = file; fp < (struct file *)v.va_file; fp++)
            if (fp->f_count && fp->f_inode == ip &&
                (fp->f_flag&FWRITE)) {
                up->u_error = ETXTBSY;
                goto bad;
            }
    }
    if (ep->ux_mag != 0410) {
        up->u_error = ENOEXEC;
        goto bad;
    }
}
(void) chksize(ts, ds, (unsigned)(SSIZE+btoc(NCARGS-1)));
up->u_exdata = exdata;
bad:
if (up->u_error) {
    iput(ip);
    ip = NULL;
}
return(ip);
}
/*
 * Read in and set up memory for executed file.
 */

```

```

getxfile(ip, nargc)
register struct inode *ip;
{
    register struct exdata *ep;
    register struct user *up;
    register struct proc *p;
    register unsigned ts, ds, ss;
    register i;

    up = &u;
    p = up->u_proc;
    ep = &up->u_exdata;
#ifdef HOWFAR
    printf("getxfile:reading in program\n");
#endif
    shmexec();
    (void) punlock();
    xfree();
    ts = btoc(ep->ux_tsize);
    ds = btoc(ep->ux_dsize + ep->ux_bsize);
    ss = SSIZE + v.v_usize + btoc(nargc);
    i = v.v_usize+ds+ss;
    expand(i);
    (void) estabur((unsigned)0, ds, ss, 0, RO);
    procclear(p);
    /*
    while (--i >= v.v_usize)
        clearseg((int) (p->p_addr+i));
    */
    xalloc(ip);
    up->u_prof.pr_scale = 0;

    /*
    * read in data segment
    */
    (void) estabur(ts, ds, (unsigned)0, 0, RO);
    if (v.v_doffset)
        up->u_base = (caddr_t)v.v_doffset;
    else
        up->u_base = (caddr_t)(v.v_ustart + ctob(stoc(ctos(ts))));
    up->u_offset = sizeof(up->u_exdata) + ep->ux_tsize;
    up->u_count = ep->ux_dsize;
    readi(ip);
    if (up->u_count!=0)
        up->u_error = EFAULT;

    /*
    * set SUID/SGID protections, if no tracing
    */
    if ((p->p_flag&STRC)==0) {
        if (ip->i_mode&ISUID)
            up->u_uid = ip->i_uid;
        if (ip->i_mode&ISGID)
            up->u_gid = ip->i_gid;
    } else
        p->p_suid = up->u_uid;

    psignal(p, SIGTRAP);
    up->u_tsize = ts;
    up->u_dsize = ds;
    up->u_ssize = ss;
    (void) estabur(ts, ds, ss, 0, RO);
}

/*
 * Clear registers on exec
 */
setregs()
{
    register struct user *up;
    register char *cp;
    register int *rp;
    register i;

    up = &u;
    for (rp = &up->u_signal[0]; rp < &up->u_signal[NSIG]; rp++)

```

```

        if ((*rp & 1) == 0)
            *rp = 0;
    for (cp = &regloc[0]; cp < &regloc[15]; )
        up->u_ar0[*cp++] = 0;
    up->u_ar0[PC] = up->u_exdata.ux_entloc & ~0L;
#ifdef HOWFAR
    printf("New pc = 0x%x\n", up->u_ar0[PC]);
#endif

    for (i=0; i<NOFILE; i++) {
        if ((up->u_pofile[i]&EXCLOSE) && up->u_ofile[i] != NULL) {
            closef(up->u_ofile[i]);
            up->u_ofile[i] = NULL;
        }
    }
    /*
    * Remember file name for accounting.
    */
    up->u_acflag &= ~AFORK;
    bcopy((caddr_t)up->u_dent.d_name, (caddr_t)up->u_comm, DIRSIZ);
}

/*
 * clear the data space for a process
 */
#ifdef NONSCATLOAD
procclear(p)
struct proc *p;
{
    register i, a;

    a = p->p_addr;
    i = p->p_size;
    while(--i >= v.v_usize)
        clearseg(a+i);
}
#else
procclear(p)
struct proc *p;
{
    register struct scatter *s;
    register i, al;

    s = scatmap;
    al = p->p_scat;
    for (i=0; i<v.v_usize; i++)
        al = s[al].sc_index;
    while (al != SCATEND) {
        clearseg(ixtoc(al));
        al = s[al].sc_index;
    }
}
#endif

/*
 * exit system call:
 * pass back caller's arg
 */
rexit()
{
    register struct a {
        int rval;
    } *uap;

    uap = (struct a *)u.u_ap;
    exit((uap->rval & 0377) << 8);
}

/*
 * Release resources.
 * Enter zombie state.
 * Wake up parent and init processes,
 * and dispose of children.
 */
exit(rv)
{

```

```

register struct user *up;
register int i;
register struct proc *p, *q;
#ifdef mc68881 /* MC68881 floating-point coprocessor */
extern short fp881; /* is there an MC68881? */
#endif

up = &u;
p = up->u_procp;
p->p_flag &= ~(STRC);
p->p_clktim = 0;
for (i=0; i<NSIG; i++)
    up->u_signal[i] = 1;
expand(v.v_usize);
(void) estabur((unsigned)0, (unsigned)0, (unsigned)0, 0, RO);
if ((p->p_pid == p->p_pgrp)
    && (up->u_ttyp != NULL)
    && (*up->u_ttyp == p->p_pgrp)) {
    *up->u_ttyp = 0;
    signal(p->p_pgrp, SIGHUP);
}
p->p_pgrp = 0;
for (i=0; i<NOFILE; i++) {
    if (up->u_ofile[i] != NULL)
        closef(up->u_ofile[i]);
}
(void) punlock();
plock(up->u_cdir);
iput(up->u_cdir);
if (up->u_rdir) {
    plock(up->u_rdir);
    iput(up->u_rdir);
}

#ifdef FLOAT /* sky floating point board */
/*
 * If this process was using the SKY FFP, restore
 * the board to a known state.
 */
if (up->u_fpinuse)
    savfp();
#endif

#ifdef mc68881 /* MC68881 floating-point coprocessor */
/*
 * If there is an MC68881, save the internal state and user
 * registers so they'll be available in a core image.
 * Then reset the coprocessor by restoring it to a null state.
 */
if (fp881) {
    fpsave();
    fpreset();
}
#endif

/*
 * call OEM supplied subroutine on process exit
 */
oemexit(p);

xfree();

semexit();
shmexit();

acct(rv);
#ifdef NONSCATLOAD
mfree(coremap, p->p_size, p->p_addr);
#else
memfree(p->p_scat);
p->p_scat = SCATEND;
#endif

cxrelse(p->p_context);
p->p_stat = SZOMB;
((struct xproc *)p)->xp_xstat = rv;
((struct xproc *)p)->xp_utime = up->u_cutime + up->u_utime;
((struct xproc *)p)->xp_stime = up->u_cstime + up->u_stime;

```

```

for (q = &proc[1]; q < (struct proc *)v.ve_proc; q++) {
    if (p->p_pid == q->p_ppid) {
        q->p_ppid = 1;
        if (q->p_stat == SZOMB)
            psignal(&proc[1], SIGCLD);
        if (q->p_stat == SSTOP)
            setrun(q);
    } else
        if (p->p_ppid == q->p_pid)
            psignal(q, SIGCLD);
        if (p->p_pid == q->p_pgrp)
            q->p_pgrp = 0;
}
#ifdef NONSCATLOAD
resume(proc[0].p_addr, up->u_qsav);
#else
resume(ixtoc(proc[0].p_scat), up->u_qsav);
#endif
/* no deposit, no return */
}

/*
 * Wait system call.
 * Search for a terminated (zombie) child,
 * finally lay it to rest, and collect its status.
 * Look also for stopped (traced) children,
 * and pass back status from them.
 */
wait()
{
    register struct user *up;
    register f;
    register struct proc *p;

    up = &u;

loop:
    f = 0;
    for (p = &proc[1]; p < (struct proc *)v.ve_proc; p++)
        if (p->p_ppid == up->u_procp->p_pid) {
            f++;
            if (p->p_stat == SZOMB) {
                freeproc(p, 1);
                return;
            }
            if (p->p_stat == SSTOP) {
                if ((p->p_flag&SWTED) == 0) {
                    p->p_flag |= SWTED;
                    up->u_rval1 = p->p_pid;
                    up->u_rval2 = (fsig(p)<<8) | 0177;
                    return;
                }
                continue;
            }
        }
    if (f) {
        (void) sleep((caddr_t)up->u_procp, PWAIT);
        goto loop;
    }
    up->u_error = ECHILD;
}

/*
 * Remove zombie children from the process table.
 */
freeproc(p, flag)
register struct proc *p;
{
    register struct user *up;

    up = &u;
    if (flag) {
        register n;

        n = up->u_procp->p_cpu + p->p_cpu;
        if (n > 80)
            n = 80;
    }
}

```

```

up->u_procp->p_cpu = n;
up->u_rvall = p->p_pid;
up->u_rval2 = ((struct xproc *)p)->xp_xstat;
}
up->u_cutime += ((struct xproc *)p)->xp_utime;
up->u_cstime += ((struct xproc *)p)->xp_stime;
p->p_stat = NULL;
p->p_pid = 0;
p->p_ppid = 0;
p->p_sig = 0L;
p->p_flag = 0;
p->p_wchan = 0;
}
/*
 * fork system call.
 */
fork()
{
    register struct user *up;
    register a, i;
    int sz, xsize[NSCATSWAP], xaddr[NSCATSWAP];

    up = &u;
    sysinfo.sysfork++;
    /*
     * Disallow if
     * No processes at all;
     * not su and too many procs owned; or
     * not su and would take last slot; or
     * not su and no space on swap.
     * Part of check done in newproc().
     */
    if (up->u_uid && up->u_ruid) {
        if ((a = malloc(swapmap, ctod(maxmem))) == NULL) {
            sz = ctod(maxmem);
            for (i = 0; i < NSCATSWAP; i++)
                if (sz == 0) {
                    xsize[i] = 0;
                    break;
                } else {
                    a = MIN(mallocl(swapmap), sz);
                    xsize[i] = a;
                    xaddr[i] = malloc(swapmap, a);
                    sz -= a;
                }
            for (i = 0; i < NSCATSWAP; i++)
                if (xsize[i] == 0)
                    break;
                else
                    mfree(swapmap, xsize[i], xaddr[i]);
            if (sz != 0) {
                printf("Not enough swap space to fork\n");
                up->u_error = ENOMEM;
                goto out;
            }
        } else
            mfree(swapmap, ctod(maxmem), a);
    }
    switch( newproc(1) ) {
        case 1: /* child -- successful newproc */
            up->u_rvall = up->u_procp->p_ppid;
            up->u_rval2 = 1; /* child */
            up->u_start = time;
            up->u_ticks = lbolt;
            up->u_mem = v.v_usize + procsz(up->u_procp);
            up->u_ior = 0;
            up->u_iow = 0;
            up->u_ioch = 0;
            up->u_cstime = 0;
            up->u_stime = 0;
            up->u_cutime = 0;
            up->u_utime = 0;
            up->u_acflag = AFORK;
            return;
        case 0: /* parent -- successful newproc */

```

```

/* up->u_rvall = pid-of-child: */
break;
default: /* unsuccessful newproc */
    up->u_error = EAGAIN;
    break;
}
out:
    up->u_rval2 = 0; /* parent */
    up->u_ar0[PC] += 2;
}
/*
 * break system call.
 * -- bad planning: "break" is a dirty word in C.
 */
#ifdef NONSCATLOAD
sbreak()
{
    register struct user *up;
    struct a {
        unsigned nsiz;
    };
    register n, d, a;
    int i;

    up = &u;
    /*
     * set n to new data size
     * set d to new-old
     * set n to new total size
     */
    if (v.v_doffset)
        n = btoc((int)((struct a *)up->u_ap->nsiz)) -
            btoc(v.v_doffset);
    else {
        n = btoc((int)((struct a *)up->u_ap->nsiz)) -
            btoc(v.v_ustart);
        n -= stoc(ctos(up->u_tsize));
    }
    if (n < 0)
        n = 0;
    d = n - up->u_dsize;
    if (d == 0)
        return;
    n += v.v_usize + up->u_ssize;
    if (chksize(up->u_tsize, up->u_dsize + d, up->u_ssize))
        return;
    up->u_dsize += d;
    (void) estabur(up->u_tsize, up->u_dsize, up->u_ssize, 0, RO);
    if (d > 0)
        goto bigger;
    a = up->u_procp->p_addr + n - up->u_ssize;
    i = n;
    if (d < 0) {
        n = up->u_ssize;
        while (n--) {
            copyseg(a-d, a);
            a++;
        }
    }
    expand(i);
    return;
}
bigger:
    expand(n);
    a = up->u_procp->p_addr + n;
    n = up->u_ssize;
    while (n--) {
        a--;
        copyseg(a-d, a);
    }
    while (d--)
        clearseg(--a);
}
#else

```

```

sbreak()
{
    register struct scatter *s;
    register struct user *up;
    struct a {
        unsigned nsiz;
    };
    register n, d, a1, a2;
    int i;
    short t;

    up = &u;
    /*
     * set n to new data size
     * set d to new-old
     * set n to new total size
     */

    n = btoc((int)((struct a *)up->u_ap->nsiz)) - btoc(v.v_ustart);
    n -= stoc(ctos(up->u_tsize));
    if (n < 0)
        n = 0;
    d = n - up->u_dsize;
    if (d == 0)
        return;
    n += v.v_usize+up->u_ssize;
    if (chksize(up->u_tsize, up->u_dsize+d, up->u_ssize))
        return;
    s = scatmap;
    up->u_dsize += d;
    if (d > 0)
        goto bigger;
    nscatfree -= d; /* note: d is negative */
    up->u_procp->p_size = n;
    a1 = up->u_procp->p_scat;
    n = up->u_dsize + v.v_usize;
    for (i=1; i<n; i++)
        a1 = s[a1].sc_index;
    a2 = a1;
    while (d++ < 0)
        a2 = s[a2].sc_index;
    t = scatfreelist.sc_index;
    scatfreelist.sc_index = s[a1].sc_index;
    s[a1].sc_index = s[a2].sc_index;
    s[a2].sc_index = t;
    (void) estabur(up->u_tsize, up->u_dsize, up->u_ssize, 0, RO);
#ifdef OLD
    a = up->u_procp->p_addr + n - up->u_ssize;
    i = n;
    if (d < 0) {
        n = up->u_ssize;
        while (n--) {
            copyseg(a-d, a);
            a++;
        }
    }
    expand(i);
#endif
    up->u_procp->p_flag &= ~SCONTIG;
    return;

bigger:
    expand(n);
    a1 = up->u_procp->p_scat;
    /*
     * find last index of original data space
     */
    n = up->u_dsize + v.v_usize - d;
    if (n == 0)
        printf("sbreak:original size is zero\n");
    for (i=1; i<n; i++)
        a1 = s[a1].sc_index;
    /*
     * move stack if necessary
     */
    if (up->u_ssize != 0 && (int)(up->u_dsize-d) <= 0)

```

```

        printf("sbreak:bigger: unusual condition #1\n");
    if (up->u_ssize == 0) {
        while (d-- > 0)
            clearseg(ixtoc(a1 = s[a1].sc_index));
        if (s[a1].sc_index != SCATEND)
            printf("sbreak:not at end of list\n");
    } else {
        /*
         * find end of original stack space
         */
        a2 = a1;
        for (i=0; i<up->u_ssize; i++)
            a2 = s[a2].sc_index;
        t = s[a1].sc_index;
        s[a1].sc_index = s[a2].sc_index;
        s[a2].sc_index = SCATEND;
        while (d-- > 0)
            clearseg(ixtoc(a1 = s[a1].sc_index));
        s[a1].sc_index = t;
    }
    (void) estabur(up->u_tsize, up->u_dsize, up->u_ssize, 0, RO);
    up->u_procp->p_flag &= ~SCONTIG;
#endif
    a = up->u_procp->p_addr + n;
    n = up->u_ssize;
    while (n--) {
        a--;
        copyseg(a-d, a);
    }
    while (d--)
        clearseg(--a);
#endif
}

```

```

/* @(#)sys2.c 1.6 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/proc.h"
#include "sys/errno.h"
#include "sys/file.h"
#include "sys/inode.h"
#include "sys/sysinfo.h"
#ifdef UCB_NET
#include "net/misc.h"
#include "net/socket.h"
#include "net/socketvar.h"
#endif

/*
 * read system call
 */
read()
{
    sysinfo.sysread++;
    rdwr(FREAD);
}

/*
 * write system call
 */
write()
{
    sysinfo.syswrite++;
    rdwr(FWRITE);
}

/*
 * common code for read and write calls:
 * check permissions, set base, count, and offset,
 * and switch out to readi or writei code.
 */
rdwr(mode)
register mode;
{
    register struct user *up;
    register struct file *fp;
    register struct inode *ip;
    register struct a {
        int     fdes;
        char   *cbuf;
        unsigned count;
    } *uap;
    register int type;

    up = &u;
    uap = (struct a *)up->u_ap;
    fp = getf(uap->fdes);
    if (fp == NULL)
        return;
    if ((fp->f_flag&mode) == 0) {
        up->u_error = EBADF;
        return;
    }
    up->u_base = (caddr_t)uap->cbuf;
    up->u_count = uap->count;
    up->u_segflg = 0;
    up->u_fmode = fp->f_flag;
    ip = fp->f_inode;
    type = ip->i_mode&IFMT;

    /*
     * Fix from ROOT: check for file lock before attempting
     * to lock the inode.
     */
    if (type==IFREG) {
        if ((up->u_fmode&FAPPEND) && (mode == FWRITE))
            fp->f_offset = ip->i_size;
        up->u_offset = fp->f_offset;
        if (ip->i_locklist &&
            locked(1, ip, up->u_offset,
                (off_t)(up->u_offset+up->u_count)))
            return;
    }
#ifdef UCB_NET
    if (fp->f_flag & FSOCKET) {
        if (mode == FREAD)
            u.u_error = soreceive((struct socket *)fp->f_socket, (struct sockaddr *)0);
        else
            u.u_error = sosend((struct socket *)fp->f_socket, (struct sockaddr *)0);
    } else
#endif
    {
        if (type==IFREG || type==IFDIR) {
            plock(ip);
            if ((up->u_fmode&FAPPEND) && (mode == FWRITE))
                fp->f_offset = ip->i_size;
        } else if (type == IFIFO) {
            plock(ip);
            fp->f_offset = 0;
        }
        up->u_offset = fp->f_offset;
        if (mode == FREAD)
            readi(ip);
        else
            writei(ip);
        if (type==IFREG || type==IFDIR || type==IFIFO)
            prele(ip);
        fp->f_offset += uap->count-up->u_count;
    }
    up->u_rvall = uap->count-up->u_count;
    up->u_ioch += (unsigned)up->u_rvall;
    if (mode == FREAD)
        sysinfo.readch += (unsigned)up->u_rvall;
    else
        sysinfo.writech += (unsigned)up->u_rvall;
}

/*
 * open system call
 */
open()
{
    register struct a {
        char   *fname;
        int     mode;
        int     crtmode;
    } *uap;

    uap = (struct a *)u.u_ap;
    copen(uap->mode-FOPEN, uap->crtmode);
}

/*
 * creat system call
 */
creat()
{
    struct a {
        char   *fname;
        int     fmode;
    } *uap;

    uap = (struct a *)u.u_ap;
    copen(FWRITE|FCREAT|PTRUNC, uap->fmode);
}

/*
 * common code for open and creat.
 * Check permissions, allocate an open file structure,
 * and call the device open routine if any.
 */
copen(mode, arg)

```



```

register mode;
{
    register struct user *up;
    register struct inode *ip;
    register struct file *fp;
    int i;

    up = &u;
    if ((mode & (FREAD|FWRITE)) == 0) {
        up->u_error = EINVAL;
        return;
    }
    if (mode & FCREAT) {
        ip = namei(uchar, 1);
        if (ip == NULL) {
            for (i=0; i<NOFILE; i++)
                if (up->u_ofile[i] == NULL)
                    break;
            if (i >= NOFILE) {
                iput(u.u_pdir);
                up->u_error = EMFILE;
            }
            if (up->u_error)
                return;
            ip = maknode(arg&0777&(~ISVTX));
            if (ip == NULL)
                return;
            mode &= ~FTRUNC;
        } else {
            if (ip->i_locklist != NULL &&
                (ip->i_flag&IFMT) == IFREG &&
                locked(2, ip, (long) 0L, (long) (1L<<30))) {
                iput(ip);
                return;
            }
            if (mode & FEXCL) {
                up->u_error = EEXIST;
                iput(ip);
                return;
            }
            mode &= ~FCREAT;
        }
    }
#ifdef VIRTUAL451
    xfree(ip);
#endif
    } else {
        ip = namei(uchar, 0);
        if (ip == NULL)
            return;
    }
    if (!(mode & FCREAT)) {
        if (mode & FREAD)
            (void) access(ip, IREAD);
        if (mode & (FWRITE|FTRUNC)) {
            (void) access(ip, IWRITE);
            if ((ip->i_mode & IFMT) == IFDIR)
                up->u_error = EISDIR;
        }
    }
    if (up->u_error || (fp = falloc(ip, mode & FMASK)) == NULL) {
        iput(ip);
        return;
    }
    if (mode & FTRUNC)
        itrunc(ip);
    prele(ip);
    i = up->u_rvall;
    if (save(up->u_qsav)) { /* catch half-opens */
        if (up->u_error == 0)
            up->u_error = EINTR;
        up->u_ofile[i] = NULL;
    }
#ifdef UCB_NET
    fp->f_flag |= FISUSER;
#endif
    closef(fp);
} else {
    openi(ip, mode);
    if (up->u_error == 0)
        return;
    up->u_ofile[i] = NULL;
    if ((--fp->f_count) <= 0) {
        fp->f_next = ffreelist;
        ffreelist = fp;
    }
    iput(ip);
}
}
/*
 * close system call
 */
close()
{
    register struct file *fp;
    register struct a {
        int fdes;
    } *uap;

    uap = (struct a *)u.u_ap;
    fp = getf(uap->fdes);
    if (fp == NULL)
        return;
    u.u_ofile[uap->fdes] = NULL;
#ifdef UCB_NET /* so sockets close correctly */
    fp->f_flag |= FISUSER;
#endif
    closef(fp);
}
/*
 * seek system call
 */
seek()
{
    register struct file *fp;
    register struct inode *ip;
    register struct a {
        int fdes;
        off_t off;
        int sbase;
    } *uap;
    register struct user *up;

    up = &u;
    uap = (struct a *)up->u_ap;
    fp = getf(uap->fdes);
    if (fp == NULL)
        return;
#ifdef UCB_NET
    if (fp->f_flag & FSOCKET) {
        u.u_error = EPIPE;
        return;
    }
#endif
    ip = fp->f_inode;
    if ((ip->i_mode & IFMT) == IFIFO) {
        up->u_error = EPIPE;
        return;
    }
    if (uap->sbase == 1)
        uap->off += fp->f_offset;
    else if (uap->sbase == 2)
        uap->off += fp->f_inode->i_size;
    else if (uap->sbase != 0) {
        up->u_error = EINVAL;
        psignal(up->u_proc, SIGSYS);
        return;
    }
    if (uap->off < 0) {
        up->u_error = EINVAL;
        return;
    }
}

```

```

        fp->f_offset = uap->off;
        up->u_roff = uap->off;
    }

/*
 * link system call
 */
link()
{
    register struct user *up;
    register struct inode *ip, *xp;
    register struct a {
        char *target;
        char *linkname;
    } *uap;

    up = &u;
    uap = (struct a *)up->u_ap;
    ip = namei(uchar, 0);
    if (ip == NULL)
        return;
    if (ip->i_nlink >= MAXLINK) {
        up->u_error = EMLINK;
        goto outn;
    }
    if ((ip->i_mode&IFMT)==IFDIR && !suser())
        goto outn;
/*
 * Unlock to avoid possibly hanging the namei.
 * Sadly, this means races. (Suppose someone
 * deletes the file in the meantime?)
 * Nor can it be locked again later
 * because then there will be deadly
 * embraces.
 * Update inode first for robustness.
 */
    ip->i_nlink++;
    ip->i_flag |= ICHG|ISYN;
    iupdat(ip, &time, &time);
    prele(ip);
    up->u_dirp = (caddr_t)uap->linkname;
    xp = namei(uchar, 1);
    if (xp != NULL) {
        iput(xp);
        up->u_error = EEXIST;
        goto out;
    }
    if (up->u_error)
        goto out;
    if (up->u_pdir->i_dev != ip->i_dev) {
        iput(up->u_pdir);
        up->u_error = EXDEV;
        goto out;
    }
    wdir(ip);
out:
    plock(ip);
    if (up->u_error) {
        ip->i_nlink--;
        ip->i_flag |= ICHG;
    }
}
outn:
    iput(ip);
    return;
}

/*
 * mknod system call
 */
mknod()
{
    register struct inode *ip;
    register struct a {
        char *fname;
        int fmode;
        int dev;

```

```

    } *uap;

    uap = (struct a *)u.u_ap;
    if ((uap->fmode&IFMT) != IFIFO && !suser())
        return;
    ip = namei(uchar, 1);
    if (ip != NULL) {
        iput(ip);
        u.u_error = EEXIST;
        return;
    }
    if (u.u_error)
        return;
    ip = maknode(uap->fmode);
    if (ip == NULL)
        return;
    switch(ip->i_mode&IFMT) {
    case IFCHR:
    case IFBLK:
        ip->i_rdev = (dev_t)uap->dev;
        ip->i_flag |= ICHG;
    }

    iput(ip);
}

/*
 * access system call
 */
saccess()
{
    register struct user *up;
    register struct a {
        char *fname;
        int fmode;
    } *uap;

    up = &u;
    uap = (struct a *)up->u_ap;
    svuid = up->u_uid;
    svgid = up->u_gid;
    up->u_uid = up->u_ruid;
    up->u_gid = up->u_rgid;
    ip = namei(uchar, 0);
    if (ip != NULL) {
        if (uap->fmode&(IREAD>>6))
            (void) access(ip, IREAD);
        if (uap->fmode&(IWRITE>>6))
            (void) access(ip, IWRITE);
        if (uap->fmode&(IEXEC>>6))
            (void) access(ip, IEXEC);
        iput(ip);
    }
    up->u_uid = svuid;
    up->u_gid = svgid;
}

```

```

/* @(#)sys3.c 1.4 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/mount.h"
#include "sys/ino.h"
#include "sys/buf.h"
#include "sys/filsys.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/inode.h"
#include "sys/file.h"
#include "sys/conf.h"
#include "sys/stat.h"
#include "sys/ttold.h"
#include "sys/var.h"
#ifdef UCB_NET
#include "sys/termio.h"
#include "net/misc.h"
#include "net/socketvar.h"
#endif

/*
 * the fstat system call.
 */
fstat()
{
    register struct file *fp;
    register struct a {
        int fdes;
        struct stat *sb;
    } *uap;

    uap = (struct a *)u.u_ap;
    fp = getf(uap->fdes);
    if (fp == NULL)
        return;
#ifdef UCB_NET
    if (fp->f_flag & FSOCKET)
        u.u_error = sostat((struct socket *)fp->f_socket, uap->sb);
    else
#endif
    statl(fp->f_inode, uap->sb);
}

/*
 * the stat system call.
 */
stat()
{
    register struct inode *ip;
    register struct a {
        char *fname;
        struct stat *sb;
    } *uap;

    uap = (struct a *)u.u_ap;
    ip = namei(uchar, 0);
    if (ip == NULL)
        return;
    statl(ip, uap->sb);
    iput(ip);
}

/*
 * The basic routine for fstat and stat:
 * get the inode and pass appropriate parts back.
 */
statl(ip, ub)
register struct inode *ip;
struct stat *ub;
{
    register struct dinode *dp;

```

```

    register struct buf *bp;
    register struct stat *dsp;
    struct stat ds;

    if (ip->i_flag & (IACC|IUPD|ICHG))
        iupdat(ip, &time, &time);
    /*
     * first copy from inode table
     */
    dsp = &ds;
    dsp->st_dev = brdev(ip->i_dev);
    dsp->st_ino = ip->i_number;
    dsp->st_mode = ip->i_mode;
    dsp->st_nlink = ip->i_nlink;
    dsp->st_uid = ip->i_uid;
    dsp->st_gid = ip->i_gid;
    dsp->st_rdev = (dev_t)ip->i_rdev;
    dsp->st_size = ip->i_size;
    /*
     * next the dates in the disk
     */
    bp = bread(ip->i_dev, FsITOD(ip->i_dev, ip->i_number));
    dp = bp->b_un.b_dino;
    dp += FsITOO(ip->i_dev, ip->i_number);
    dsp->st_atime = dp->di_atime;
    dsp->st_mtime = dp->di_mtime;
    dsp->st_ctime = dp->di_ctime;
    brelse(bp);
    if (copyout((caddr_t)dsp, (caddr_t)ub, sizeof(ds)) < 0)
        u.u_error = EFAULT;
}

/*
 * the dup system call.
 */
dup()
{
    register struct file *fp;
    int i;
    struct a {
        int fdes;
    } *uap;

    uap = (struct a *)u.u_ap;
    fp = getf(uap->fdes);
    if (fp == NULL)
        return;
    if ((i = ufallot(0)) < 0)
        return;
    u.u_ofile[i] = fp;
    fp->f_count++;
}

/*
 * the file control system call.
 */
fcntl()
{
    register struct file *fp;
    register struct a {
        int fdes;
        int cmd;
        int arg;
    } *uap;
    register i;
    register struct user *up;

    up = &u;
    uap = (struct a *)up->u_ap;
    fp = getf(uap->fdes);
    if (fp == NULL)
        return;
    switch(uap->cmd) {
    case 0:
        i = uap->arg;
        if (i < 0 || i > NCFILE) {

```

```

        up->u_error = EINVAL;
        return;
    }
    if ((i = ufalloc(i)) < 0)
        return;
    up->u_ofile[i] = fp;
    fp->f_count++;
    break;

case 1:
    up->u_rvall = up->u_pofile[uap->fdes];
    break;

case 2:
    up->u_pofile[uap->fdes] = uap->arg;
    break;

case 3:
    up->u_rvall = fp->f_flag+FOPEN;
    break;

case 4:
    fp->f_flag &= (FREAD|FWRITE);
    fp->f_flag |= (uap->arg-FOPEN) & ~(FREAD|FWRITE);
    break;

default:
    up->u_error = EINVAL;
}
}

/*
 * character special i/o control
 */
ioctl()
{
    register struct file *fp;
    register struct inode *ip;
    register struct a {
        int    fdes;
        int    cmd;
        int    arg;
    } *uap;
    register dev_t dev;
#ifdef UCB_NET
    register unsigned fmt;
#endif

    uap = (struct a *)u.u_ap;
    if ((fp = getf(uap->fdes)) == NULL)
        return;
#ifdef UCB_NET
    if (fp->f_flag & FSOCKET) {
        solioctl((struct socket *)fp->f_socket, uap->cmd, (caddr_t)uap->arg);
        return;
    }
#endif
    ip = fp->f_inode;
#ifdef UCB_NET
    fmt = ip->i_mode & IFMT;
    if (fmt != IFCHR) {
        if (uap->cmd == FIONREAD && (fmt == IFREG || fmt == IFDIR
            || fmt == IFIFO)) {
            off_t nread;

            if ((ip->i_mode & IFMT) == IFIFO)
                nread = ip->i_size;
            else
                nread = ip->i_size - fp->f_offset;
            if (copyout((caddr_t)&nread, (caddr_t)uap->arg,
                sizeof(off_t)))
                u.u_error = EFAULT;
        } else if (uap->cmd == FIONBIO /*|| uap->cmd == FIOASYNC*/)
            return;
        else
            u.u_error = ENOTTY;
    }

```

```

        return;
    }
}
#else
    if ((ip->i_mode & IFMT) != IFCHR) {
        u.u_error = ENOTTY;
        return;
    }
#endif
#ifdef UCB_NET
    dev = (dev_t)ip->i_rdev;
    (*cdevsw[(short)major(dev)].d_ioctl)(minor(dev), uap->cmd, uap->arg, fp->f_flag);
}

/*
 * old stty and gtty
 */
stty()
{
    register struct a {
        int    fdes;
        int    arg;
        int    narg;
    } *uap;

    uap = (struct a *)u.u_ap;
    uap->narg = uap->arg;
    uap->arg = TIOCESTP;
    ioctl();
}

gtty()
{
    register struct a {
        int    fdes;
        int    arg;
        int    narg;
    } *uap;

    uap = (struct a *)u.u_ap;
    uap->narg = uap->arg;
    uap->arg = TIOCGESTP;
    ioctl();
}

/*
 * the mount system call.
 */
smount()
{
    register struct user *up;
    register dev_t dev;
    register struct inode *ip;
    register struct mount *mp;
    struct mount *smp;
    register struct filsys *fp;
    struct inode *bip = NULL;
    register struct a {
        char    *fspec;
        char    *freg;
        int     ronly;
    } *uap;

    up = &u;
    uap = (struct a *)up->u_ap;
    if (!suser())
        return;
    ip = namei(uchar, 0);
    if (ip == NULL)
        return;
    if ((ip->i_mode & IFMT) != IFBLK)
        up->u_error = ENOTBLK;
    dev = (dev_t)ip->i_rdev;
    if (bmajor(dev) >= bdevcnt)
        if (!up->u_error)
            up->u_error = ENXIO;
    if (up->u_error)
        goto out;

```

```

bip = ip;
up->u_dirp = (caddr_t)uap->freg;
ip = namei(uchar, 0);
if(ip == NULL) {
    iput(bip);
    return;
}
if ((ip->i_mode&IFMT) != IFDIR) {
    up->u_error = ENOTDIR;
    goto out;
}
if (ip->i_count != 1)
    goto out;
if (ip->i_number == ROOTINO)
    goto out;
smp = NULL;
for(mp = &mount[0]; mp < (struct mount *)v.ve_mount; mp++) {
    if(mp->m_flags != MFREE) {
        if (brdev(dev) == brdev(mp->m_dev))
            goto out;
        } else
        if(smp == NULL)
            smp = mp;
    }
mp = smp;
if(mp == NULL)
    goto out;
mp->m_flags = MINTER;
mp->m_dev = brdev(dev);
(*bdevsw[(short)bmajor(dev)].d_open)(minor(dev),
    uap->ronly ? (FREAD | FKERNEL) : (FREAD | FWRITE | FKERNEL));
if(up->u_error)
    goto out1;
mp->m_bufp = getebk();
fp = mp->m_bufp->b_un.b_filsys;
up->u_offset = SUPERBOFF;
up->u_count = sizeof(struct filsys);
up->u_base = (caddr_t)fp;
up->u_segflg = 1;
readi(bip);
if (up->u_error) {
    brelse(mp->m_bufp);
    goto out1;
}
mp->m_inodp = ip;
mp->m_flags = MINUSE;
if (fp->s_magic != FsmAGIC)
    fp->s_type = Fs1b; /* assume old file system */
if (fp->s_type == Fs2b)
    mp->m_dev |= Fs2BLK;
#if FsTYPE == 4
if (fp->s_type == Fs4b)
    mp->m_dev |= Fs4BLK;
#endif
if (brdev(pipedev) == brdev(mp->m_dev))
    pipedev = mp->m_dev;
fp->s_ilock = 0;
fp->s_flock = 0;
fp->s_ninode = 0;
fp->s_inode[0] = 0;
fp->s_ronly = uap->ronly & 1;
if (mp->m_mount = iget(mp->m_dev, ROOTINO))
    prele(mp->m_mount);
else {
    brelse(mp->m_bufp);
    goto out1;
}
ip->i_flag |= IMOUNT;
iput(bip);
prele(ip);
return;
out1:
mp->m_flags = MFREE;
out:
if (bip != NULL)

```

```

    iput(bip);
    if (up->u_error == 0)
        up->u_error = EBUSY;
    iput(ip);
}
/*
 * the umount system call.
 */
sumount()
{
    register dev_t dev;
    register struct inode *ip;
    register struct mount *mp;
    register struct a {
        char *fspec;
    };
    if(!user())
        return;
    dev = getmdev();
    if(u.u_error)
        return;
    for(mp = &mount[0]; mp < (struct mount *)v.ve_mount; mp++)
        if(mp->m_flags == MINUSE && brdev(dev) == brdev(mp->m_dev))
            goto found;
    u.u_error = EINVAL;
    return;
found:
    dev = mp->m_dev;
    (void) xumount(dev); /* remove unused sticky files from text table */
    update();
    if (mp->m_mount) {
        plock(mp->m_mount);
        iput(mp->m_mount);
        mp->m_mount = NULL;
    }
    for(ip = &inode[0]; ip < (struct inode *)v.ve_inode; ip++)
        if(ip->i_number != 0 && dev == ip->i_dev) {
            u.u_error = EBUSY;
            return;
        }
    (*bdevsw[(short)bmajor(dev)].d_close)(minor(dev), FKERNEL);
    binval(dev);
    ip = mp->m_inodp;
    ip->i_flag &= ~IMOUNT;
    plock(ip);
    iput(ip);
    brelse(mp->m_bufp);
    mp->m_bufp = NULL;
    mp->m_flags = MFREE;
}
/*
 * Common code for mount and umount.
 * Check that the user's argument is a reasonable
 * thing on which to mount, and return the device number if so.
 */
dev_t
getmdev()
{
    dev_t dev;
    register struct inode *ip;

    ip = namei(uchar, 0);
    if(ip == NULL)
        return(NODEV);
    if((ip->i_mode&IFMT) != IFBLK)
        u.u_error = ENOTBLK;
    dev = (dev_t)ip->i_rdev;
    if(bmajor(dev) >= bdevcnt)
        u.u_error = ENXIO;

    iput(ip);
    return(dev);
}

```

sys3.c

Fri Sep 5 19:08:03 1986

4

```

/* @(#)sys4.c 1.5 */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/inode.h"
#include "sys/file.h"
#include "sys/filsys.h"
#include "sys/proc.h"
#include "sys/var.h"

/*
 * Everything in this file is a routine implementing a system call.
 */

ptime()
{
    u.u_rtime = time;
}

stime()
{
    register struct a {
        time_t time;
    } *uap;
    struct filsys *fp;

    uap = (struct a *)u.u_ap;
    if (suser()) {
        logtchg(uap->time);
        time = uap->time;
        if (fp = getfs(rootdev))
            fp->s_fmod = 1;
    }
}

setuid()
{
    register unsigned uid;
    register struct a {
        int uid;
    } *uap;
    register struct user *up;

    up = &u;
    uap = (struct a *)up->u_ap;
    uid = uap->uid;
    if (uid >= MAXUID) {
        up->u_error = EINVAL;
        return;
    }
    if (uid && (uid == up->u_ruid || uid == up->u_procp->p_suid))
        up->u_uid = uid;
    else if (suser()) {
        up->u_uid = uid;
        up->u_procp->p_uid = uid;
        up->u_procp->p_suid = uid;
        up->u_ruid = uid;
    }
}

getuid()
{
    register struct user *up;

    up = &u;
    up->u_rval1 = up->u_ruid;
    up->u_rval2 = up->u_uid;
}

```

```

setgid()
{
    register unsigned gid;
    register struct a {
        int gid;
    } *uap;
    register struct user *up;

    up = &u;
    uap = (struct a *)up->u_ap;
    gid = uap->gid;
    if (gid >= MAXUID) {
        up->u_error = EINVAL;
        return;
    }
    if (up->u_rgid == gid || suser()) {
        up->u_gid = gid;
        up->u_rgid = gid;
    }
}

getgid()
{
    register struct user *up;

    up = &u;
    up->u_rval1 = up->u_rgid;
    up->u_rval2 = up->u_gid;
}

getpid()
{
    register struct user *up;

    up = &u;
    up->u_rval1 = up->u_procp->p_pid;
    up->u_rval2 = up->u_procp->p_ppid;
}

setpgrp()
{
    register struct proc *p = u.u_procp;
    register struct a {
        int flag;
    } *uap;

    uap = (struct a *)u.u_ap;
    if (uap->flag) {
        if (p->p_pgrp != p->p_pid)
            u.u_ttyp = NULL;
        p->p_pgrp = p->p_pid;
    }
    u.u_rval1 = p->p_pgrp;
}

sync()
{
    update();
}

nice()
{
    register n;
    register struct a {
        int niceness;
    } *uap;
    register struct user *up;

    up = &u;
    uap = (struct a *)up->u_ap;
    n = uap->niceness;
    if ((n < 0 || n > 2*NZERO) && !suser())
        n = 0;
    n += up->u_procp->p_nice;
    if (n >= 2*NZERO)

```

```

        n = 2*NZERO -1;
    if (n < 0)
        n = 0;
    up->u_procp->p_nice = n;
    up->u_rvall = n - NZERO;
}

/*
 * Unlink system call.
 * Hard to avoid races here, especially
 * in unlinking directories.
 */
unlink()
{
    register struct inode *ip, *pp;
    struct a {
        char *fname;
    };
    register struct user *up;

    up = &u;
    pp = namei(uchar, 2);
    if (pp == NULL)
        return;
    /*
     * Check for unlink(".*")
     * to avoid hanging on the iget
     */
    if (pp->i_number == up->u_dent.d_ino) {
        ip = pp;
        ip->i_count++;
    } else
        ip = iget(pp->i_dev, up->u_dent.d_ino);
    if (ip == NULL)
        goto out1;
    if ((ip->i_mode&IFMT) == IFDIR && !suser())
        goto out;
    /*
     * Don't unlink a mounted file.
     */
    if (ip->i_dev != pp->i_dev) {
        up->u_error = EBUSY;
        goto out;
    }
    if (ip->i_flag&ITEXT)
        xrele(ip); /* try once to free text */
    if (ip->i_flag&ITEXT && ip->i_nlink == 1) {
        up->u_error = ETXTBSY;
        goto out;
    }
    up->u_offset -= sizeof(struct direct);
    up->u_base = (caddr_t)&up->u_dent;
    up->u_count = sizeof(struct direct);
    up->u_dent.d_ino = 0;
    up->u_segflg = 1;
    up->u_fmode = FWRITE|FSYNC;
    write1(pp);
    if (up->u_error)
        goto out;
    ip->i_nlink--;
    ip->i_flag |= ICHG;

out:
    iput(ip);
out1:
    iput(pp);
}

chdir()
{
    chdirec(&u.u_cdir);
}

chroot()
{
    if (!suser())
        return;
    chdirec(&u.u_rdir);
}

register struct inode **ipp;
{
    register struct inode *ip;
    struct a {
        char *fname;
    };

    ip = namei(uchar, 0);
    if (ip == NULL)
        return;
    if ((ip->i_mode&IFMT) != IFDIR) {
        u.u_error = ENOTDIR;
        goto bad;
    }
    if (access(ip, IEXEC))
        goto bad;
    prele(ip);
    if (*ipp) {
        plock(*ipp);
        iput(*ipp);
    }
    *ipp = ip;
    return;

bad:
    iput(ip);
}

chmod()
{
    register struct inode *ip;
    register struct a {
        char *fname;
        int fmode;
    } *uap;

    uap = (struct a *)u.u_ap;
    if ((ip = owner()) == NULL)
        return;
    ip->i_mode &= ~0777;
    if (u.u_uid) {
        uap->fmode &= ~ISVTX;
        if (u.u_gid != ip->i_gid)
            uap->fmode &= ~ISGID;
    }
    ip->i_mode |= uap->fmode&0777;
    ip->i_flag |= ICHG;
    if (ip->i_flag&ITEXT && (ip->i_mode&ISVTX) == 0)
        xrele(ip);
    iput(ip);
}

chown()
{
    register struct inode *ip;
    register struct a {
        char *fname;
        int uid;
        int gid;
    } *uap;

    uap = (struct a *)u.u_ap;
    if ((ip = owner()) == NULL)
        return;
    ip->i_uid = uap->uid;
    ip->i_gid = uap->gid;
    if (u.u_uid != 0)
        ip->i_mode &= ~(ISUID|ISGID);
    ip->i_flag |= ICHG;
    iput(ip);
}

```



```

ssig()
{
    register a;
    register struct proc *p;
    struct a {
        int    signo;
        int    fun;
    } *uap;
    register struct user *up;

    up = &u;
    uap = (struct a *)up->u_ap;
    a = uap->signo;
    if (a <= 0 || a > NSIG || a == SIGKILL) {
        up->u_error = EINVAL;
        return;
    }
    up->u_rvall = up->u_signal[a-1];
    up->u_signal[a-1] = uap->fun;
    up->u_procp->p_sig &= ~(1<<(a-1));
    if (a == SIGCLD) {
        a = up->u_procp->p_pid;
        for (p = &proc[1]; p < (struct proc *)v.ve_procp; p++) {
            if (a == p->p_ppid && p->p_stat == SZOMB)
                psignal(up->u_procp, SIGCLD);
        }
    }
    if (uap->fun&1)
        up->u_procp->p_sigign |= (1<<(uap->signo-1));
    else
        up->u_procp->p_sigign &= ~(1<<(uap->signo-1));
}

kill()
{
    register struct proc *p, *q;
    register arg;
    register struct a {
        int    pid;
        int    signo;
    } *uap;
    int f;
    register struct user *up;

    up = &u;
    uap = (struct a *)up->u_ap;
    if (uap->signo < 0 || uap->signo > NSIG) {
        up->u_error = EINVAL;
        return;
    }
    /* Prevent proc 1 (init) from being SIGKILLed */
    if (uap->signo == SIGKILL && uap->pid == 1) {
        up->u_error = EINVAL;
        return;
    }
    f = 0;
    arg = uap->pid;
    if (arg > 0)
        p = &proc[1];
    else
        p = &proc[2];
    q = up->u_procp;
    if (arg == 0 && q->p_pgrp == 0) {
        up->u_error = ESRCH;
        return;
    }
    for(;; p < (struct proc *)v.ve_procp; p++) {
        if (p->p_stat == NULL)
            continue;
        if (arg > 0 && p->p_pid != arg)
            continue;
        if (arg == 0 && p->p_pgrp != q->p_pgrp)
            continue;
        if (arg < -1 && p->p_pgrp != -arg)
            continue;

```

```

        if (! (up->u_uid == 0 ||
            up->u_uid == p->p_uid ||
            up->u_ruid == p->p_ruid ||
            up->u_uid == p->p_suid ||
            up->u_ruid == p->p_suid ))
            if (arg > 0) {
                up->u_error = EPERM;
                return;
            } else
                continue;
        f++;
        if (uap->signo)
            psignal(p, uap->signo);
        if (arg > 0)
            break;
    }
    if (f == 0)
        up->u_error = ESRCH;
}

times()
{
    register struct a {
        time_t (*times)[4];
    } *uap;
    register struct user *up;
    time_t loctime[4];

    up = &u;
    uap = (struct a *)up->u_ap;
    if (v.v_hz==60) {
        if (copyout((caddr_t)&up->u_time, (caddr_t)uap->times,
            sizeof(*uap->times)))
            up->u_error = EFAULT;
        SPL7();
        up->u_rtime = lbolt;
        SPL0();
    } else {
        loctime[0] = up->u_time * 60 / v.v_hz;
        loctime[1] = up->u_stime * 60 / v.v_hz;
        loctime[2] = up->u_cutime * 60 / v.v_hz;
        loctime[3] = up->u_cstime * 60 / v.v_hz;
        if (copyout((caddr_t)&loctime[0], (caddr_t)uap->times,
            sizeof(loctime)) < 0)
            up->u_error = EFAULT;
        SPL7();
        up->u_rtime = lbolt*60/v.v_hz;
        SPL0();
    }
}

profil()
{
    register struct a {
        short *bufbase;
        unsigned bufsize;
        unsigned pcoffset;
        unsigned pcscale;
    } *uap;
    register struct user *up;

    up = &u;
    uap = (struct a *)up->u_ap;
    up->u_prof.pr_base = uap->bufbase;
    up->u_prof.pr_size = uap->bufsize;
    up->u_prof.pr_off = uap->pcoffset;
    up->u_prof.pr_scale = uap->pcscale;
}

/*
 * alarm clock signal
 */
alarm()
{
    register struct proc *p;
    register c;

```

```

register struct a {
    int    deltat;
} *uap;

uap = (struct a *)u.u_ap;
p = u.u_procp;
c = p->p_clktim;
p->p_clktim = uap->deltat;
u.u_rvall = c;
}

/*
 * indefinite wait.
 * no one should wakeup($u)
 */
pause()
{
    for(;;)
        (void) sleep((caddr_t)&u, PSLEP);
}

/*
 * mode mask for creation of files
 */
umask()
{
    register struct a {
        int    mask;
    } *uap;
    register t;
    register struct user *up;

    up = &u;
    uap = (struct a *)up->u_ap;
    t = up->u_cmask;
    up->u_cmask = uap->mask & 0777;
    up->u_rvall = t;
}

/*
 * Set IUPD and IACC times on file.
 */
utime()
{
    register struct a {
        char    *fname;
        time_t  *tptr;
    } *uap;
    register struct inode *ip;
    time_t tv[2];

    uap = (struct a *)u.u_ap;
    if (uap->tptr != NULL) {
        if (copyin((caddr_t)uap->tptr, (caddr_t)tv, sizeof(tv))) {
            u.u_error = EFAULT;
            return;
        }
    }
    else {
        tv[0] = time;
        tv[1] = time;
    }
    ip = namei(uchar, 0);
    if (ip == NULL)
        return;
    if (u.u_uid != ip->i_uid && u.u_uid != 0) {
        if (uap->tptr != NULL)
            u.u_error = EPERM;
        else
            (void) access(ip, IWRITE);
    }
    if (!u.u_error) {
        ip->i_flag |= IACC|IUPD|ICRG;
        iupdat(ip, &tv[0], &tv[1]);
    }
    iput(ip);
}

```

```

}

ulimit()
{
    register struct a {
        int    cmd;
        long   arg;
    } *uap;
    register struct user *up;
    register n, ts;

    up = &u;
    uap = (struct a *)up->u_ap;
    switch(uap->cmd) {
    case 2:
        if (uap->arg > up->u_limit && !suser())
            return;
        up->u_limit = uap->arg;
    case 1:
        up->u_roff = up->u_limit;
        break;
    case 3:
        ts = up->u_tsize;
        n = maxmem - up->u_ssize - v.v_usize;
        if (ts > minmem)
            n = MAX((int)n-ts, (int)minmem-up->u_ssize-v.v_usize);
        n = MIN(n, MAXMEM-stoc(ctos(ts))-stoc(ctos(up->u_ssize)));
        up->u_roff = v.v_ustart + ctob(stoc(ctos(ts)) + n);
        break;
    default:
        up->u_error = EINVAL;
    }
}

/*
 * phys - Set up a physical address in user's address space.
 */
phys()
{
    register struct a {
        unsigned phnum;    /* phys segment number */
        unsigned laddr;    /* logical address */
        unsigned bcount;  /* byte count */
        unsigned phaddr;  /* physical address */
    } *uap;

    if (!suser()) return;
    uap = (struct a *)u.u_ap;
    dopphys(uap->phnum, uap->laddr, uap->bcount, uap->phaddr);
}

/*
 * reboot the system
 */
reboot()
{
    register i;

    update();
    for (i = 0; i < 1000000; i++)
        ;
    doboot();
}

```

```

/* @(#)sysent.c 1.3 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/system.h"

/*
 * This table is the switch used to transfer
 * to the appropriate routine for processing a system call.
 */

int    alarm();
int    chdir();
int    chmod();
int    chown();
int    chroot();
int    close();
int    creat();
int    dup();
int    exec();
int    exece();
int    fcntl();
int    fork();
int    fstat();
int    getgid();
int    getpid();
int    getuid();
int    gtime();
int    gtty();
int    ioctl();
int    kill();
int    link();
int    lock();
int    mknod();
int    chmod();
int    msgsys();
int    nice();
int    nosys();
int    nullsys();
int    open();
int    pause();
int    pipe();
int    profil();
int    ptrace();
int    read();
int    rexit();
int    success();
int    sbreak();
int    seek();
int    semsys();
int    setgid();
int    setpgrp();
int    setuid();
int    shmsys();
int    smount();
int    ssig();
int    stat();
int    stime();
int    stty();
int    sumount();
int    sync();
int    sysacct();
int    times();
int    ulimit();
int    umask();
int    unlink();
int    utime();
int    utssys();
int    wait();
int    write();

#ifdef UCB_NET
#include "net/misc.h"
/* net stuff */
int    select(); /* not implimented for character devices yet */
int    gethostname();
int    sethostname();
int    ssocket();

```

```

int    sconnect();
int    saccept();
int    ssend();
int    sreceive();
int    ssocketaddr();
int    netreset();
#endif

```

```

/*
 * Local system calls
 */

```

```

int    locking();
int    phys();
int    reboot();

```

```

struct sysent sysent[] =
{

```

```

    nosys,          /* 0 = indir */
    rexit,          /* 1 = exit */
    fork,          /* 2 = fork */
    read,          /* 3 = read */
    write,         /* 4 = write */
    open,          /* 5 = open */
    close,         /* 6 = close */
    wait,          /* 7 = wait */
    creat,         /* 8 = creat */
    link,          /* 9 = link */
    unlink,        /* 10 = unlink */
    exec,          /* 11 = exec */
    chdir,         /* 12 = chdir */
    gtime,         /* 13 = time */
    mknod,         /* 14 = mknod */
    chmod,         /* 15 = chmod */
    chown,         /* 16 = chown; now 3 args */
    sbreak,        /* 17 = break */
    stat,          /* 18 = stat */
    seek,          /* 19 = seek */
    getpid,        /* 20 = getpid */
    smount,        /* 21 = mount */
    sumount,       /* 22 = umount */
    setuid,        /* 23 = setuid */
    getuid,        /* 24 = getuid */
    stime,         /* 25 = stime */
    ptrace,        /* 26 = ptrace */
    alarm,         /* 27 = alarm */
    fstat,         /* 28 = fstat */
    pause,         /* 29 = pause */
    utime,         /* 30 = utime */
    stty,          /* 31 = stty */
    gtty,          /* 32 = gtty */
    success,       /* 33 = access */
    nice,          /* 34 = nice */
    nosys,         /* 35 = sleep; inoperative */
    sync,          /* 36 = sync */
    kill,          /* 37 = kill */
    nosys,         /* 38 = x */
    setpgrp,       /* 39 = setpgrp */
    nosys,         /* 40 = tell - obsolete */
    dup,           /* 41 = dup */
    pipe,          /* 42 = pipe */
    times,         /* 43 = times */
    profil,        /* 44 = prof */
    lock,          /* 45 = proc lock */
    setgid,        /* 46 = setgid */
    getgid,        /* 47 = getgid */
    ssig,          /* 48 = sig */
    msgsys,        /* 49 = IPC Messages */
    nosys,         /* 50 = reserved for local use */
    sysacct,       /* 51 = turn acct off/on */
    shmsys,        /* 52 = IPC Shared Memory */
    semsys,        /* 53 = IPC Semaphores */
    ioctl,         /* 54 = ioctl */
    phys,          /* 55 = phys */
    locking,       /* 56 = file locking */
    utssys,        /* 57 = utssys */
    nosys,         /* 58 = reserved for USG */

```

```

exece,          /* 59 = exece */
umask,         /* 60 = umask */
chroot,       /* 61 = chroot */
fcntl,       /* 62 = fcntl */
ulimit,     /* 63 = ulimit */

reboot,      /* 64 = reboot */
nosys,      /* 65 = x */
nosys,      /* 66 = x */
nosys,      /* 67 = x */
nosys,      /* 68 = x */
nosys,      /* 69 = x */
#ifdef UCB_NET
select,     /* 70 = select */
gethostname, /* 71 = gethostname */
sethostname, /* 72 = sethostname */
socket,     /* 73 = socket */
accept,    /* 74 = accept */
connect,   /* 75 = connect */
receive,   /* 76 = receive */
send,     /* 77 = send */
socketaddr, /* 78 = socketaddr */
netreset, /* 79 = netreset */
#else
nosys,     /* 70 = x */
nosys,     /* 71 = x */
nosys,     /* 72 = x */
nosys,     /* 73 = x */
nosys,     /* 74 = x */
nosys,     /* 75 = x */
nosys,     /* 76 = x */
nosys,     /* 77 = x */
nosys,     /* 78 = x */
nosys,     /* 79 = x */
#endif UCB_NET
nosys,     /* 80 = x */
nosys,     /* 81 = x */
nosys,     /* 82 = x */
nosys,     /* 83 = x */
nosys,     /* 84 = x */
nosys,     /* 85 = x */
nosys,     /* 86 = x */
nosys,     /* 87 = x */
nosys,     /* 88 = x */
nosys,     /* 89 = x */
nosys,     /* 90 = x */
nosys,     /* 91 = x */
nosys,     /* 92 = x */
nosys,     /* 93 = x */
nosys,     /* 94 = x */
nosys,     /* 95 = x */
nosys,     /* 96 = x */
nosys,     /* 97 = x */
nosys,     /* 98 = x */
nosys,     /* 99 = x */
nosys,     /* 100 = x */
nosys,     /* 101 = x */
nosys,     /* 102 = x */
nosys,     /* 103 = x */
nosys,     /* 104 = x */
nosys,     /* 105 = x */
nosys,     /* 106 = x */
nosys,     /* 107 = x */
nosys,     /* 108 = x */
nosys,     /* 109 = x */
nosys,     /* 110 = x */
nosys,     /* 111 = x */
nosys,     /* 112 = x */
nosys,     /* 113 = x */
nosys,     /* 114 = x */
nosys,     /* 115 = x */
nosys,     /* 116 = x */
nosys,     /* 117 = x */
nosys,     /* 118 = x */
nosys,     /* 119 = x */
nosys,     /* 120 = x */
nosys,     /* 121 = x */
nosys,     /* 122 = x */
nosys,     /* 123 = x */
nosys,     /* 124 = x */
nosys,     /* 125 = x */
nosys,     /* 126 = x */
nosys,     /* 127 = x */
};

```

```

#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/mmu.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/proc.h"
#include "sys/seq.h"
#include "sys/signal.h"
#include "sys/errno.h"
#include "sys/user.h"
#include "sys/system.h"
#include "sys/inode.h"
#include "sys/ino.h"
#include "sys/file.h"
#include "sys/conf.h"
#include "net/misc.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "net/ubavar.h"
#include "sys/map.h"
#include "sys/callo.h"
#include "net/if.h"
#include "net/in.h"
#include "net/in_system.h"
#include "net/ip.h"
#include "net/ip_var.h"
#include "sys/var.h"

/*
 * SCCS id @(#)syslocal.c 1.6 (Berkeley) 2/27/82
 */

/*
 * These routines implement local system calls
 */

short netoff = 0;
int protoslow;
int protofast;
int ifnetslow;
int nselcoll;
char netstak[3000]; /* stack when system is "in the network" (mch.s) */
char * svstak; /* glibbe where mch.s saves current stack */

/*
 * Initialize network code. Called from main().
 */
netinit()
{
    extern struct uba_device ubdinit[];
    register struct uba_driver *udp;
    register struct uba_device *ui = &ubdinit[0];

    if(netoff) return;
    mbinit();
    for(ui = &ubdinit[0]; udp = ui->ui_driver; ui++) {
        /* ud_probe could go here */
        ui->ui_alive = 1;
        udp->ud_dinfo[ui->ui_unit] = ui;
        (*udp->ud_attach)(ui);
    }
}

#ifdef INET
loattach(); /* XXX */
ifinit();
pfinit(); /* must follow interfaces */
#endif

#ifdef TCPACKMOST
/*
 * Fool some systems into thinking that tcp_maxseg == sbpace ==
 * max window. Ergo, force acking of most packets when talking to
 * (the vax)
 */
#endif

/*
 */
{
    extern int tcp_sendspace, tcp_recvspace;

    tcp_sendspace = tcp_recvspace = 1024;
}

#endif

/* netreset -- network reset system call */
/* 3/23/83 billn: doesnt quite work yet */
netreset()
{
    register wait = 100000;
    register s = spl6();
    register struct file *fp;
    extern int netisr;

    netoff = 1; /* prevent net traffic (during resets) */
    netisr = 0;

    /* unbeknownst sockets to UNIX */
    for (fp = &file[0]; fp < (struct file *)v.ve_file; fp++) {
        if ((fp->f_flag & FSOCKET) && fp->f_count) {
            bzero ((char *)fp, sizeof (struct file));
        }
    }

    mbinit(); /* relink mbufs */
#ifdef INET
    ifinit(); /* re-init interfaces */
    pfinit(); /* must follow interfaces */
#endif
    splx(s);
    while (wait--); /* wait for pending ints. */
    netoff = 0; /* allow traffic */
}

splimp()
{
    return spl6();
}

splnet()
{
    return spl6();
}

/*
 * Entered via software interrupt vector at spl1. Check netisr bit array
 * for tasks requesting service.
 */
netintr()
{
    register int onetisr;

    while(spl7(), (onetisr = netisr)) {
        netisr = 0;
        (void) splnet();
        if (onetisr & (1<<NETISR_RAW))
            rawintr();
        if (onetisr & (1<<NETISR_IP))
            ipintr();
        if (protofast <= 0) {
            protofast = HZ / PR_FASTHZ;
            pffasttimo();
        }
        if (protoslow <= 0) {
            protoslow = HZ / PR_SLOWHZ;
            pfslowtimo();
        }
        if (ifnetslow <= 0) {
            ifnetslow = HZ / IFNET_SLOWHZ;
            if_slowtimo();
        }
    }
}

```

```

}

int    enprint = 0;        /* enable nprintf */

/*
 * net printf.  prints to net log area in memory (nlbase, nlsiz).
 */
/* VARARGS1 */
nprintf (fmt, xl)
char *fmt;
unsigned xl;
{
    if (enprint == 0) return;
    /* billn -- do regular printf for now
    prf (fmt, &xl, 4);
    */
    printf (fmt, xl);
}

/*
 * Select system call.
 */
select()
{
    register struct uap {
        int    nfd;
        fd_set *rp, *wp;
        long   timo;
    } *ap = (struct uap *)u.u_ap;
    fd_set rd, wr;
    int nfds = 0;
    long selscan();
    extern setrun();
    long readable = 0, writeable = 0;
    time_t t = time;
    int s, tsel, ncoll, rem;
    label_t lqsav;

    if (ap->nfd > NOFILE)
        ap->nfd = NOFILE;
    if (ap->nfd < 0) {
        u.u_error = EBADF;
        return;
    }
    if (ap->rp && copyin((caddr_t)ap->rp, (caddr_t)&rd, sizeof(fd_set)))
        return;
    if (ap->wp && copyin((caddr_t)ap->wp, (caddr_t)&wr, sizeof(fd_set)))
        return;
retry:
    ncoll = nselcoll;
    u.u_procp->p_flag |= SSEL;
    if (ap->rp)
        readable = selscan(ap->nfd, rd, &nfds, FREAD);
    if (ap->wp)
        writeable = selscan(ap->nfd, wr, &nfds, FWRITE);
    if (u.u_error)
        goto done;
    if (readable || writeable)
        goto done;
    rem = (ap->timo+999)/1000 - (time - t);
    if (ap->timo == 0 || rem <= 0)
        goto done;
    s = spl6();
    if ((u.u_procp->p_flag & SSEL) == 0 || nselcoll != ncoll) {
        u.u_procp->p_flag &= ~SSEL;
        splx(s);
        goto retry;
    }
    u.u_procp->p_flag &= ~SSEL;
    if (rem) {
        bcopy((caddr_t)u.u_qsav, (caddr_t)lqsav, sizeof (label_t));
        if (save(u.u_qsav)) {
            rm_callout(setrun, (caddr_t)u.u_procp);
            u.u_error = EINTR;

```

```

        splx(s);
        goto done;
    }
    rem = rem*v.v_hz;
    timeout(setrun, (caddr_t)u.u_procp, rem);
}
sleep((caddr_t)&selwait, PZERO+1);
if (rem) {
    bcopy((caddr_t)lqsav, (caddr_t)u.u_qsav, sizeof (label_t));
    rm_callout(setrun, (caddr_t)u.u_procp);
}
splx(s);
goto retry;
done:
rd.fds_bits[0] = readable;
wr.fds_bits[0] = writeable;
u.u_rvall = nfds;
if (ap->rp)
    (void) copyout((caddr_t)&rd, (caddr_t)ap->rp, sizeof(fd_set));
if (ap->wp)
    (void) copyout((caddr_t)&wr, (caddr_t)ap->wp, sizeof(fd_set));
}

/*
 * remove entry in callout vector
 * which is scanned by clock interrupt
 */
rm_callout(func, arg)
int (*func)();
caddr_t arg;
{
    register struct callo *p1, *p2;
    register int tt;
    int pri;

    p1 = &callout[0];
    pri = spl7();
    while(p1->c_func != 0) {
        if ((p1->c_func == func) && (p1->c_arg == arg))
            break;
        p1++;
    }
    if (p1 >= (struct callo *)v.ve_call-1) {
        printf("Timeout entry not found, not deleted\n");
        return;
    }
    /* copy everything that follows in the list up one
    position, adding our unused time to theirs */
    tt = p1->c_time;
    p2 = p1;
    while(p1->c_func != 0) {
        p2++;
        p1->c_time = p2->c_time + tt;
        p1->c_func = p2->c_func;
        p1->c_arg = p2->c_arg;
        p1 = p2;
    }
    splx(pri);
}

long
selscan(nfd, fds, nfdp, flag)
int nfd;
fd_set fds;
int *nfdp, flag;
{
    struct file *fp;
    struct inode *ip;
    long bits, res = 0;
    int i, able;

    bits = fds.fds_bits[0];
    while (i = ffs(bits)) {
        if (i >= nfd)
            break;
        bits &= ~(1L<<(i-1));

```

```

fp = u.u_ofile[i-1];
if (fp == NULL) {
    u.u_error = EBADF;
    return (0);
}
if (fp->f_flag & FSOCKET)
    able = sselect((struct socket *)fp->f_socket, flag);
else {
    ip = fp->f_inode;
    switch (ip->i_mode & IFMT) {

#ifdef notdef
        case IFCHR:
            able =
                (*cdevsw[major(ip->i_rdev)].d_select)
                ((int)ip->i_rdev, flag);
#else
            /*
             * for now the only char device we need
             * to select on is the control side of
             * ptys -- for the rlogin daemon. At
             * some point someone can put general
             * select code into all char. devices.
             */
            {
                extern int ptc_dev;

                if (major((int)(ip->i_rdev)) == ptc_dev)
                    able = ptcselect((int)ip->i_rdev, flag);
                else
                    able = 0;
            }
#endif

        break;

        case IFIFO:
        case IFBLK:
        case IFREG:
        case IFDIR:
            able = 1;
            break;
    }
}
if (able) {
    res |= (1L<<(i-1));
    (*nfdp)++;
}
return (res);
}

ffs(mask)
long mask;
{
    register int i;
    register imask;

    if (mask == 0) return (0);
    imask = loint(mask);
    for(i=1; i<=16; i++) {
        if (imask & 1)
            return (i);
        imask >>= 1;
    }
    imask = hiint(mask);
    for(; i<=32; i++) {
        if (imask & 1)
            return (i);
        imask >>= 1;
    }
    return (0); /* can't get here anyway! */
}

#ifdef notdef
/*ARGSUSED*/
seltrue(dev, flag)

```

```

    dev_t dev;
    int flag;
{
    return (1);
}
#endif

selwakeup(p, coll)
register struct proc *p;
int coll;
{
    int s;

    if (coll) {
        nselcoll++;
        wakeup((caddr_t)&selwait);
    }
    s = spl6();
    if (p) {
        if (p->p_wchan == (caddr_t)&selwait)
            setrun(p);
        else {
            if (p->p_flag & SSEL)
                p->p_flag &= ~SSEL;
        }
    }
    splx(s);
}

#ifdef notdef
/* ARGSUSED */
nulselect(x, y)
{
    return 0;
}
#endif

char hostname[32] = "hostnameunknown";
int hostnamelen = 16;

gethostname()
{
    register struct a {
        char *hostname;
        int len;
    } *uap = (struct a *)u.u_ap;
    register int len;

    len = uap->len;
    if (len > hostnamelen)
        len = hostnamelen;
    if (copyout((caddr_t)hostname, (caddr_t)uap->hostname, len))
        u.u_error = EFAULT;
}

sethostname()
{
    register struct a {
        char *hostname;
        int len;
    } *uap = (struct a *)u.u_ap;

    if (!user())
        return;
    if (uap->len > sizeof(hostname) - 1) {
        u.u_error = EINVAL;
        return;
    }
    hostnamelen = uap->len;
    if (copyin((caddr_t)uap->hostname, hostname, uap->len + 1))
        u.u_error = EFAULT;
}
/*

```

```

* Some misc. subroutines.  Prob should be in a sep module
*/

#ifdef notdef      /* system 3 has it's own */
/*
* Provide about n microseconds of delay
*/
delay(n)
long n;
{
    register hi,low;

    low = (n&0177777);
    hi = n>>16;
    if(hi==0) hi=1;
    do {
        do { } while(--low);
    } while(--hi);
}
#endif

/*
* compare bytes; same result as VAX cmpc3.
*/
bcmp(s1, s2, n)
register char *s1, *s2;
register n;
{
    do
        if(*s1++ != *s2++) break;
    while(--n);
    return(n);
}

/*
* Insert an entry onto queue.
*/
_insq(e,prev)
register struct vaxque *e,*prev;
{
    register x = spl7();
    e->vq_prev = prev;
    e->vq_next = prev->vq_next;
    if (prev->vq_next)
        prev->vq_next->vq_prev = e;
    prev->vq_next = e;
    splx(x);
}

/*
* Remove an entry from queue.
*/
_remque(e)
register struct vaxque *e;
{
    register x = spl7();

    e->vq_prev->vq_next = e->vq_next;
    if (e->vq_next)
        e->vq_next->vq_prev = e->vq_prev;
    splx(x);
}

struct proc *
pfind(pid)
int pid;
{
    register struct proc *p;

    for (p=proc; p < &proc[v.v_proc]; p++)
        if (p->p_pid == pid)
            return (p);
    return ((struct proc *)0);
}

```

```

/* bzero(p,n) -- zero n bytes starting at p */

bzero(p,n)
register char * p;
register n;
{
    if (n)
        do {
            *p++ = 0;
        } while (--n);
}

/*
* iomalloc -- allocate clks of mem. for io.
* Right now, no dma.
*/
mbioalloc()
{
    return 0;
}

/*
* mballloc should be combined with/done like memap(). THIS STUFF DEPENDS
* ON MSIZE BEING A POWER OF 2.
*/
#define MBUFCONFIG 1      /* undef extern of Mbuf, mbuf */
#include <net/mbuf.h>

char mbufbufs[(NMBUFS+1)*MSIZE];

struct mbuf *
mballoc()
{
    unsigned int location = (unsigned int)&mbufbufs[0];
    int slop = location & (MSIZE-1);

    /* round actual buffers to MSIZE boundry */
    return ((struct mbuf*)(location + MSIZE - slop));
}

```



```

/*      tcp_debug.c      4.3      82/03/29      */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/system.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/socket.h"
#include "net/socketvar.h"
#define PRUREQUESTS
#include "net/protosw.h"
#include "net/in.h"
#include "net/route.h"
#include "net/in_pcb.h"
#include "net/in_system.h"
#include "net/if.h"
#include "net/ip.h"
#include "net/ip_var.h"
#include "net/tcp.h"
#define TCPSTATES
#include "net/tcp_fsm.h"
#include "net/tcp_seq.h"
#define TCPTIMERS
#include "net/tcp_timer.h"
#include "net/tcp_var.h"
#include "net/tcpip.h"
#define TANAMES
#include "net/tcp_debug.h"
#include "errno.h"

int      tcpconsdebug = 1;
/*
 * Tcp debug routines
 */
tcp_trace(act, ostate, tp, ti, req)
short act, ostate;
struct tcpcb *tp;
struct tcphdr *ti;
int req;
{
    tcp_seq seq, ack;
    int len, flags;
    struct tcp_debug *td = &tcp_debug[tcp_debx++];

    if (tcp_debx == TCP_NDEBUG)
        tcp_debx = 0;
    td->td_time = iptime();
    td->td_act = act;
    td->td_ostate = ostate;
    td->td_tcb = (caddr_t)tp;
    if (tp)
        td->td_cb = *tp;
    else
        bzero((caddr_t)&td->td_cb, sizeof (*tp));
    if (ti)
        td->td_ti = *ti;
    else
        bzero((caddr_t)&td->td_ti, sizeof (*ti));
    td->td_req = req;
    if (tcpconsdebug == 0)
        return;
    if (tp)
        printf("%x %s:", tp, tcpstates[ostate]);
    else
        printf("???????? ");
    printf("%s ", tanames[act]);
    switch (act) {
        case TA_INPUT:
        case TA_OUTPUT:
            seq = ti->ti_seq;
            ack = ti->ti_ack;
            len = ti->ti_len;
            if (act == TA_OUTPUT) {
                if (act == TA_OUTPUT) {
                    seq = ntohl(seq);
                    ack = ntohl(ack);
                    len = ntohs((u_short)len);
                }
            }
            #endif
            if (act == TA_OUTPUT)
                len -= sizeof (struct tcphdr);
            if (len)
                printf("[%x..%x]", seq, seq+len);
            else
                printf("%x", seq);
            printf("#%x", ack);
            flags = ti->ti_flags;
            if (flags) {
                #ifndef lint
                char *cp = "<";
                #define pf(f) { if (ti->ti_flags&TH/**/f) { printf("%s%s", cp, "f"); cp = ", "; } }
                pf(SYN); pf(ACK); pf(FIN); pf(RST);
                #endif
                printf(">");
            }
            break;
        case TA_USER:
            printf("%s", prurequests[req&0xfff]);
            if ((req & 0xfff) == PRU_SLOWTIMO)
                printf("< %s>", tcptimers[req>>8]);
            break;
    }
    if (tp)
        printf(" -> %s", tcpstates[tp->t_state]);
    /* print out internal state of tp !?! */
    printf("\n");
    if (tp == 0 || tcpconsdebug == 2)
        return;
    printf("\trcv_{nxt,wnd} (%x,%x) snd_{una,nxt,max} (%x,%x,%x)\n",
           tp->rcv_nxt, tp->rcv_wnd, tp->snd_una, tp->snd_nxt, tp->snd_max);
    printf("\tsnd_{w11,w12,wnd} (%x,%x,%x)\n",
           tp->snd_w11, tp->snd_w12, tp->snd_wnd);
}

```

```

/*      %M%      %I%      %E%      */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/system.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "net/in.h"
#include "net/route.h"
#include "net/in_pcb.h"
#include "net/in_system.h"
#include "net/if.h"
#include "net/ip.h"
#include "net/ip_var.h"
#include "net/tcp.h"
#include "net/tcp_fsm.h"
#include "net/tcp_seq.h"
#include "net/tcp_timer.h"
#include "net/tcp_var.h"
#include "net/tcpip.h"
#include "net/tcp_debug.h"
#include "errno.h"

int    tcpprintfs = 1; /* 1=print, 2= panic */
short  tcpcksum = 1;
struct sockadr_in tcp_in = { AF_INET };
struct tcphdr tcp_saveti;
extern tcponelack;

struct tcpcb *tcp_newtcpb();
/*
 * TCP input routine, follows pages 65-76 of the
 * protocol specification dated September, 1981 very closely.
 */
tcp_input(m0)
    register struct mbuf *m0;
{
    register struct tcphdr *ti;
    register struct inpcb *inp;
    register struct mbuf *m;
    struct mbuf *om = 0;
    int len, tlen, off;
    register struct tcpcb *tp = 0;
    register int tiflags;
    register struct socket *so;
    register int todrop, acked;
    short ostate;
    struct in_addr laddr;

    /*
     * Get IP and TCP header together in first mbuf.
     * Note: IP leaves IP header in first mbuf.
     */
#ifdef SIGH
    { extern int intcpinput; }
    if (intcpinput) printf("intcpinput..."); else intcpinput++;
#endif
    m = m0;
    ti = mtod(m, struct tcphdr *);
    if (((struct ip *)ti)->ip_hl > (sizeof (struct ip) >> 2))
        ip_stripoptions((struct ip *)ti, (struct mbuf *)0);
    if (m->m_off > MMAXOFF || m->m_len < sizeof (struct tcphdr) ) {
        if ((m = m_pullup(m, sizeof (struct tcphdr))) == 0) {
            tcpstat.tcps_hdrops++;
        }
    }
#ifdef SIGH
    intcpinput--;
#endif
    return;
}
    ti = mtod(m, struct tcphdr *);

/*
 * Checksum extended TCP header and data.
 */
    tlen = ((struct ip *)ti)->ip_len;
    len = sizeof (struct ip) + tlen;
    /*
     * if ((tlen > 576) || (len > 700))
     *     printf("len=%d, tlen=%d !!!\n", len, tlen);
     */
    if (tcpcksum) {
        register sum;
        ti->ti_next = ti->ti_prev = 0;
        ti->ti_xl = 0;
        ti->ti_len = (u_short)tlen;
#ifdef WATCHOUT
        ti->ti_len = htons((u_short)ti->ti_len);
#endif
        mprint(m, "tin before ck");
        nprintf("len%o\n", (unsigned)len);
        /*
         * if (len == 40)
         *     printf("segment:\n");for(j=0;j<4;j++){for(i=0;i<10;i++)printf("%x ",((char *)ti)[i+10*
         *     if (ti->ti_sum = in_cksum(m, len)) {
         *         /*
         *         if (sum = in_cksum(m, len)) {
         *             nprintf("badsum%o\n", ti->ti_sum);
         *             tcpstat.tcps_badsum++;
         *             if (tcpprintfs)
         *                 printf("tcp cksum %x, length %d, loca %x\n",ti->ti_sum,len,ti);
         *             /*
         *             printf("tcp cksum %x, length %d, loca %x\n",sum,len,ti);
         *             if (len == 40)
         *                 printf("segment:\n");for(j=0;j<4;j++){for(i=0;i<10;i++)printf("%x ",((char *)t
         *                 if (tcpprintfs==2) panic("tcp ck\n");
         *                 goto drop;
         *             }
         *             ti->ti_sum = sum;
         *         }
         *     }
         *
         * Check that TCP offset makes sense,
         * pull out TCP options and adjust length.
         */
        off = ti->ti_off << 2;
        if (off < sizeof (struct tcphdr) || off > tlen) {
            tcpstat.tcps_badoff++;
            goto drop;
        }
        tlen -= off;
        ti->ti_len = tlen;
        if (off > sizeof (struct tcphdr)) {
            if ((m = m_pullup(m, sizeof (struct ip) + off)) == 0) {
                tcpstat.tcps_hdrops++;
            }
        }
#ifdef SIGH
        intcpinput--;
#endif
    }
    return;
}
    ti = mtod(m, struct tcphdr *);
}
/*

```

```

    * Drop TCP and IP headers.
    */
    m->m_off += sizeof(struct tcphdr);
    m->m_len -= sizeof(struct tcphdr);

#ifdef WATCHOUT
    /*
     * Convert TCP protocol specific fields to host format.
     */
    ti->ti_seq = ntohl(ti->ti_seq);
    ti->ti_ack = ntohl(ti->ti_ack);
    ti->ti_win = ntohs(ti->ti_win);
    ti->ti_urp = ntohs(ti->ti_urp);
#endif

    /*
     * Locate pcb for segment. On match, update the local
     * address stored in the block to reflect anchoring.
     */
    inp = in_pcblookup
        (&tc, ti->ti_src, ti->ti_sport, ti->ti_dst, ti->ti_dport,
         INPLOOKUP_WILDCARD);

    /*
     * If the state is CLOSED (i.e., TCB does not exist) then
     * all data in the incoming segment is discarded.
     */
    if (inp == 0)
        goto dropwithreset;
    tp = intotcp(inp);
    if (tp == 0)
        goto dropwithreset;
    so = inp->inp_socket;
    if (so->so_options & SO_DEBUG) {
        ostate = tp->t_state;
        tcp_saveti = *ti;
    }

    /*
     * Segment received on connection.
     * Reset idle time and keep-alive timer.
     */
    tp->t_idle = 0;
    tp->t_timer[TCPT_KEEP] = TCPTV_KEEP;

    /*
     * Process options.
     */
    if (om) {
        tcp_dooptions(tp, om);
        om = 0;
    }

    /*
     * Calculate amount of space in receive window,
     * and then do TCP input processing.
     */
    tp->rcv_wnd = sbspace(&so->so_rcv);
    if (tp->rcv_wnd < 0)
        tp->rcv_wnd = 0;

    switch (tp->t_state) {

    /*
     * If the state is LISTEN then ignore segment if it contains an RST.
     * If the segment contains an ACK then it is bad and send a RST.
     * If it does not contain a SYN then it is not interesting; drop it.
     * Otherwise initialize tp->rcv_nxt, and tp->irs, select an initial
     * tp->iss, and send a segment:
     * <SEQ=ISS><ACK=RCV_NXT><CTL=SYN,ACK>
     * Also initialize tp->snd_nxt to tp->iss+1 and tp->snd_una to tp->iss.
     * Fill in remote peer address fields if not previously specified.
     * Enter SYN_RECEIVED state, and process any other fields of this
     * segment in this state.
     */
    case TCPS_LISTEN:
        if (tiflags & TH_RST)
            goto drop;
        if (tiflags & TH_ACK)
            goto dropwithreset;
        if ((tiflags & TH_SYN) == 0)
            goto drop;
        tcp_in.sin_addr = ti->ti_src;
        tcp_in.sin_port = ti->ti_sport;
        laddr = inp->inp_laddr;
        if (inp->inp_laddr.s_addr == 0)
            inp->inp_laddr = ti->ti_dst;
        if (in_pcbconnect(inp, (struct sockaddr_in *)&tcp_in)) {
            inp->inp_laddr = laddr;
            goto drop;
        }
        tp->t_template = tcp_template(tp);
        if (tp->t_template == 0) {
            in_pcbdisconnect(inp);
            inp->inp_laddr = laddr;
            tp = 0;
            goto drop;
        }
        tp->iss = tcp_iss; tcp_iss += TCP_ISSINCR/2;
        tp->irs = ti->ti_seq;
        tcp_sendseqinit(tp);
        tcp_rcvseqinit(tp);
        tp->t_state = TCPS_SYN_RECEIVED;
        tp->t_timer[TCPT_KEEP] = TCPTV_KEEP;
        goto trimthenstep6;

    /*
     * If the state is SYN_SENT:
     * if seq contains an ACK, but not for our SYN, drop the input.
     * if seq contains a RST, then drop the connection.
     * if seq does not contain SYN, then drop it.
     * Otherwise this is an acceptable SYN segment
     * initialize tp->rcv_nxt and tp->irs
     * if seq contains ack then advance tp->snd_una
     * if SYN has been acked change to ESTABLISHED else SYN_RCVD state
     * arrange for segment to be acked (eventually)
     * continue processing rest of data/controls, beginning with URG
     */
    case TCPS_SYN_SENT:
        if ((tiflags & TH_ACK) &&
            /* this should be SEQ_LT; is SEQ_LEQ for BBN vax TCP only */
            (SEQ_LT(ti->ti_ack, tp->iss) ||
             SEQ_GT(ti->ti_ack, tp->snd_max)))
            goto dropwithreset;
        if (tiflags & TH_RST) {
            if (tiflags & TH_ACK) {
                tcp_drop(tp, ECONNREFUSED);
                tp = 0;
            }
            goto drop;
        }
        if ((tiflags & TH_SYN) == 0)
            goto drop;
        tp->snd_una = ti->ti_ack;
        if (SEQ_LT(tp->snd_nxt, tp->snd_una))
            tp->snd_nxt = tp->snd_una;
        tp->t_timer[TCPT_REXMT] = 0;
        tp->irs = ti->ti_seq;
        tcp_rcvseqinit(tp);
        tp->t_flags |= TF_ACKNOW;
        if (SEQ_GT(tp->snd_una, tp->iss)) {
            if (so->so_options & SO_ACCEPTCONN)
                so->so_state |= SS_CONNAWAITING;
            soisconnected(so);
            tp->t_state = TCPS_ESTABLISHED;
            (void) tcp_reass(tp, (struct tcphdr *)0);
        } else
            tp->t_state = TCPS_SYN_RECEIVED;
        goto trimthenstep6;

    trimthenstep6:
    /*

```

```

    * Advance ti->ti_seq to correspond to first data byte.
    * If data, trim to stay within window,
    * dropping FIN if necessary.
    */
    ti->ti_seq++;
    if (ti->ti_len > tp->rcv_wnd) {
        todrop = ti->ti_len - tp->rcv_wnd;
        m_adj(m, -todrop);
        ti->ti_len = tp->rcv_wnd;
        ti->ti_flags &= ~TH_FIN;
    }
    tp->snd_wll = ti->ti_seq - 1;
    goto step6;
}

/*
 * States other than LISTEN or SYN_SENT.
 * First check that at least some bytes of segment are within
 * receive window.
 */
if (tp->rcv_wnd == 0) {
    /*
     * If window is closed can only take segments at
     * window edge, and have to drop data and PUSH from
     * incoming segments.
     */
    if (tp->rcv_nxt != ti->ti_seq)
        goto dropafterack;
    if (ti->ti_len > 0) {
        m_adj(m, ti->ti_len);
        ti->ti_len = 0;
        ti->ti_flags &= ~(TH_PUSH|TH_FIN);
    }
} else {
    /*
     * If segment begins before rcv_nxt, drop leading
     * data (and SYN); if nothing left, just ack.
     */
    todrop = tp->rcv_nxt - ti->ti_seq;
    if (todrop > 0) {
        if (ti->ti_flags & TH_SYN) {
            ti->ti_flags &= ~TH_SYN;
            ti->ti_seq++;
            if (ti->ti_urp > 1)
                ti->ti_urp--;
            else
                ti->ti_urp--;
            todrop--;
        }
        if (todrop > ti->ti_len ||
            todrop == ti->ti_len && (ti->ti_flags & TH_FIN) == 0)
            goto dropafterack;
        m_adj(m, todrop);
        ti->ti_seq += todrop;
        ti->ti_len -= todrop;
        if (ti->ti_urp > todrop)
            ti->ti_urp -= todrop;
        else {
            ti->ti_flags &= ~TH_URG;
            ti->ti_flags &= ~TH_URG;
            ti->ti_urp = 0;
        }
    }
}
/*
 * If segment ends after window, drop trailing data
 * (and PUSH and FIN); if nothing left, just ACK.
 */
todrop = (ti->ti_seq+ti->ti_len) - (tp->rcv_nxt+tp->rcv_wnd);
if (todrop > 0) {
    if (todrop >= ti->ti_len)
        goto dropafterack;
    m_adj(m, -todrop);
    ti->ti_len -= todrop;
    ti->ti_flags &= ~(TH_PUSH|TH_FIN);
}

```

```

}

/*
 * If a segment is received on a connection after the
 * user processes are gone, then RST the other end.
 */
if ((so->so_state & SS_USERGONE) && tp->t_state > TCPS_CLOSE_WAIT &&
    ti->ti_len) {
    tcp_close(tp);
    tp = 0;
    goto dropwithreset;
}

/*
 * If the RST bit is set examine the state:
 * SYN_RECEIVED STATE:
 *   If passive open, return to LISTEN state.
 *   If active open, inform user that connection was refused.
 * ESTABLISHED, FIN_WAIT_1, FIN_WAIT_2, CLOSE_WAIT STATES:
 *   Inform user that connection was reset, and close tcb.
 * CLOSING, LAST_ACK, TIME_WAIT STATES
 *   Close the tcb.
 */
if (ti->ti_flags & TH_RST) switch (tp->t_state) {

case TCPS_SYN_RECEIVED:
    if (inp->inp_socket->so_options & SO_ACCEPTCONN) {
        /* a miniature tcp_close, but invisible to user */
        if (tp->t_template) MSFREE(tp->t_template);
        MSFREE(tp);
        inp->inp_ppcb = 0;
        tp = tcp_newtcpcb(inp);
        if (tp == 0) { /* unlikely but just in case */
            tcp_drop(tp, ENOBUFS);
            tp = 0;
            goto drop;
        }
        tp->t_state = TCPS_LISTEN;
        inp->inp_faddr.s_addr = 0;
        inp->inp_fport = 0;
        inp->inp_laddr.s_addr = 0; /* not quite right */
        tp = 0;
        goto drop;
    }
    tcp_drop(tp, ECONNREFUSED);
    tp = 0;
    goto drop;

case TCPS_ESTABLISHED:
case TCPS_FIN_WAIT_1:
case TCPS_FIN_WAIT_2:
case TCPS_CLOSE_WAIT:
    tcp_drop(tp, ECONNRESET);
    tp = 0;
    goto drop;

case TCPS_CLOSING:
case TCPS_LAST_ACK:
case TCPS_TIME_WAIT:
    tcp_close(tp);
    tp = 0;
    goto drop;
}

/*
 * If a SYN is in the window, then this is an
 * error and we send an RST and drop the connection.
 */
if (ti->ti_flags & TH_SYN) {
    tcp_drop(tp, ECONNRESET);
    tp = 0;
    goto dropwithreset;
}

/*
 * If the ACK bit is off we drop the segment and return.
 */

```

```

/*
if ((tiflags & TH_ACK) == 0)
    goto drop;

/*
 * Ack processing.
 */
switch (tp->t_state) {

/*
 * In SYN_RECEIVED state if the ack ACKs our SYN then enter
 * ESTABLISHED state and continue processing, otherwise
 * send an RST.
 */
case TCPS_SYN_RECEIVED:
    if (SEQ_GT(tp->snd_una, ti->ti_ack) ||
        SEQ_GT(ti->ti_ack, tp->snd_max))
        goto dropwithreset;
    tp->snd_una++; /* SYN acked */
    if (SEQ_LT(tp->snd_nxt, tp->snd_una))
        tp->snd_nxt = tp->snd_una;
    tp->t_timer[TCPT_REXMT] = 0;
    if (so->so_options & SO_ACCEPTCONN)
        so->so_state |= SS_CONNAWAITING;
    soisconnected(so);
    tp->t_state = TCPS_ESTABLISHED;
    (void) tcp_reass(tp, (struct tcphdr *)0);
    tp->snd_wll = ti->ti_seq - 1;
    /* fall into ... */

/*
 * In ESTABLISHED state: drop duplicate ACKs; ACK out of range
 * ACKs. If the ack is in the range
 * tp->snd_una < ti->ti_ack <= tp->snd_max
 * then advance tp->snd_una to ti->ti_ack and drop
 * data from the retransmission queue. If this ACK reflects
 * more up to date window information we update our window information.
 */
case TCPS_ESTABLISHED:
case TCPS_FIN_WAIT_1:
case TCPS_FIN_WAIT_2:
case TCPS_CLOSE_WAIT:
case TCPS_CLOSING:
case TCPS_LAST_ACK:
case TCPS_TIME_WAIT:
#define ourfinisacked (acked > 0)

    if (SEQ_LEQ(ti->ti_ack, tp->snd_una))
        break;
    if (SEQ_GT(ti->ti_ack, tp->snd_max))
        goto dropafterack;
    acked = ti->ti_ack - tp->snd_una;

/*
 * If transmit timer is running and timed sequence
 * number was acked, update smoothed round trip time.
 */
    if (tp->t_rtt && SEQ_GT(ti->ti_ack, tp->t_rtseq)) {
        if (tp->t_srtt == 0)
            tp->t_srtt = tp->t_rtt * 10;
        else
            tp->t_srtt =
                (tcp_alpha * tp->t_srtt) / 10 +
                (10 - tcp_alpha) * tp->t_rtt;
    }
    /* printf("rtt %d srtt(*10) now %d\n", tp->t_rtt, tp->t_srtt); */
    tp->t_rtt = 0;
}

if (ti->ti_ack == tp->snd_max)
    tp->t_timer[TCPT_REXMT] = 0;
else {
    TCPT_RANGESET(tp->t_timer[TCPT_REXMT],
        (tcp_beta * tp->t_srtt) / 100, TCPTV_MIN, TCPTV_MAX);
    tp->t_rtt = 1;
    tp->t_rxtshift = 0;
}
}

```

```

if (acked > so->so_snd.sb_cc) {
    sbdrop(&so->so_snd, so->so_snd.sb_cc);
    tp->snd_wnd -= so->so_snd.sb_cc;
} else {
    sbdrop(&so->so_snd, acked);
    tp->snd_wnd -= acked;
    acked = 0;
}

if ((so->so_snd.sb_flags & SB_WAIT) || so->so_snd.sb_sel)
    sowakeup(so);
tp->snd_una = ti->ti_ack;
if (SEQ_LT(tp->snd_nxt, tp->snd_una))
    tp->snd_nxt = tp->snd_una;

switch (tp->t_state) {

/*
 * In FIN_WAIT_1 STATE in addition to the processing
 * for the ESTABLISHED state if our FIN is now acknowledged
 * then enter FIN_WAIT_2.
 */
case TCPS_FIN_WAIT_1:
    if (ourfinisacked) {
        /*
         * If we can't receive any more
         * data, then closing user can proceed.
         */
        if (so->so_state & SS_CANTRCVMORE)
            soisdisconnected(so);
        tp->t_state = TCPS_FIN_WAIT_2;
    }
    break;

/*
 * In CLOSING STATE in addition to the processing for
 * the ESTABLISHED state if the ACK acknowledges our FIN
 * then enter the TIME-WAIT state, otherwise ignore
 * the segment.
 */
case TCPS_CLOSING:
    if (ourfinisacked) {
        tp->t_state = TCPS_TIME_WAIT;
        tcp_canceltimers(tp);
        tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
        soisdisconnected(so);
    }
    break;

/*
 * The only thing that can arrive in LAST_ACK state
 * is an acknowledgment of our FIN. If our FIN is now
 * acknowledged, delete the TCB, enter the closed state
 * and return.
 */
case TCPS_LAST_ACK:
    if (ourfinisacked) {
        tcp_close(tp);
        tp = 0;
    }
    goto drop;

/*
 * In TIME_WAIT state the only thing that should arrive
 * is a retransmission of the remote FIN. Acknowledge
 * it and restart the finack timer.
 */
case TCPS_TIME_WAIT:
    tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
    goto dropafterack;
}

#undef ourfinisacked
}

step6:
/*
 * Update window information.

```

```

/*
if (SEQ_LT(tp->snd_wll, ti->ti_seq) || tp->snd_wll == ti->ti_seq &&
    (SEQ_LT(tp->snd_wl2, ti->ti_ack) ||
    tp->snd_wl2 == ti->ti_ack && ti->ti_win > tp->snd_wnd)) {
    tp->snd_wnd = ti->ti_win;
    tp->snd_wll = ti->ti_seq;
    tp->snd_wl2 = ti->ti_ack;
    /*
    if (tp->snd_wnd > 0)
    /*
    if (tp->snd_wnd != 0)
        tp->t_timer[TCPT_PERSIST] = 0;
}

/*
 * Process segments with URG.
 */
if ((tiflags & TH_URG) && ti->ti_urg &&
    TCPS_HAVERCVDFIN(tp->t_state) == 0) {
    /*
    * If this segment advances the known urgent pointer,
    * then mark the data stream. This should not happen
    * in CLOSE_WAIT, CLOSING, LAST_ACK or TIME_WAIT STATES since
    * a FIN has been received from the remote side.
    * In these states we ignore the URG.
    */
    if (SEQ_GT(ti->ti_seq+ti->ti_urg, tp->rcv_up)) {
        tp->rcv_up = ti->ti_seq + ti->ti_urg;
        so->so_oobmark = so->so_rcv.sb_cc +
            (tp->rcv_up - tp->rcv_next) - 1;
        if (so->so_oobmark == 0)
            so->so_state |= SS_RCVATMARK;
}

#ifdef TCPTRUEOOB
    if ((tp->t_flags & TF_DOOOB) == 0)
        sohasoutofband(so);
    tp->t_oobflags &= ~TCPOOB_HAVEDATA;
}
/*
 * Remove out of band data so doesn't get presented to user.
 * This can happen independent of advancing the URG pointer,
 * but if two URG's are pending at once, some out-of-band
 * data may creep in... ick.
 */
if (ti->ti_urg <= ti->ti_len) {
    tcp_pulloutofband(so, ti);
}

}

/*
 * Process the segment text, merging it into the TCP sequencing queue,
 * and arranging for acknowledgment of receipt if necessary.
 * This process logically involves adjusting tp->rcv_wnd as data
 * is presented to the user (this happens in tcp_usrreq.c,
 * case PRU_RCVD). If a FIN has already been received on this
 * connection then we just ignore the text.
 */
if ((ti->ti_len || (tiflags&TH_FIN)) &&
    TCPS_HAVERCVDFIN(tp->t_state) == 0) {
    tiflags = tcp_reass(tp, ti);
    if (tcpnodelack == 0)
        tp->t_flags |= TF_DEBLACK;
    else
        tp->t_flags |= TF_ACKNOW;
} else {
    m_freem(m);
    tiflags &= ~TH_FIN;
}

/*
 * If FIN is received ACK the FIN and let the user know
 * that the connection is closing.
 */
if (tiflags & TH_FIN) {
    if (TCPS_HAVERCVDFIN(tp->t_state) == 0) {
        socantrcvmore(so);
        tp->t_flags |= TF_ACKNOW;
        tp->rcv_next++;
    }
    switch (tp->t_state) {
        /*
        * In SYN_RECEIVED and ESTABLISHED STATES
        * enter the CLOSE_WAIT state.
        */
        case TCPS_SYN_RECEIVED:
        case TCPS_ESTABLISHED:
            tp->t_state = TCPS_CLOSE_WAIT;
            break;

        /*
        * If still in FIN_WAIT_1 STATE FIN has not been acked so
        * enter the CLOSING state.
        */
        case TCPS_FIN_WAIT_1:
            tp->t_state = TCPS_CLOSING;
            break;

        /*
        * In FIN_WAIT_2 state enter the TIME_WAIT state,
        * starting the time-wait timer, turning off the other
        * standard timers.
        */
        case TCPS_FIN_WAIT_2:
            tp->t_state = TCPS_TIME_WAIT;
            tcp_canceltimers(tp);
            tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
            soisdisconnected(so);
            break;

        /*
        * In TIME_WAIT state restart the 2 MSL time_wait timer.
        */
        case TCPS_TIME_WAIT:
            tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
            break;
    }
}

if (so->so_options & SO_DEBUG)
    tcp_trace(TA_INPUT, ostate, tp, &tcp_saveti, 0);

/*
 * Return any desired output.
 */
#ifdef SIGH
(void) tcp_output(tp);
#endif
intcplinput--;
#endif
return;

dropafterack:
/*
 * Generate an ACK dropping incoming segment if it occupies
 * sequence space, where the ACK reflects our state.
 */
if ((tiflags&TH_RST) ||
    tlen == 0 && (tiflags&(TH_SYN|TH_FIN)) == 0)
    goto drop;
if (tp->t_inpcb->inp_socket->so_options & SO_DEBUG)
    tcp_trace(TA_RESPOND, ostate, tp, &tcp_saveti, 0);
tcp_respond(tp, ti, tp->rcv_next, tp->snd_next, TH_ACK);
#ifdef SIGH
intcplinput--;
#endif
return;

dropwithreset:
if (om)
    (void) m_free(om);

/*
 * Generate a RST, dropping incoming segment.
 * Make ACK acceptable to originator of segment.

```

```

    /*
    if (tiflags & TH_RST)
        goto drop;
    if (tiflags & TH_ACK)
        tcp_respond(tp, ti, (tcp_seq)0, ti->ti_ack, TH_RST);
    else {
        if (tiflags & TH_SYN)
            ti->ti_len++;
        tcp_respond(tp, ti, ti->ti_seq+ti->ti_len, (tcp_seq)0,
            TH_RST|TH_ACK);
    }
#ifdef SIGH
    intcpinput--;
#endif
    return;

drop:
    /*
    * Drop space held by incoming segment and return.
    */
    if (tp && (tp->t_inpcb->inp_socket->so_options & SO_DEBUG))
        tcp_trace(TA_DROP, ostate, tp, &tcp_saveti, 0);
    m_freem(m);
#ifdef SIGH
    intcpinput--;
#endif
    return;
}

tcp_dooptions(tp, om)
    register struct tcpcb *tp;
    register struct mbuf *om;
{
    register u_char *cp;
    register int opt, optlen, cnt;

    MAPSAVE();
    cp = mtod(om, u_char *);
    cnt = om->m_len;
    for (; cnt > 0; cnt -= optlen, cp += optlen) {
        opt = UCHAR(cp[0]);
        if (opt == TCPOPT_EOL)
            break;
        if (opt == TCPOPT_NOP)
            optlen = 1;
        else
            optlen = UCHAR(cp[1]);
        switch (opt) {

        default:
            break;

        case TCPOPT_MAXSEG:
            if (optlen != 4)
                continue;
            bcopy((caddr_t)(cp+2), (caddr_t)&tp->t_maxseg, 2);
#ifdef SMALLTCP
            tp->t_maxseg = 256;
#endif
#ifdef WATCHOUT
            tp->t_maxseg = ntohs((u_short)tp->t_maxseg);
#endif

            break;

#ifdef TCPTRUEOOB
        case TCPOPT_WILLOOB:
            tp->t_flags |= TF_DOOOB;
            break;

        case TCPOPT_OOBDATA: {
            int seq;
            register struct socket *so = tp->t_inpcb->inp_socket;
            tcp_seq mark;

```

```

            if (optlen != 8)
                continue;
            seq = UCHAR(cp[2]);
            if (seq < UCHAR(tp->t_iobseq))
                seq += 256;
            printf("oobdata cp[2] %d iobseq %d seq %d\n", cp[2], tp->t_iobseq, seq);
            if (seq - UCHAR(tp->t_iobseq) > 128) {
                printf("bad seq\n");
                tp->t_oobflags |= TCPOOB_OWEACK;
                break;
            }
            tp->t_iobseq = cp[2];
            tp->t_iobc = cp[3];
            bcopy(cp+4, &mark, sizeof mark);
#ifdef WATCHOUT
            mark = ntohl(mark);
#endif
            so->so_oobmark = so->so_rcv.sb_cc + (mark-tp->rcv_next);
            if (so->so_oobmark == 0)
                so->so_state |= SS_RCVATMARK;
            printf("take oob data %x input iobseq now %x\n", tp->t_iobc, tp->t_iobseq);
            sohasoutofband(so);
            break;
        }

        case TCPOPT_OOBACK: {
            int seq;

            if (optlen != 4)
                continue;
            if (tp->t_oobseq != cp[2]) {
                printf("wrong ack\n");
                break;
            }
            printf("take oob ack %x and cancel rexmt\n", cp[2]);
            tp->t_oobflags &= ~TCPOOB_NEEDACK;
            tp->t_timer[TCPT_OOBREXMT] = 0;
            break;
        }
    }
    (void) m_free(om);
    MAPREST();
}

/*
 * Pull out of band byte out of a segment so
 * it doesn't appear in the user's data queue.
 * It is still reflected in the segment length for
 * sequencing purposes.
 */
tcp_pulloutofband(so, ti)
    register struct socket *so;
    register struct tcpiphdr *ti;
{
    register struct mbuf *m;
    register int cnt = ti->ti_urp - 1;

    MAPSAVE();
    m = dtom(ti);
    while (cnt >= 0) {
        if (m->m_len > cnt) {
            char *cp = mtod(m, caddr_t) + cnt;
            struct tcpcb *tp = sototpcb(so);

            tp->t_iobc = *cp;
            tp->t_oobflags |= TCPOOB_HAVEDATA;
            bcopy(cp+1, cp, (int)(m->m_len - cnt - 1));
            m->m_len--;
            goto out;
        }
        cnt -- m->m_len;
        m = m->m_next;
        if (m == 0)
            break;
    }
}

```

```

    }
    panic("tcp_pulloutofband");
out:
    MAPREST();
}

#ifdef WATCHOUT /* this is for the ll */
#define ti_mbuf ti_sum
#define DTOM(d) ((struct mbuf *) ((d)->ti_mbuf))
#define INSQUE(i,p) { \
    struct tcpiphdr *tii; \
    MSGET(tii, struct tcpiphdr, 0); if (tii == 0) goto drop; \
    insque(tii,p); \
    tii->ti_len = (i)->ti_len; tii->ti_seq = (i)->ti_seq; \
    tii->ti_flags = (i)->ti_flags; tii->ti_mbuf = m0; i = tii; }
#else
#define DTOM(d) dtom(d)
#define INSQUE(i,p) insque(i,p)
#endif

/*
 * Insert segment ti into reassembly queue of tcp with
 * control block tp. Return TH_FIN if reassembly now includes
 * a segment with FIN.
 */
tcp_reass(tp, ti)
    register struct tcpcb *tp;
    register struct tcpiphdr *ti;
{
    register struct tcpiphdr *q,*qp;
    register struct socket *so = tp->inpcb->inp_socket;
    register struct mbuf *m,*m0;
    register int cnt,flags,empty = 0;

    /*
     * Call with ti==0 after become established to
     * force pre-ESTABLISHED data up to user socket.
     */
    if (ti == 0)
        goto present;
    m0 = dtom(ti);

    /*
     * Find a segment which begins after this one does.
     */
    for (q = tp->seg_next; q != (struct tcpiphdr *)tp;
         q = (struct tcpiphdr *)q->ti_next)
        if (SEQ_GT(q->ti_seq, ti->ti_seq))
            break;

    /*
     * If there is a preceding segment, it may provide some of
     * our data already. If so, drop the data from the incoming
     * segment. If it provides all of our data, drop us.
     */
    if ((struct tcpiphdr *)q->ti_prev != (struct tcpiphdr *)tp) {
        register int i;
        q = (struct tcpiphdr *)q->ti_prev;
        /* conversion to int (in i) handles seq wraparound */
        i = q->ti_seq + q->ti_len - ti->ti_seq;
        if (i > 0) {
            if (i >= ti->ti_len)
                goto drop;
            m_adj(m0, i);
            ti->ti_len -= i;
            ti->ti_seq += i;
        }
        q = (struct tcpiphdr *) (q->ti_next);
    }

    /*
     * While we overlap succeeding segments trim them or,
     * if they are completely covered, dequeue them.
     */
    while (q != (struct tcpiphdr *)tp) {
        register int i = (ti->ti_seq + ti->ti_len) - q->ti_seq;

```

```

        if (i <= 0)
            break;
        if (i < q->ti_len) {
            q->ti_seq += i;
            q->ti_len -= i;
            m_adj(DTOM(q), i);
            break;
        }
        qp = q;
        q = (struct tcpiphdr *)q->ti_next;
        m = DTOM(qp);
        remque(qp);
#ifdef WATCHOUT
        MSFREE(qp);
#endif
        m_freem(m);
    }

    /*
     * Stick new segment in its place. Insque stuff can be expensive,
     * so avoid if possible.
     */
    if (tp->seg_next == (struct tcpiphdr *)tp) /* queue was empty */
        empty++;
    else
        INSQUE(ti, q->ti_prev);

present:
    /*
     * Present data to user, advancing rcv_nxt through
     * completed sequence space.
     */
    if (TCPS_HAVERCVDSYN(tp->t_state) == 0)
        goto out;
    if (!empty) ti = tp->seg_next;
    if (ti == (struct tcpiphdr *)tp || ti->ti_seq != tp->rcv_nxt)
        goto out;
    if (tp->t_state == TCPS_SYN_RECEIVED && ti->ti_len)
        goto out;
    if (empty) {
        tp->rcv_nxt += ti->ti_len;
        flags = ti->ti_flags & TH_FIN;
        if (so->so_state & SS_CANTRCVMORE)
            m_freem(m0);
        else
            sbappend(&so->so_rcv, m0);

        sorwakeup(so);
        return (flags);
    }
    do {
        tp->rcv_nxt += ti->ti_len;
        flags = ti->ti_flags & TH_FIN;
        remque(ti);
        m = DTOM(ti);
        qp = ti;
        ti = (struct tcpiphdr *)ti->ti_next;
#ifdef WATCHOUT
        MSFREE(qp);
#endif
    } while (ti != (struct tcpiphdr *)tp && ti->ti_seq == tp->rcv_nxt);
    sorwakeup(so);
    return (flags);

drop:
    m_freem(m0);
    return (0);

out:
    if (empty) INSQUE(ti, tp);
    /*
     * If there are more than 10 tcp segments already queued,
     * drop the oldest one from the queue. This sometimes happens
     * when we get flooded with packets from unbuffered writes and

```



```
    * there are gaps in the sequence numbers. (Why are gaps so likely
    * with unbuffered writes?) A retransmit (covering all the
    * one byte segments queued) will occur in a moment.
    */
    cnt = 0;
    for (q = tp->seg_next; q != (struct tcpiphdr *)tp;
         q = (struct tcpiphdr *)q->ti_next)
        cnt++;
    if (cnt > 10) {
        q = tp->seg_next;
        m = DTOM(q);
        remque(q);
#ifdef WATCHOUT
        MSFREE(q);
#endif
        m_freem(m);
    }
    return (0);
}
```

```

/*      %M%      %I%      %E%      */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/system.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "net/in.h"
#include "net/route.h"
#include "net/in_pcb.h"
#include "net/in_systm.h"
#include "net/ip.h"
#include "net/ip_var.h"
#include "net/tcp.h"
#define TCPOUTFLAGS
#include "net/tcp_fsm.h"
#include "net/tcp_seq.h"
#include "net/tcp_timer.h"
#include "net/tcp_var.h"
#include "net/tcpip.h"
#include "net/tcp_debug.h"
#include "errno.h"

char *tcpstates[]; /* XXX */
#define returnerr(e) { error = (e); goto out; }

/*
 * Initial options: indicate max segment length 1/2 of space
 * allocated for receive; if TCPTRUEOOB is defined, indicate
 * willingness to do true out-of-band.
 */
#ifndef TCPTRUEOOB
u_char tcp_initopt[4] = { TCPOPT_MAXSEG, 4, 0x0, 0x0, };
#else
u_char tcp_initopt[6] = { TCPOPT_MAXSEG, 4, 0x0, 0x0, TCPOPT_WILLOOB, 2 };
#endif

/*
 * Tcp output routine: figure out what should be sent and send it.
 */
tcp_output(tp)
    register struct tcpcb *tp;
{
    register struct socket *so = tp->t_inpcb->inp_socket;
    register int len;
    register struct mbuf *m0;
    int off, flags, win, error;
    register struct mbuf *m;
    register struct tcphdr *ti;
    register u_char *opt = 0;
    register unsigned optlen = 0;
    register int sendalot;

    /*
     * Determine length of data that should be transmitted,
     * and flags that will be used.
     * If there is some data or critical controls (SYN, RST)
     * to send, then transmit; otherwise, investigate further.
     */
    #ifdef SIGH
    { extern int intcpoutput; }
    if (intcpoutput) printf("intcpoutput..."); intcpoutput++;
    #endif
    again:
    sendalot = 0;
    off = tp->snd_nxt - tp->snd_una;
    len = MIN(so->so_snd.sb_cc, tp->snd_wnd+tp->t_force) - off;
    if (len < 0) {
        returnerr(0); /* ??? */ /* past FIN */
    }

```

```

    if (len > tp->t_maxseg) {
        len = tp->t_maxseg;
        sendalot = 1;
    }

    flags = tcp_outflags[tp->t_state];
    if ( SEQ_LT((tp->snd_nxt + len), (tp->snd_una + so->so_snd.sb_cc) ) )
        flags &= ~TH_FIN;
    if (flags & (TH_SYN|TH_RST|TH_FIN))
        goto send;
    if (SEQ_GT(tp->snd_up, tp->snd_una))
        goto send;

    /*
     * Sender silly window avoidance. If can send all data,
     * a maximum segment, at least 1/4 of window do it,
     * or are forced, do it; otherwise don't bother.
     */
    if (len) {
        if (len == tp->t_maxseg || off+len >= so->so_snd.sb_cc)
            goto send;
        if (len * 4 >= tp->snd_wnd) /* a lot */
            goto send;
        if (tp->t_force)
            goto send;
    }

    /*
     * Send if we owe peer an ACK.
     */
    if (tp->t_flags&TF_ACKNOW)
        goto send;

    #ifndef TCPTRUEOOB
    /*
     * Send if an out of band data or ack should be transmitted.
     */
    if (tp->t_oobflags&(TCPOOB_OWEACK|TCPOOB_NEEDACK))
        goto send;
    #endif

    /*
     * Calculate available window, and also amount
     * of window known to peer (as advertised window less
     * next expected input.) If this is 35% or more of the
     * maximum possible window, then want to send a segment to peer.
     */
    win = sbsspace(&so->so_rcv);
    if (win > 0 &&
        ((100*(win-(tp->rcv_adv-tp->rcv_nxt))/so->so_rcv.sb_hiwat) >= 35))
        goto send;

    /*
     * TCP window updates are not reliable, rather a polling protocol
     * using 'persist' packets is used to insure receipt of window
     * updates. The three 'states' for the output side are:
     * idle not doing retransmits or persists
     * persisting to move a zero window
     * (re)transmitting and thereby not persisting
     *
     * tp->t_timer[TCPT_PERSIST]
     * Is set when we are in persist state.
     * tp->t_force
     * Is set when we are called to send a persist packet.
     * tp->t_timer[TCPT_REXMT]
     * Is set when we are retransmitting
     * The output side is idle when both timers are zero.
     *
     * If send window is closed, there is data to transmit, and no
     * retransmit or persist is pending, then go to persist state,
     * arranging to force out a byte to get more current window information
     * if nothing happens soon.
     */
    if (tp->snd_wnd == 0 && so->so_snd.sb_cc &&
        tp->t_timer[TCPT_REXMT] == 0 && tp->t_timer[TCPT_PERSIST] == 0) {
        tp->t_rxtshift = 0;

```

```

        tcp_setpersist(tp);
    }

    /*
     * No reason to send a segment, just return.
     */
    returnerr(0);

send:
    /*
     * Grab a header mbuf, attaching a copy of data to
     * be transmitted, and initialize the header from
     * the template for sends on this connection.
     */
    MGET(m, 0);
    if (m == 0) {
#ifdef SIGH
        printf ("cant get header in tcpoutput\n");
#endif
        returnerr(ENOBUFS);
    }
    m->m_off = MMAXOFF - sizeof (struct tcphdr);
    m->m_len = sizeof (struct tcphdr);
    if (len) {
        m->m_next = m_copy(so->so_snd.sb_mb, off, len);
        if (m->m_next == 0)
            len = 0;
    }
    ti = mtod(m, struct tcphdr *);
    if (tp->t_template == 0)
        panic("tcp_output");
    bcopy((caddr_t)tp->t_template, (caddr_t)ti, sizeof (struct tcphdr));

    /*
     * Fill in fields, remembering maximum advertised
     * window for use in delaying messages about window sizes.
     */
    ti->ti_seq = tp->snd_nxt;
    ti->ti_ack = tp->rcv_nxt;
#ifdef WATCHOUT
    ti->ti_seq = htonl(ti->ti_seq);
    ti->ti_ack = htonl(ti->ti_ack);
#endif

    /*
     * Before ESTABLISHED, force sending of initial options
     * unless TCP set to not do any options.
     */
    if (tp->t_state < TCPS_ESTABLISHED) {
        if (tp->t_flags & TF_NOOPT)
            goto noopt;
        opt = tcp_initopt;
        optlen = sizeof (tcp_initopt);
        *(u_short *) (opt + 2) = so->so_rcv.sb_hiwat / 2;
#ifdef SMALLTCP
        *(u_short *) (opt + 2) = 256;
#endif
#ifdef TCPACKMOST
        *(u_short *) (opt + 2) = so->so_rcv.sb_hiwat;
#endif
#ifdef WATCHOUT
        *(u_short *) (opt + 2) = htons(*(u_short *) (opt + 2));
#endif
    } else {
        if (tp->t_tcpopt == 0)
            goto noopt;
        opt = mtod(tp->t_tcpopt, u_char *);
        optlen = tp->t_tcpopt->m_len;
    }
#ifdef TCPTRUEOOB
    if (opt)
#endif
    else
    if (opt || (tp->t_oobflags & (TCPOOB_OWEACK|TCPOOB_NEEDACK)))
    {
        m->m_next = m_get(M_DONTWAIT);
        if (m->m_next == 0) {
            (void) m_free(m);
            m_freem(m0);
            returnerr(ENOBUFS);
        }
        m->m_next->m_next = m0;
        m0 = m->m_next;
        m0->m_off = MMINOFF;
        m0->m_len = optlen;
#ifdef WATCHOUT
        if (opt == tcp_initopt)
            bcopy((caddr_t)opt, mtod(m0, caddr_t), (int)optlen);
        else
            MBCOPY(tp->t_tcpopt, 0, m0, 0, (int)optlen);
#endif
        bcopy((caddr_t)opt, mtod(m0, caddr_t), (int)optlen);
#ifdef TCPTRUEOOB
        opt = (u_char *) (mtod(m0, caddr_t) + optlen);
        if (tp->t_oobflags & TCPOOB_OWEACK) {
            printf("tp %x send OOBACK for %x\n", tp->t_iobseq);
            *opt++ = TCPOPT_OOBACK;
            *opt++ = 3;
            *opt++ = tp->t_iobseq;
            m0->m_len += 3;
            tp->t_oobflags &= ~TCPOOB_OWEACK;
            /* sender should rexmt oob to force ack repeat */
        }
        if (tp->t_oobflags & TCPOOB_NEEDACK) {
            tcp_seq oobseq;
            printf("tp %x send OOBDATA seq %x data %x\n", tp->t_oobseq, tp->t_oobc);
            *opt++ = TCPOPT_OOBDATA;
            *opt++ = 8;
            *opt++ = tp->t_oobseq;
            *opt++ = tp->t_oobc;
            oobseq = tp->t_oobmark - tp->snd_nxt;
#ifdef WATCHOUT
            oobseq = htonl(oobseq);
#endif
            bcopy((caddr_t)oobseq, opt, sizeof oobseq);
            m0->m_len += 8;
            TCPT_RANGESET(tp->t_timer[TCPT_OOBREXMT],
                (tcp_beta*tp->t_srtt)/100, TCPTV_MIN, TCPTV_MAX);
        }
#endif
        while (m0->m_len & 0x3) {
            *opt++ = TCPOPT_EOL;
            m0->m_len++;
        }
        optlen = m0->m_len;
        ti = mtod(m, struct tcphdr *); /* needed for ll */
        ti->ti_off = (sizeof (struct tcphdr) + optlen) >> 2;
    }
    noopt:
    ti->ti_flags = flags;
    win = sbsspace(&so->so_rcv);
#ifdef TCPACKMOST
    if (win < so->so_rcv.sb_hiwat / 4) /* avoid silly window */
#endif
    else
    if (win < so->so_rcv.sb_hiwat) /* avoid silly window */
    #endif
        win = 0;
    if (win > 0)
#ifdef WATCHOUT
        ti->ti_win = htons((u_short)win);
    #endif
    else
        ti->ti_win = win;
    #endif
    if (SEQ_GT(tp->snd_up, tp->snd_nxt)) {
        ti->ti_urp = tp->snd_up - tp->snd_nxt;
#ifdef WATCHOUT
        ti->ti_urp = htons(ti->ti_urp);
#endif
    #endif
        ti->ti_flags |= TH_URG;
    } else

```

```

/*
 * If no urgent pointer to send, then we pull
 * the urgent pointer to the left edge of the send window
 * so that it doesn't drift into the send window on sequence
 * number wraparound.
 */
tp->snd_up = tp->snd_una;          /* drag it along */
/*
 * If anything to send and we can send it all, set PUSH.
 * (This will keep happy those implementations which only
 * give data to the user when a buffer fills or a PUSH comes in.
 */
if (len && off+len == so->so_snd.sb_cc)
    ti->ti_flags |= TH_PUSH;

/*
 * Put TCP length in extended header, and then
 * checksum extended header and data.
 */
if (len + optlen) {
    ti->ti_len = sizeof (struct tcphdr) + optlen + len;
#ifdef WATCHOUT
    ti->ti_len = htons((u_short)ti->ti_len);
#endif
}
mbprint(m, "before tout ck");

ti->ti_sum = in_cksum(m, sizeof (struct tcphdr) + (int)optlen + len);
nprintf("tout sum%x len%x\n", ti->ti_sum, sizeof(struct tcphdr)+optlen+len);

/*
 * In transmit state, time the transmission and arrange for
 * the retransmit. In persist state, reset persist time for
 * next persist.
 */
if (tp->t_force == 0) {
    /*
     * Advance snd_next over sequence space of this segment.
     */
    if (flags & (TH_SYN|TH_FIN))
        tp->snd_next++;
    tp->snd_next += len;

    /*
     * Time this transmission if not a retransmission and
     * not currently timing anything.
     */
    if (SEQ_GT(tp->snd_next, tp->snd_max) && tp->t_rtt == 0) {
        tp->t_rtt = 1;
        tp->t_rtseq = tp->snd_next - len;
    }
    if (SEQ_GT(tp->snd_next, tp->snd_max))
        tp->snd_max = tp->snd_next;

    /*
     * Set retransmit timer if not currently set.
     * Initial value for retransmit timer to tcp_beta*tp->t_rtt.
     * Initialize shift counter which is used for exponential
     * backoff of retransmit time.
     */
    if (tp->t_timer[TCPT_REXMT] == 0 &&
        tp->snd_next != tp->snd_una) {
        TCPT_RANGESET(tp->t_timer[TCPT_REXMT],
            (tcp_beta*tp->t_rtt)/100, TCPTV_MIN, TCPTV_MAX);
        tp->t_rxtshift = 0;
    }
    tp->t_timer[TCPT_PERSIST] = 0;
} else {
    if (SEQ_GT(tp->snd_una+1, tp->snd_max))
        tp->snd_max = tp->snd_una+1;
}

/*
 * Trace.
 */
if (so->so_options & SO_DEBUG)

```

```

tcp_trace(TA_OUTPUT, tp->t_state, tp, ti, 0);

/*
 * Fill in IP length and desired time to live and
 * send to IP level.
 */
((struct ip *)ti)->ip_len = sizeof (struct tcphdr) + optlen + len;
((struct ip *)ti)->ip_ttl = TCP_TTL;
if (error = ip_output(m, tp->t_ipopt, (so->so_options & SO_DONTROUTE) ?
    &routetoif : &tp->t_inpcb->inp_route, 0)) {
    returnerr(error);
}

/*
 * Data sent (as far as we can tell).
 * If this advertises a larger window than any other segment,
 * then remember the size of the advertised window.
 * Drop send for purpose of ACK requirements.
 */
if (win > 0 && SEQ_GT(tp->rcv_next+win, tp->rcv_adv))
    tp->rcv_adv = tp->rcv_next + win;
tp->t_flags &= ~(TF_ACKNOW|TF_DELACK);
if (sendalot && tp->t_force == 0)
    goto again;
returnerr(0);

out:
#ifdef SIGH
    intcpoutput--;
#endif
return (error);
}

tcp_setpersist(tp)
    register struct tcpcb *tp;
{
    if (tp->t_timer[TCPT_REXMT])
        panic("tcp_output REXMT");
    /*
     * Start/restart persistence timer.
     */
    TCPT_RANGESET(tp->t_timer[TCPT_PERSIST],
        ((tcp_beta * tp->t_rtt)/100) << tp->t_rxtshift,
        TCPTV_PERSMIN, TCPTV_MAX);
    tp->t_rxtshift++;
    if (tp->t_rxtshift >= TCP_MAXRXTSHIFT)
        tp->t_rxtshift = 0;
}

```

```

/*      tcp_subr.c      4.27      82/06/20      */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/system.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "net/protosw.h"
#include "net/in.h"
#include "net/route.h"
#include "net/in_pcb.h"
#include "net/in_system.h"
#include "net/if.h"
#include "net/ip.h"
#include "net/ip_var.h"
#include "net/ip_icmp.h"
#include "net/tcp.h"
#include "net/tcp_fsm.h"
#include "net/tcp_seq.h"
#include "net/tcp_timer.h"
#include "net/tcp_var.h"
#include "net/tcpip.h"
#include "errno.h"

/*
 * Tcp initialization
 */
tcp_init()
{
    tcp_iss = 1;          /* wrong */
    tcb.inp_next = tcb.inp_prev = &tcb;
    tcp_alpha = TCP_ALPHA;
    tcp_beta = TCP_BETA;
}

/*
 * Create template to be used to send tcp packets on a connection.
 * Call after host entry created, allocates an mbuf and fills
 * in a skeletal tcp/ip header, minimizing the amount of work
 * necessary when the connection is used.
 */
struct tcphdr *
tcp_template(tp)
    register struct tcpcb *tp;
{
    register struct inpcb *inp = tp->t_inpcb;
    register struct tcphdr *n;

    MSGET(n, struct tcphdr, 1);
    if (n == 0)
        return (0);
    n->ti_pr = IPPROTO_TCP;
    n->ti_len = htons(sizeof (struct tcphdr) - sizeof (struct ip));
    n->ti_src = inp->inp_laddr;
    n->ti_dst = inp->inp_faddr;
    n->ti_sport = inp->inp_lport;
    n->ti_dport = inp->inp_fport;
    n->ti_off = 5;
    return (n);
}

/*
 * Send a single message to the TCP at address specified by
 * the given TCP/IP header.  If flags==0, then we make a copy
 * of the tcphdr at ti and send directly to the addressed host.
 * This is used to force keep alive messages out using the TCP
 * template for a connection tp->t_template.  If flags are given
 * then we send a message back to the TCP which originated the
 * segment ti, and discard the mbuf containing it and any other
 * attached mbufs.
 */
/* In any case the ack and sequence number of the transmitted
 * segment are as specified by the parameters.
 */
tcp_respond(tp, ti, ack, seq, flags)
    register struct tcpcb *tp;
    register struct tcphdr *ti;
    register tcp_seq ack, seq;
    register int flags;
{
    register struct mbuf *m;
    register int win = 0, tlen;
    register struct route *ro = 0;

    if (tp) {
        win = sbspace(&tp->t_inpcb->inp_socket->so_rcv);
        ro = &tp->t_inpcb->inp_route;
    }
    if (flags == 0) {
        m = m_get(M_DONTWAIT);
        if (m == 0)
            return;
        m->m_off = MMINOFF;
        m->m_len = sizeof (struct tcphdr) + 1;
        bcopy((caddr_t)ti, mtod(m, caddr_t), sizeof *ti);
        ti = mtod(m, struct tcphdr *);
        flags = TH_ACK;
        tlen = 1;
    } else {
        m = dtom(ti);
        m_freem(m->m_next);
        m->m_next = 0;
#ifdef newmbufs
        m->m_off = (int)ti - (int)m;
#else
        m->m_off = (int)ti - mtobuf(m, int);
#endif
        m->m_len = sizeof (struct tcphdr);
#define xchg(a,b,type) { type t; t=a; a=b; b=t; }
        xchg(ti->ti_dst.s_addr, ti->ti_src.s_addr, u_long);
        xchg(ti->ti_dport, ti->ti_sport, u_short);
#undef xchg
        tlen = 0;
        ti->ti_next = ti->ti_prev = 0;
        ti->ti_x1 = 0;
        ti->ti_len = sizeof (struct tcphdr) + tlen;
        ti->ti_seq = seq;
        ti->ti_ack = ack;
#ifdef WATCHOUT
        ti->ti_len = htons((u_short)ti->ti_len);
        ti->ti_seq = htonl(ti->ti_seq);
        ti->ti_ack = htonl(ti->ti_ack);
#endif
#ifdef WATCHOUT
        ti->ti_x2 = 0;
        ti->ti_off = sizeof (struct tcphdr) >> 2;
        ti->ti_flags = flags;
        ti->ti_win = win;
#endif
        ti->ti_win = htons(ti->ti_win);
#ifdef WATCHOUT
        ti->ti_urp = 0;
#endif
        /* billn
        ti->ti_sum = 0;
        */
        ti->ti_sum = in_cksum(m, sizeof (struct tcphdr) + tlen);
        ((struct ip *)ti)->ip_len = sizeof (struct tcphdr) + tlen;
        ((struct ip *)ti)->ip_ttl = TCP_TTL;
        (void) ip_output(m, (struct mbuf *)0, ro, 0);
    }
}

/*
 * Create a new TCP control block, making an
 * empty reassembly queue and hooking it to the argument
 * protocol control block.

```

```

*/
struct tcpcb *
tcp_newtcpcb(inp)
    register struct inpcb *inp;
{
    register struct tcpcb *tp;

    MSGET(tp, struct tcpcb, 1);
    if (tp == 0)
        return (0);
    tp->seg_next = tp->seg_prev = (struct tcpiphdr *)tp;
    /* Correction from Dan@sri-tsc
    tp->t_maxseg = 576;          /* satisfy the rest of the world */
    tp->t_maxseg = 512;        /* satisfy the rest of the world */
    tp->t_flags = 0;          /* sends options! */
    tp->t_inpcb = inp;
    inp->inp_ppcb = (caddr_t)tp;
    return (tp);
}

/*
 * Drop a TCP connection, reporting
 * the specified error.  If connection is synchronized,
 * then send a RST to peer.
 */
tcp_drop(tp, errno)
    register struct tcpcb *tp;
    register int errno;
{
    struct socket *so = tp->t_inpcb->inp_socket;

    if (TCPS_HAVERCVDSYN(tp->t_state)) {
        tp->t_state = TCPS_CLOSED;
        (void) tcp_output(tp);
    }
    so->so_error = errno;
    tcp_close(tp);
}

tcp_abort(inp)
    register struct inpcb *inp;
{
    tcp_close((struct tcpcb *)inp->inp_ppcb);
}

#ifdef WATCHOUT
#define ti_mbuf ti_sum
#define DTOM(d) ((struct mbuf *) ((d)->ti_mbuf))
#else
#define DTOM(d) dtom(d)
#endif

/*
 * Close a TCP control block:
 * discard all space held by the tcp
 * discard internet protocol block
 * wake up any sleepers
 */
tcp_close(tp)
    register struct tcpcb *tp;
{
    register struct tcpiphdr *t;
#ifdef WATCHOUT
    register struct tcpiphdr *to;
#endif
    register struct inpcb *inp = tp->t_inpcb;
    register struct socket *so = inp->inp_socket;

    for (t = tp->seg_next; t != (struct tcpiphdr *)tp;) {
        m_freem(DTOM(t));
#ifdef WATCHOUT
        to = t;
#endif
        t = (struct tcpiphdr *)t->ti_next;
#ifdef WATCHOUT
        MSFREE(to);

```

```

#endif
    }
    if (tp->t_template)
        MSFREE(tp->t_template);
    if (tp->t_tcptopt)
        (void) m_free(tp->t_tcptopt);
    if (tp->t_ipopt)
        (void) m_free(tp->t_ipopt);
    MSFREE(tp);
    inp->inp_ppcb = 0;
    soisdisconnected(so);
    in_pcbdetach(inp);
}

tcp_drain()
{
}

tcp_ctlinput(cmd, arg)
    register int cmd;
    register caddr_t arg;
{
    register struct in_addr *sin;
    extern u_char inetctlerrmap[];

    if (cmd < 0 || cmd > PRC_NCMTDS)
        return;
    switch (cmd) {

    case PRC_ROUTEDEAD:
        break;

    case PRC_QUENCH:
        break;

    /* these are handled by ip */
    case PRC_IFDOWN:
    case PRC_HOSTDEAD:
    case PRC_HOSTUNREACH:
        break;

    default:
        sin = &((struct icmp *)arg)->icmp_ip.ip_dst;
        in_pcbnotify(&tcpcb, sin, inetctlerrmap[cmd], tcp_abort);
    }
}

```

```

/* tcp_timer.c 4.23 82/06/20 */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/system.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "net/protosw.h"
#include "net/in.h"
#include "net/route.h"
#include "net/in_pcb.h"
#include "net/in_system.h"
#include "net/if.h"
#include "net/ip.h"
#include "net/ip_var.h"
#include "net/tcp.h"
#include "net/tcp_fsm.h"
#include "net/tcp_seq.h"
#include "net/tcp_timer.h"
#include "net/tcp_var.h"
#include "net/tcpip.h"
#include "errno.h"

int tcpsndelack = 0;
#ifdef SIGH
int infasttimo = 0;
int inslowtimo = 0;
#endif
/*
 * Fast timeout routine for processing delayed acks
 */
tcp_fasttimo()
{
    register struct inpcb *inp;
    register struct tcpcb *tp;
    register int s = splnet();

#ifdef SIGH
    if (infasttimo) printf("infasttimo..."); else infasttimo++;
#endif
    inp = tcb.inp_next;
    if (inp)
        for (; inp != &tcb; inp = inp->inp_next)
            if ((tp = (struct tcpcb *)inp->inp_ppcb) &&
                (tp->t_flags & TF_DELACK)) {
                tp->t_flags &= ~TF_DELACK;
                tp->t_flags |= TF_ACKNOW;
                (void) tcp_output(tp);
            }
#ifdef SIGH
    infasttimo--;
#endif
    splx(s);
}

/*
 * Tcp protocol timeout routine called every 500 ms.
 * Updates the timers in all active tcb's and
 * causes finite state machine actions if timers expire.
 */
tcp_slowtimo()
{
    register struct inpcb *ip, *ipnxt;
    register struct tcpcb *tp;
    register int s = splnet();
    register int i;

    /*
     * Search through tcb's and update active timers.
     */
#ifdef SIGH
    if (inslowtimo) printf("inslowtimo..."); else inslowtimo++;

```

```

#endif
    ip = tcb.inp_next;
    if (ip == 0) {
        splx(s);
#ifdef SIGH
        inslowtimo--;
#endif
        return;
    }
    while (ip != &tcb) {
        tp = intotcp(ip);
        if (tp == 0)
            continue;
        ipnxt = ip->inp_next;
        for (i = 0; i < TCPT_NTIMERS; i++) {
            if (tp->t_timer[i] && --tp->t_timer[i] == 0) {
                (void) tcp_usrreq(tp->t_inpcb->inp_socket,
                    PRU_SLOWTIMO, (struct mbuf *)0,
                    (caddr_t)i);
                if (ipnxt->inp_prev != ip)
                    goto tpgone;
            }
        }
        tp->t_idle++;
        if (tp->t_rtt)
            tp->t_rtt++;
    }
    tpgone:
        ip = ipnxt;
        tcp_iss += TCP_ISSINCR/PR_SLOWHZ; /* increment iss */
        splx(s);
#ifdef SIGH
        inslowtimo--;
#endif
}

/*
 * Cancel all timers for TCP tp.
 */
tcp_canceltimers(tp)
    register struct tcpcb *tp;
{
    register int i;

    for (i = 0; i < TCPT_NTIMERS; i++)
        tp->t_timer[i] = 0;
}

int tcp_backoff[TCPT_MAXRXTSHIFT] = /* scaled by 10 */
    { 10, 12, 14, 17, 20, 30, 50, 80, 160, 320 };
int tcpexmtprint = 1;
int tcpexprexmtbackoff = 0;
/*
 * TCP timer processing.
 */
tcp_timers(tp, timer)
    register struct tcpcb *tp;
    register int timer;
{
    switch (timer) {
        /*
         * 2 MSL timeout in shutdown went off. Delete connection
         * control block.
         */
        case TCPT_2MSL:
            tcp_close(tp);
            return;

        /*
         * Retransmission timer went off. Message has not
         * been acked within retransmit interval. Back off
         * to a longer retransmit interval and retransmit all
         * unacknowledged messages in the window.
         */
    }
}

```

```

case TCPT_REXMT:
    tp->t_rxtshift++;
    if (tp->t_rxtshift > TCP_MAXRXTSHIFT) {
        tcp_drop(tp, ETIMEDOUT);
        return;
    }
    TCPT_RANGESET(tp->t_timer[TCPT_REXMT],
        (int)tp->t_srtt/10, TCPTV_MIN, TCPTV_MAX);
    if (tcpexprexmtbackoff) {
        TCPT_RANGESET(tp->t_timer[TCPT_REXMT],
            tp->t_timer[TCPT_REXMT] << tp->t_rxtshift,
            TCPTV_MIN, TCPTV_MAX);
    } else {
        TCPT_RANGESET(tp->t_timer[TCPT_REXMT],
            (tp->t_timer[TCPT_REXMT] *
            tcp_backoff[tp->t_rxtshift - 1])/10,
            TCPTV_MIN, TCPTV_MAX);
    }
    if (tcprexmtprint)
        printf("rexmt set to %d\n", tp->t_timer[TCPT_REXMT]);
    tp->snd_nxt = tp->snd_una;
    /* this only transmits one segment! */
    (void) tcp_output(tp);
    return;

/*
 * Persistence timer into zero window.
 * Force a byte to be output, if possible.
 */
case TCPT_PERSIST:
    tcp_setpersist(tp);
    tp->t_force = 1;
    (void) tcp_output(tp);
    tp->t_force = 0;
    return;

/*
 * Keep-alive timer went off; send something
 * or drop connection if idle for too long.
 */
case TCPT_KEEP:
    if (tp->t_state < TCPS_ESTABLISHED)
        goto dropit;
    if (tp->t_inpcb->inp_socket->so_options & SO_KEEPAIVE) {
        if (tp->t_idle >= TCPTV_MAXIDLE)
            goto dropit;
        /*
         * Saying tp->rcv_nxt-1 lies about what
         * we have received, and by the protocol spec
         * requires the correspondent TCP to respond.
         * Saying tp->snd_una-1 causes the transmitted
         * byte to lie outside the receive window; this
         * is important because we don't necessarily
         * have a byte in the window to send (consider
         * a one-way stream!)
         */
        tcp_respond(tp,
            tp->t_template, tp->rcv_nxt-1, tp->snd_una-1, 0);
    } else
        tp->t_idle = 0;
    tp->t_timer[TCPT_KEEP] = TCPTV_KEEP;
    return;
dropit:
    tcp_drop(tp, ETIMEDOUT);
    return;

#ifdef TCPTRUECOB
/*
 * Out-of-band data retransmit timer.
 */
case TCPT_OOBREXMT:
    if (tp->t_flags & TF_NOOPT)
        return;
    (void) tcp_output(tp);
    TCPT_RANGESET(tp->t_timer[TCPT_OOBREXMT],
        (2 * tp->t_srtt)/10, TCPTV_MIN, TCPTV_MAX);
#endif
return;
#endif
}
}

```



```

/*      tcp_usrreq.c      1.59      82/06/20      */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/system.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "net/protosw.h"
#include "net/in.h"
#include "net/route.h"
#include "net/in_pcb.h"
#include "net/in_system.h"
#include "net/if.h"
#include "net/ip.h"
#include "net/ip_var.h"
#include "net/tcp.h"
#include "net/tcp_fsm.h"
#include "net/tcp_seq.h"
#include "net/tcp_timer.h"
#include "net/tcp_var.h"
#include "net/tcpip.h"
#include "net/tcp_debug.h"
#include "errno.h"

/*
 * TCP protocol interface to socket abstraction.
 */
extern char *tcpstates[];
struct tcpcb *tcp_newtcpcb();

/*
 * Process a TCP user request for TCP tb.  If this is a send request
 * then m is the mbuf chain of send data.  If this is a timer expiration
 * (called from the software clock routine), then timertype tells which timer.
 */
tcp_usrreq(so, req, m, addr)
    struct socket *so;
    int req;
    struct mbuf *m;
    caddr_t addr;
{
    register struct inpcb *inp = sotoinpcb(so);
    register struct tcpcb *tp;
    int s = splnet();
    int error = 0;
    int ostate = 0;
    extern struct tcpcb * tcp_disconnect();
    extern struct tcpcb * tcp_usrclosed();

    /*
     * When a TCP is attached to a socket, then there will be
     * a (struct inpcb) pointed at by the socket, and this
     * structure will point at a subsidiary (struct tcpcb).
     * The normal sequence of events is:
     *   PRU_ATTACH          creating these structures
     *   PRU_CONNECT        connecting to a remote peer
     *   (PRU_SEND|PRU_RCVD)* exchanging data
     *   PRU_DISCONNECT    disconnecting from remote peer
     *   PRU_DETACH        deleting the structures
     * With the operations from PRU_CONNECT through PRU_DISCONNECT
     * possible repeated several times.
     *
     * MULTIPLE CONNECTS ARE NOT YET IMPLEMENTED.
     */
    if (inp == 0 && req != PRU_ATTACH) {
        splx(s);
        return (EINVAL);          /* XXX */
    }
    if (inp) {
        tp = intotcpcb(inp);
#ifdef KPROF
        tcp_accounts[tp->t_state][req]++;
#endif
}
}

#endif
    ostate = tp->t_state;
}
switch (req) {
/*
 * TCP attaches to socket via PRU_ATTACH, reserving space,
 * and internet and TCP control blocks.
 * If the socket is to receive connections,
 * then the LISTEN state is entered.
 */
case PRU_ATTACH:
    if (inp) {
        error = EISCONN;
        break;
    }
    error = tcp_attach(so, (struct sockaddr *)addr);
    if (error)
        break;
    if ((so->so_options & SO_DONTLINGER) == 0)
        so->so_linger = TCP_LINGERTIME;
    tp = sototcpcb(so);
    break;

/*
 * PRU_DETACH detaches the TCP protocol from the socket.
 * If the protocol state is non-embryonic, then can't
 * do this directly: have to initiate a PRU_DISCONNECT,
 * which may finish later; embryonic TCB's can just
 * be discarded here.
 */
case PRU_DETACH:
    tp = tcp_disconnect(tp);
    break;

/*
 * Initiate connection to peer.
 * Create a template for use in transmissions on this connection.
 * Enter SYN_SENT state, and mark socket as connecting.
 * Start keep-alive timer, and seed output sequence space.
 * Send initial segment on connection.
 */
case PRU_CONNECT:
    error = in_pcbconnect(inp, (struct sockaddr_in *)addr);
    if (error)
        break;
    tp->t_template = tcp_template(tp);
    if (tp->t_template == 0) {
        in_pcbdisconnect(inp);
        error = ENOBUFS;
        break;
    }
    soisconnecting(so);
    tp->t_state = TCPS_SYN_SENT;
    tp->t_timer[TCPT_KEEP] = TCPTV_KEEP;
    tp->iss = tcp_iss; tcp_iss += TCP_ISSINCR/2;
    tcp_sendseqinit(tp);
    error = tcp_output(tp);
    break;

/*
 * Initiate disconnect from peer.
 * If connection never passed embryonic stage, just drop;
 * else if don't need to let data drain, then can just drop anyways,
 * else have to begin TCP shutdown process: mark socket disconnecting,
 * drain unread data, state switch to reflect user close, and
 * send segment (e.g. FIN) to peer.  Socket will be really disconnected
 * when peer sends FIN and acks ours.
 *
 * SHOULD IMPLEMENT LATER PRU_CONNECT VIA REALLOC TCPCB.
 */
case PRU_DISCONNECT:
    tp = tcp_disconnect(tp);
    break;

/*

```

```

* Accept a connection. Essentially all the work is
* done at higher levels; just return the address
* of the peer, storing through addr.
*/
case PRU_ACCEPT: {
    struct sockaddr_in *sin = (struct sockaddr_in *)addr;

    if (sin) {
        bzero((caddr_t)sin, sizeof (*sin));
        sin->sin_family = AF_INET;
        sin->sin_port = inp->inp_fport;
        sin->sin_addr = inp->inp_faddr;
    }
}
break;

/*
* Mark the connection as being incapable of further output.
*/
case PRU_SHUTDOWN:
    socantsendmore(so);
    if (tp = tcp_usrclosed(tp))
        error = tcp_output(tp);
    break;

/*
* After a receive, possibly send window update to peer.
*/
case PRU_RCVD:
    (void) tcp_output(tp);
    break;

/*
* Do a send by putting data in output queue and updating urgent
* marker if URG set. Possibly send more data.
*/
case PRU_SEND:
    sbappend(&so->so_snd, m);
#endif
    if (tp->t_flags & TF_PUSH)
        tp->snd_end = tp->snd_una + so->so_snd.sb_cc;
#endif
    error = tcp_output(tp);
    break;

/*
* Abort the TCP.
*/
case PRU_ABORT:
    tcp_drop(tp, ECONNABORTED);
    tp = 0;
    break;

/* SOME AS YET UNIMPLEMENTED HOOKS */
case PRU_CONTROL:
    error = EOPNOTSUPP;
    break;

case PRU_SENSE:
    error = EOPNOTSUPP;
    break;

/* END UNIMPLEMENTED HOOKS */

case PRU_RCVOOB:
    if (so->so_oobmark == 0 &&
        (so->so_state & SS_RCVATMARK) == 0) {
        error = EINVAL;
        break;
    }
    if ((tp->t_oobflags & TCPOOB_HAVEDATA) == 0) {
        error = EWOULDBLOCK;
        break;
    }
    *mtod(m, caddr_t) = tp->t_ioobc;
    break;

case PRU_SENDOOB:
#ifdef TCPTRUEOOB
    if (tp->t_flags & TF_DOOOB) {
        tp->t_oobseq++;
        tp->t_oobc = *mtod(m, caddr_t);
        tp->t_oobmark = tp->snd_una + so->so_snd.sb_cc;
        printf("sendoob seq now %x oobc %x\n", tp->t_oobseq, tp->t_oobc);
        tp->t_oobflags |= TCPOOB_NEEDACK;
        /* what to do ...? */
        if (error = tcp_output(tp))
            break;
    }
#endif
    if (sbspace(&so->so_snd) < -512) {
        error = ENOBUFS;
        break;
    }
    tp->snd_up = tp->snd_una + so->so_snd.sb_cc + 1;
    sbappend(&so->so_snd, m);
#endif
    if (tp->t_flags & TF_PUSH)
        tp->snd_end = tp->snd_una + so->so_snd.sb_cc;
#endif
    tp->t_force = 1;
    error = tcp_output(tp);
    tp->t_force = 0;
    break;

case PRU_SOCKADDR:
    in_setsockaddr((struct sockaddr_in *)addr, inp);
    break;

/*
* TCP slow timer went off; going through this
* routine for tracing's sake.
*/
case PRU_SLOWTIMO:
    tcp_timers(tp, (int)addr);
    req |= (int)addr << 8; /* for debug's sake */
    break;

default:
    panic("tcp_usrreq");
}
if (so->so_options & SO_DEBUG)
    tcp_trace(TA_USER, ostate, tp, (struct tcpcb *)0, req);
splx(s);
return (error);
}

int tcp_sendspace = 1024*2;
int tcp_recvspace = 1024*2;
/*
* Attach TCP protocol to socket, allocating
* internet protocol control block, tcp control block,
* bufer space, and entering LISTEN state if to accept connections.
*/
tcp_attach(so, sa)
    struct socket *so;
    struct sockaddr *sa;
{
    register struct tcpcb *tp;
    struct inpcb *inp;
    int error;

    error = in_pcbattach(so, &tcp,
        tcp_sendspace, tcp_recvspace, (struct sockaddr_in *)sa);
    if (error)
        return (error);
    inp = (struct inpcb *)so->so_pcb;
    tp = tcp_newtcpcb(inp);
    if (tp == 0) {
        in_pcbdetach(inp);
        return (ENOBUFS);
    }
    if (so->so_options & SO_ACCEPTCONN)

```

```

        tp->t_state = TCPS_LISTEN;
    else
        tp->t_state = TCPS_CLOSED;
    return (0);
}

/*
 * Initiate (or continue) disconnect.
 * If embryonic state, just send reset (once).
 * If not in "let data drain" option, just drop.
 * Otherwise (hard), mark socket disconnecting and drop
 * current input data; switch states based on user close, and
 * send segment to peer (with FIN).
 */
struct tcpcb *
tcp_disconnect(tp)
    struct tcpcb *tp;
{
    struct socket *so = tp->t_inpcb->inp_socket;

    if (tp->t_state <= TCPS_LISTEN) {
        tcp_close(tp);
        tp = 0;
    } else if (so->so_linger == 0) {
        tcp_drop(tp, 0);
        tp = 0;
    } else {
        soisdisconnecting(so);
        sbflush(&so->so_rcv);
        if (tp = tcp_usrclosed(tp))
            (void) tcp_output(tp);
    }
    return (tp);
}

/*
 * User issued close, and wish to trail through shutdown states:
 * if never received SYN, just forget it.  If got a SYN from peer,
 * but haven't sent FIN, then go to FIN_WAIT_1 state to send peer a FIN.
 * If already got a FIN from peer, then almost done; go to LAST_ACK
 * state.  In all other cases, have already sent FIN to peer (e.g.
 * after PRU_SHUTDOWN), and just have to play tedious game waiting
 * for peer to send FIN or not respond to keep-alives, etc.
 * We can let the user exit from the close as soon as the FIN is acked.
 */
struct tcpcb *
tcp_usrclosed(tp)
    struct tcpcb *tp;
{
    switch (tp->t_state) {

    case TCPS_LISTEN:
    case TCPS_SYN_SENT:
        tp->t_state = TCPS_CLOSED;
        tcp_close(tp);
        tp = 0;
        break;

    case TCPS_SYN_RECEIVED:
    case TCPS_ESTABLISHED:
        tp->t_state = TCPS_FIN_WAIT_1;
        break;

    case TCPS_CLOSE_WAIT:
        tp->t_state = TCPS_LAST_ACK;
        break;
    }
    if (tp && tp->t_state >= TCPS_FIN_WAIT_2)
        soisdisconnected(tp->t_inpcb->inp_socket);
    return (tp);
}

```

```

/*
 * Lisa INS8250A device driver
 *
 * Copyright 1984 UniSoft Corporation
 */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/file.h"
#include "sys/tty.h"
#include "sys/termio.h"
#include "sys/conf.h"
#include "sys/sysinfo.h"
#include "sys/var.h"
#include "setjmp.h"
#include "sys/reg.h"
#include "sys/mmu.h"
#include "sys/proc.h"

int tepoch();

/*
 * structure to access Tecmar device registers
 */
struct tedevice {
    char fill1;
    char te_rbr; /* +1 data register */
#define te_dvlsb te_rbr /* lsb of divisor latch */
    char fill2;
    char te_ier; /* +3 interrupt enable register */
#define te_dvmsb te_ier /* msb of divisor latch */
    char fill3;
    char te_iir; /* +5 interrupt id register */
    char fill4;
    char te_lcr; /* +7 line control register */
    char fill5;
    char te_mcr; /* +9 modem control register */
    char fill6;
    char te_lsr; /* +11 line status register */
    char fill7;
    char te_msr; /* +13 modem status register */
    char fill8;
    char te_scrat; /* +15 scratch register (8250-A only) */
    char fill[0x200-16]; /* sized to make tedevice be 0x200 long */
};

/*
 * structure to access the interrupt reset bit
 */
struct teidevice {
    struct tedevice fill[7]; /* filler */
    char skip;
    char te_intr; /* interrupt reset location */
};

/*
 * array used to remap ivec interrupt board slot number to tty slot
 */
int te_remap[3];

/*
 * Slot id to interrupt reset address
 */
struct teidevice *te_idevice[3] = {
    (struct teidevice *) (STDIO),
    (struct teidevice *) (STDIO+0x4000),
    (struct teidevice *) (STDIO+0x8000)
};

extern struct tty te_tty[1];
extern struct ttyptr te_ttptr[1];

```

```

extern char te_dparam[1];
extern char te_modem[1];
extern int te_cnt;

#define MODEM 0x80 /* modem control on bit */
#define teved (d)&0x7f /* from unix device number to device */

#define BAUD50 2304
#define BAUD75 1356
#define BAUD110 1047
#define BAUD134 857
#define BAUD150 768
#define BAUD300 384
#define BAUD600 192
#define BAUD1200 96
#define BAUD1800 64
#define BAUD2000 58
#define BAUD2400 48
#define BAUD3600 32
#define BAUD4800 24
#define BAUD7200 16
#define BAUD9600 12
#define BAUD19200 6
#define BAUD38400 3
#define BAUD56000 2

/* -1 means hangup, -2 means invalid */
int tebaudmap[] = {
    -1, BAUD50, BAUD75, BAUD110, BAUD134, BAUD150, BAUD180,
    BAUD300, BAUD600, BAUD1200, BAUD1800, BAUD2400,
    BAUD4800, BAUD9600, BAUD19200, BAUD38400
};

/* Interrupt enable register bits */
#define ERBFI 0x01 /* Enable received data available interrupt */
#define ETBEI 0x02 /* Enable transmitter enable holding register empty interrupt */
#define ELSI 0x04 /* Enable receiver line status interrupt */
#define EDSSI 0x08 /* Enable modem status interrupt */

/* Interrupt ident register bits */
#define IRQ 0x01 /* Interrupt request, 0 if interrupt pending */
#define THE 0x02 /* Transmitter holding register empty */
#define IID 0x06 /* Interrupt ID bit mask */
#define RID 0x04 /* Interrupt receive bit mask */

/* Line control register bits */
#define BITS5 0x00 /* 5 bits */
#define BITS6 0x01 /* 6 bits */
#define BITS7 0x02 /* 7 bits */
#define BITS8 0x03 /* 8 bits */
#define STOP1 0x00 /* One stop bit */
#define STOP2 0x04 /* Two stop bit */
#define PEN 0x08 /* Parity enable */
#define EPS 0x10 /* Even parity select */
#define SPS 0x20 /* Stick parity */
#define SBRK 0x40 /* Set break */
#define DLAB 0x80 /* Divisor latch access. i/o direction bit */

/* Modem control register bits */
#define DTR 0x01 /* Data terminal ready */
#define RTS 0x02 /* Request to send */

/* Line status register bits */
#define DATARDY 0x01 /* Data ready */
#define OV_ERR 0x02 /* Overrun error on receiver */
#define PE_ERR 0x04 /* Parity error */
#define FR_ERR 0x08 /* Framing error */
#define BR_INT 0x10 /* Break interrupt */
#define THRE 0x20 /* Transmitter holding register */
#define TEMT 0x40 /* Transmitter empty */

/* Modem status register bits */
#define DCTS 0x01 /* Delta clear to send */
#define DDSR 0x02 /* Delta data set ready */
#define TERI 0x04 /* Trailing edge ring indicator */
#define DDCD 0x08 /* Delta data carrier detect */

```

```

#define CTS    0x10 /* Clear to send */
#define DSR    0x20 /* Data set ready */
#define RI     0x40 /* Ring indicator */
#define DCD    0x80 /* Data carrier detect */

int teslotsused = 0;

/*
 * Initialize the baud rate
 * slot = 0, 1, or 2
 */
teinit(slot)
{
    register i, val;

    if (teslotsused+4 > te_cnt) {
        printf("\n\nSystem only configured for %d tecmar ports\n\n", te_cnt);
        return(1);
    }
    te_remap[slot] = teslotsused;
    val = STDIO + slot*0x4000 + 0x200;
    for (i = teslotsused; i < teslotsused+4; i++) {
        te_ttptr[i].tt_addr = val;
        te_ttptr[i].tt_tty = ste_tty[i];
        val += 0x200;
    }
    teslotsused += 4;
    return(0);
}

/* ARGSUSED */
teopen(dev, flag)
dev_t dev;
{
    register struct tedevice *addr;
    register struct tty *tp;
    register d;

#ifdef SINGLEUSER
    register struct proc *p;
#endif

    d = tedevice(dev);
    if (d >= te_cnt) {
        u.u_error = ENXIO;
        return;
    }
    tp = te_ttptr[d].tt_tty;
#ifdef SINGLEUSER
    p = u.u_proc;
    if ((p->p_pid == p->p_pgrp)
        && (u.u_ttyp == NULL)
        && (tp->t_pgrp == 0)) {
        u.u_error = ENOTTY;
        return;
    }
#endif
    addr = (struct tedevice *)te_ttptr[d].tt_addr;
    if (tp == 0 || addr == 0) {
        u.u_error = ENXIO;
        return;
    }
    tp->t_index = d;
    SPL5();
    if ((tp->t_state & (ISOPEN|WOPEN)) == 0) {
        tp->t_proc = tepoch;
        ttinit(tp);
        tp->t_iflag = ICRNL | ISTRIP;
        tp->t_oflag = OPOST | ONLCR | TAB3;
        tp->t_lflag = ISIG | ICANON | ECHO | ECHOK;
        tp->t_cflag = sspeed | CS8 | CREAD | HUPCL;
        teparam(dev);
    }
    te_modem[d] = dev & MODEM;
    if ((dev & MODEM) == 0 || addr->te_msr & DSR)
        tp->t_state |= CARR_ON;
    else

```

```

        tp->t_state &= ~CARR_ON;
    if (!(flag & FNDELAY))
        while ((tp->t_state & CARR_ON) == 0) {
            tp->t_state |= WOPEN;
            (void) sleep((caddr_t)&tp->t_rawq, TTOPRI);
        }
    SPL0();
    (*linesw[tp->t_line].l_open)(tp);
}

/* ARGSUSED */
teclose(dev, flag)
dev_t dev;
int flag;
{
    register struct tedevice *addr;
    register struct tty *tp;
    register d;

    d = tedevice(dev);
    tp = te_ttptr[d].tt_tty;
    (*linesw[tp->t_line].l_close)(tp);
    if (tp->t_cflag & HUPCL) {
        addr = (struct tedevice *)te_ttptr[d].tt_addr;
        d = 0;
        addr->te_mcr = d;
    }
}

teread(dev)
dev_t dev;
{
    register struct tty *tp;

    tp = te_ttptr[tedevice(dev)].tt_tty;
    (*linesw[tp->t_line].l_read)(tp);
}

tewrite(dev)
dev_t dev;
{
    register struct tty *tp;

    tp = te_ttptr[tedevice(dev)].tt_tty;
    (*linesw[tp->t_line].l_write)(tp);
}

teproc(tp, cmd)
register struct tty *tp;
{
    register struct ccblock *tbuf;
    register struct tedevice *addr;
    register dev_t dev;
    int s;
    extern ttrstrt();

    s = spltty();
    dev = tp->t_index;
    addr = (struct tedevice *)te_ttptr[dev].tt_addr;
    switch (cmd) {

    case T_TIME:
        tp->t_state &= ~TIMEOUT;
        goto start;

    case T_WFLUSH:
        tbuf = &tp->t_tbuf;
        tbuf->c_size -= tbuf->c_count;
        tbuf->c_count = 0;
        /* fall through */

    case T_RESUME:
        tp->t_state &= ~TTSTOP;
        goto start;

    case T_OUTPUT:
start:

```

```

if (tp->t_state & (TTSTOP|TIMEOUT|BUSY)) {
    /* if ((tp->t_state & BUSY) == 0) {
        addr = (struct tedevice *)((int)addr & 0xFCC000);
        ((struct teidevice *)addr)->te_intr = dev;
    } */
    break;
}
if (tp->t_state & TTXOFF) {
    tp->t_state &= ~TTXOFF;
    tp->t_state |= BUSY;
    addr->te_rbr = CSTOP;
    break;
}
if (tp->t_state & TTXON) {
    tp->t_state &= ~TTXON;
    tp->t_state |= BUSY;
    addr->te_rbr = CSTART;
    break;
}
tbuf = &tp->t_tbuf;
if ((tbuf->c_ptr == 0) || (tbuf->c_count == 0)) {
    if (tbuf->c_ptr)
        tbuf->c_ptr -= tbuf->c_size - tbuf->c_count;
    if (!(CPRES & (*linesw[tp->t_line].l_output)(tp))) {
        break;
    }
}
tp->t_state |= BUSY;
addr->te_ier = ERBFI | ETBEI | ELSI | EDSSI;
addr->te_rbr = *tbuf->c_ptr++;
tbuf->c_count--;
break;

case T_SUSPEND:
    tp->t_state |= TTSTOP;
    break;

case T_BLOCK:
    tp->t_state &= ~TTXON;
    tp->t_state |= TBLOCK;
    tp->t_state |= TTXOFF;
    goto start;

case T_RFLUSH:
    if (!(tp->t_state & TBLOCK))
        break;
    /* fall through */

case T_UNBLOCK:
    tp->t_state &= ~(TTXOFF|TBLOCK);
    tp->t_state |= TTXON;
    goto start;

case T_BREAK:
    tp->t_state |= TIMEOUT;
    timeout((ttrstrt, (caddr_t)tp, v.v_hz>>2));
    break;
}
splx(s);
}

teioctl(dev, cmd, arg, mode)
dev_t dev;
{
    if (ttiocom(te_ttptr[tedev(dev)].tt_tty, cmd, arg, mode))
        teparam(dev);
}

teparam(dev)
register dev_t dev;
{
    register struct tty *tp;
    register struct tedevice *addr;
    register int s, speed, oldpri;
    char c;

    tp = te_ttptr[tedev(dev)].tt_tty;
    addr = (struct tedevice *)te_ttptr[tedev(dev)].tt_addr;
    /* check for invalid speed */
    if ((speed = tebaudmap[tp->t_cflag & CBAUD]) == -2) {
        u.u_error = EINVAL;
        return;
    }
    s = 0;
    /*
     * hangup the line
     */
    if (speed == -1) {
        addr->te_mcr = s;
        return;
    }
    if (tp->t_state & BUSY) {
        te_dparam[tedev(dev)] = 1;
        return;
    }
    /*
     * set new speed
     */
    oldpri = spltty();
    addr->te_lcr = DLAB;
    addr->te_dvlsb = speed;
    addr->te_dvmsb = speed >> 8;
    addr->te_lcr = s;

    /*
     * set line control information
     */
    if ((tp->t_cflag & CSIZE) == CS8)
        s |= BITS8;
    else if ((tp->t_cflag & CSIZE) == CS7)
        s |= BITS7;
    else if ((tp->t_cflag & CSIZE) == CS6)
        s |= BITS6;
    else if ((tp->t_cflag & CSIZE) == CS5)
        s |= BITS5;
    if (tp->t_cflag & CSTOPB)
        s |= STOP2;
    if (tp->t_cflag & PARENB)
        if ((tp->t_cflag & PARODD) == 0)
            s |= PEN|EPS;
    addr->te_lcr = s;

    /*
     * set modem control information
     */
    addr->te_mcr = DTR | RTS;

    /*
     * enable interrupts
     */
    addr->te_ier = ERBFI | ETBEI | ELSI | EDSSI;

    /*
     * reset pending interrupts
     */
    c = addr->te_rbr;
    c = addr->te_lsr;
    c = addr->te_msr;
    c = addr->te_iir;

    splx(oldpri);
}

/* VARARGS */
teintr(ap)
struct args *ap;
{
    register struct tedevice *addr;

```

```

register struct ccblock *cbp;
register struct tty *tp;
register int c, lcnt, flg, iir, lsr;
register char ctmp;
int i, any;
struct teidevice *laddr;
int index, s;
char lbuf[3];

s = spl5();
index = te_remap[ap->a_dev];
laddr = te_idevice[ap->a_dev];
c = 0;
laddr->te_intr = c; /* reset master interrupt */
again:
any = 0;
for (i = index; i < index+4; i++) {
    addr = (struct teidevice *)te_ttptr[i].tt_addr;
restart:
    iir = addr->te_iir;
    if (iir & IRQ)
        continue;
    tp = te_ttptr[i].tt_tty;
    lsr = addr->te_lsr;
    if (iir & RID) {
        sysinfo.rovint++;
        c = addr->te_rbr & 0xFF;
        if (tp->t_rbuf.c_ptr == NULL) {
            any++;
            goto restart;
        }
        if ((lsr & DATARDY) == 0)
            c = 0;
        if (tp->t_iflag & IXON) {
            ctmp = c & 0177;
            if (tp->t_state & TTSTOP) {
                if (ctmp == CSTART || tp->t_iflag & IXANY)
                    (*tp->t_proc)(tp, T_RESUME);
            } else {
                if (ctmp == CSTOP)
                    (*tp->t_proc)(tp, T_SUSPEND);
            }
            if (ctmp == CSTART || ctmp == CSTOP) {
                any++;
                goto restart;
            }
        }
        /*
         * Check for errors
         */
        lcnt = 1;
        flg = tp->t_iflag;
        if (lsr & (PE_ERR|FR_ERR|OV_ERR|BR_INT)) {
            if ((lsr & PE_ERR) && (flg & INPCK))
                c |= PERROR;
            if (lsr & OV_ERR)
                c |= OVERRUN;
            if (lsr & FR_ERR)
                c |= FRERROR;
            if (lsr & BR_INT)
                c = OVERRUN; /* reset char on a break */
        }
        if (c & (FRERROR|PERROR|OVERRUN)) {
            if ((c & 0377) == 0) {
                if (flg & IGNBRK) {
                    any++;
                    goto restart;
                }
                if (flg & BRKINT) {
                    signal(tp->t_pgrp, SIGINT);
                    ttyflush(tp, (FREAD|FWRITE));
                    any++;
                    goto restart;
                }
            } else {
                if (flg & IGNPAR) {
                    any++;
                    goto restart;
                }
            }
        }
        if (flg & PARMRK) {
            lbuf[2] = 0377;
            lbuf[1] = 0;
            lcnt = 3;
            sysinfo.rawch += 2;
        } else
            c = 0;
    } else {
        if (flg & ISTRIP)
            c &= 0177;
        else {
            if (c == 0377 && flg & PARMRK) {
                lbuf[1] = 0377;
                lcnt = 2;
            }
        }
    }
    /*
     * Stash character in r_buf
     */
    cbp = &tp->t_rbuf;
    if (cbp->c_ptr == NULL) {
        any++;
        goto restart;
    }
    if (lcnt != 1) {
        lbuf[0] = c;
        while (lcnt) {
            *cbp->c_ptr++ = lbuf[--lcnt];
            if (--cbp->c_count == 0) {
                cbp->c_ptr -= cbp->c_size;
                (*linesw[tp->t_line].l_input)(tp);
            }
        }
        if (cbp->c_size != cbp->c_count) {
            cbp->c_ptr -= cbp->c_size - cbp->c_count;
            (*linesw[tp->t_line].l_input)(tp);
        }
    } else {
        *cbp->c_ptr = c;
        cbp->c_count--;
        (*linesw[tp->t_line].l_input)(tp);
    }
    any++;
    goto restart;
}
if (iir & THE) {
    sysinfo.xmtint++;
    tp->t_state &= ~BUSY;
    if (te_dparam[i]) {
        te_dparam[i] = 0;
        teparam(i);
    }
    tepoch(tp, T_OUTPUT);
} else {
    /*
     * must be a modem transition interrupt
     */
    temodem(tp, addr->te_msr);
}
any++;
goto restart;
}
if (any != 0)
    goto again;
splx(s);
}
temodem(tp, msr)
register struct tty *tp;
{
    if ((tp->t_state & (ISOPEN|WOPEN)) == 0)

```

```
    return;
if (msr & DSR || te_modem[tp->t_index] == 0) {
    if ((tp->t_state&CARR_ON) == 0) {
        tp->t_state |= CARR_ON;
        if (tp->t_state&WOOPEN)
            wakeup((caddr_t)tp->t_rawq);
    }
} else {
    if (tp->t_state&CARR_ON) {
        tp->t_state &= ~CARR_ON;
        if (tp->t_rbuf.c_ptr != NULL) {
            ttyflush(tp, FREAD|FWRITE);
            signal(tp->t_pgrp, SIGHUP);
        }
    }
}
}
```



```

/*
 * Lisa INS8250A device driver
 *
 * Copyright 1984 UniSoft Corporation
 */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/types.h"
#include "sys/systm.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/file.h"
#include "sys/tty.h"
#include "sys/termio.h"
#include "sys/conf.h"
#include "sys/sysinfo.h"
#include "sys/var.h"
#include "setjmp.h"
#include "sys/reg.h"
#include "sys/mmu.h"
#include "sys/proc.h"

int teproc();

/*
 * structure to access Tecmar device registers
 */
struct tedevice {
    char fill1;
    char te_rbr; /* +1 data register */
#define te_dvlsb te_rbr /* 1sb of divisor latch */
    char fill2;
    char te_ier; /* +3 interrupt enable register */
#define te_dvmsb te_ier /* msb of divisor latch */
    char fill3;
    char te_iir; /* +5 interrupt id register */
    char fill4;
    char te_lcr; /* +7 line control register */
    char fill5;
    char te_mcr; /* +9 modem control register */
    char fill6;
    char te_lsr; /* +11 line status register */
    char fill7;
    char te_msr; /* +13 modem status register */
    char fill8;
    char te_scrat; /* +15 scratch register (8250-A only) */
    char fill[0x200-16]; /* sized to make tedevice be 0x200 long */
};

/*
 * structure to access the interrupt reset bit
 */
struct teldevice {
    struct tedevice fill[7]; /* filler */
    char skip;
    char te_intr; /* interrupt reset location */
};

/*
 * array used to remap ivec interrupt board slot number to tty slot
 */
int te_remap[3];

/*
 * Slot id to interrupt reset address
 */
struct teldevice *te_idevice[3] = {
    (struct tedevice *) (STDIO),
    (struct teldevice *) (STDIO+0x4000),
    (struct teldevice *) (STDIO+0x8000)
};

extern struct tty te_tty[1];
extern struct ttypr te_ttptr[1];

```

```

extern char te_dparam[1];
extern char te_modem[1];
extern int te_cnt;

#define MODEM 0x80 /* modem control on bit */
#define teudev ((d)&0x7f) /* from unix device number to device */

#define BAUD50 2304
#define BAUD75 1356
#define BAUD110 1047
#define BAUD134 857
#define BAUD150 768
#define BAUD300 384
#define BAUD600 192
#define BAUD1200 96
#define BAUD1800 64
#define BAUD2000 58
#define BAUD2400 48
#define BAUD3600 32
#define BAUD4800 24
#define BAUD7200 16
#define BAUD9600 12
#define BAUD19200 6
#define BAUD38400 3
#define BAUD56000 2

/* -1 means hangup, -2 means invalid */
int tebaudmap[] = {
    -1, BAUD50, BAUD75, BAUD110, BAUD134, BAUD150, BAUD134,
    BAUD300, BAUD600, BAUD1200, BAUD1800, BAUD2400,
    BAUD4800, BAUD9600, BAUD19200, BAUD38400
};

/* Interrupt enable register bits */
#define ERBFI 0x01 /* Enable received data available interrupt */
#define ETBEI 0x02 /* Enable transmitter enable holding register empty interrupt */
#define ELSI 0x04 /* Enable receiver line status interrupt */
#define EDSSI 0x08 /* Enable modem status interrupt */

/* Interrupt ident register bits */
#define IRQ 0x01 /* Interrupt request, 0 if interrupt pending */
#define THE 0x02 /* Transmitter holding register empty */
#define IID 0x06 /* Interrupt ID bit mask */
#define RID 0x04 /* Interrupt receive bit mask */

/* Line control register bits */
#define BITS5 0x00 /* 5 bits */
#define BITS6 0x01 /* 6 bits */
#define BITS7 0x02 /* 7 bits */
#define BITS8 0x03 /* 8 bits */
#define STOP1 0x00 /* One stop bit */
#define STOP2 0x04 /* Two stop bit */
#define PEN 0x08 /* Parity enable */
#define EPS 0x10 /* Even parity select */
#define SPS 0x20 /* Stick parity */
#define SBRK 0x40 /* Set break */
#define DLAB 0x80 /* Divisor latch access. i/o direction bit */

/* Modem control register bits */
#define DTR 0x01 /* Data terminal ready */
#define RTS 0x02 /* Request to send */

/* Line status register bits */
#define DATARDY 0x01 /* Data ready */
#define OV_ERR 0x02 /* Overrun error on receiver */
#define PE_ERR 0x04 /* Parity error */
#define FR_ERR 0x08 /* Framing error */
#define BR_INT 0x10 /* Break interrupt */
#define THRE 0x20 /* Transmitter holding register */
#define TEMT 0x40 /* Transmitter empty */

/* Modem status register bits */
#define DCTS 0x01 /* Delta clear to send */
#define DDSR 0x02 /* Delta data set ready */
#define TERE 0x04 /* Trailing edge ring indicator */
#define DDCD 0x08 /* Delta data carrier detect */

```

```

#define CTS    0x10    /* Clear to send */
#define DSR    0x20    /* Data set ready */
#define RI     0x40    /* Ring indicator */
#define DCD    0x80    /* Data carrier detect */

int teslotsused = 0;

/*
 * Initialize the baud rate
 * slot = 0, 1, or 2
 */
teinit(slot)
{
    register i, val;

    if (teslotsused+4 > te_cnt) {
        printf("\n\nSystem only configured for %d tecmar ports\n\n", te_cnt);
        return(1);
    }
    te_remap[slot] = teslotsused;
    val = STDIO + slot*0x4000 + 0x200;
    for (i = teslotsused; i < teslotsused+4; i++) {
        te_ttptr[i].tt_addr = val;
        te_ttptr[i].tt_tty = &te_tty[i];
        val += 0x200;
    }
    teslotsused += 4;
    return(0);
}

/* ARGSUSED */
teopen(dev, flag)
dev_t dev;
{
    register struct tedevice *addr;
    register struct tty *tp;
    register d;

#ifdef SINGLEUSER
    register struct proc *p;
#endif SINGLEUSER

    d = tedevice(dev);
    if (d >= te_cnt) {
        u.u_error = ENXIO;
        return;
    }
    tp = te_ttptr[d].tt_tty;
#ifdef SINGLEUSER
    p = u.u_procp;
    if ((p->p_pid == p->p_pgrp)
        && (u.u_ttyp == NULL)
        && (tp->t_pgrp == 0)) {
        u.u_error = ENOTTY;
        return;
    }
#endif SINGLEUSER
    addr = (struct tedevice *)te_ttptr[d].tt_addr;
    if (tp == 0 || addr == 0) {
        u.u_error = ENXIO;
        return;
    }
    tp->t_index = d;
    SPL5();
    if ((tp->t_state & (ISOPEN|WOPEN)) == 0) {
        tp->t_proc = tepoch;
        ttinit(tp);
        tp->t_iflag = ICRNL | ISTRIP;
        tp->t_oflag = OPOST | ONLCR | TAB3;
        tp->t_lflag = ISIG | ICANON | ECHO | ECHOK;
        tp->t_cflag = speed | CS8 | CREAD | HUPCL;
        teparam(dev);
    }
    te_modem[d] = dev & MODEM;
    if ((dev & MODEM) == 0 || addr->te_msr & DSR)
        tp->t_state |= CARR_ON;
    else

```

```

        tp->t_state &= ~CARR_ON;
    if (!(flag & FNDELAY))
        while ((tp->t_state & CARR_ON) == 0) {
            tp->t_state |= WOPEN;
            (void) sleep((caddr_t)&tp->t_rawq, TTOPRI);
        }
    SPL0();
    (*linesw[tp->t_line].l_open)(tp);
}

/* ARGSUSED */
teclose(dev, flag)
dev_t dev;
int flag;
{
    register struct tedevice *addr;
    register struct tty *tp;
    register d;

    d = tedevice(dev);
    tp = te_ttptr[d].tt_tty;
    (*linesw[tp->t_line].l_close)(tp);
    if (tp->t_cflag & HUPCL) {
        addr = (struct tedevice *)te_ttptr[d].tt_addr;
        d = 0;
        addr->te_mcr = d;
    }
}

teread(dev)
dev_t dev;
{
    register struct tty *tp;

    tp = te_ttptr[tedevice(dev)].tt_tty;
    (*linesw[tp->t_line].l_read)(tp);
}

tewrite(dev)
dev_t dev;
{
    register struct tty *tp;

    tp = te_ttptr[tedevice(dev)].tt_tty;
    (*linesw[tp->t_line].l_write)(tp);
}

teproc(tp, cmd)
register struct tty *tp;
{
    register struct ccblock *tbuf;
    register struct tedevice *addr;
    register dev_t dev;
    int s;
    extern ttrstrt();

    s = spltty();
    dev = tp->t_index;
    addr = (struct tedevice *)te_ttptr[dev].tt_addr;
    switch (cmd) {

    case T_TIME:
        tp->t_state &= ~TIMEOUT;
        goto start;

    case T_WFLUSH:
        tbuf = &tp->t_tbuf;
        tbuf->c_size -= tbuf->c_count;
        tbuf->c_count = 0;
        /* fall through */

    case T_RESUME:
        tp->t_state &= ~TTSTOP;
        goto start;

    case T_OUTPUT:
start:

```

```

if (tp->t_state & (TTSTOP|TIMEOUT|BUSY)) {
    /* if ((tp->t_state & BUSY) == 0) {
        addr = (struct tedevice *)((int)addr & 0xFCC000);
        ((struct teidevice *)addr)->te_intr = dev;
    } */
    break;
}
if (tp->t_state & TTXOFF) {
    tp->t_state &= ~TTXOFF;
    tp->t_state |= BUSY;
    addr->te_rbr = CSTOP;
    break;
}
if (tp->t_state & TTXON) {
    tp->t_state &= ~TTXON;
    tp->t_state |= BUSY;
    addr->te_rbr = CSTART;
    break;
}
tbuf = &tp->t_tbuf;
if ((tbuf->c_ptr == 0) || (tbuf->c_count == 0)) {
    if (tbuf->c_ptr)
        tbuf->c_ptr -= tbuf->c_size - tbuf->c_count;
    if (!(CPRES & (*linesw[tp->t_line].l_output)(tp))) {
        break;
    }
}
tp->t_state |= BUSY;
addr->te_ier = ERBFI | ETBEI | ELSI | EDSSI;
addr->te_rbr = *tbuf->c_ptr++;
tbuf->c_count--;
break;

case T_SUSPEND:
    tp->t_state |= TTSTOP;
    break;

case T_BLOCK:
    tp->t_state &= ~TTXON;
    tp->t_state |= TBLOCK;
    tp->t_state |= TTXOFF;
    goto start;

case T_RFLUSH:
    if (!(tp->t_state & TBLOCK))
        break;
    /* fall through */

case T_UNBLOCK:
    tp->t_state &= ~(TTXOFF|TBLOCK);
    tp->t_state |= TTXON;
    goto start;

case T_BREAK:
    tp->t_state |= TIMEOUT;
    timeout(ttrstrt, (caddr_t)tp, v.v_hz>>2);
    break;
}
splx(s);

}

teioctl(dev, cmd, arg, mode)
dev_t dev;
{
    if (ttiocm(te_ttptr[tedev(dev)].tt_tty, cmd, arg, mode))
        teparam(dev);
}

teparam(dev)
register dev_t dev;
{
    register struct tty *tp;
    register struct tedevice *addr;
    register int s, speed, oldpri;
    char c;

    tp = te_ttptr[tedev(dev)].tt_tty;
    addr = (struct tedevice *)te_ttptr[tedev(dev)].tt_addr;
    /* check for invalid speed */
    if ((speed = tebaudmap[tp->t_cflag & CBAUD]) == -2) {
        u.u_error = EINVAL;
        return;
    }
    s = 0;

    /*
     * hangup the line
     */
    if (speed == -1) {
        addr->te_mcr = s;
        return;
    }

    if (tp->t_state & BUSY) {
        te_dparam[tedev(dev)] = 1;
        return;
    }

    /*
     * set new speed
     */
    oldpri = spltty();
    addr->te_lcr = DLAB;
    addr->te_dvlsb = speed;
    addr->te_dvmsb = speed >> 8;
    addr->te_lcr = s;

    /*
     * set line control information
     */
    if ((tp->t_cflag & CSIZE) == CS8)
        s |= BITS8;
    else if ((tp->t_cflag & CSIZE) == CS7)
        s |= BITS7;
    else if ((tp->t_cflag & CSIZE) == CS6)
        s |= BITS6;
    else if ((tp->t_cflag & CSIZE) == CS5)
        s |= BITS5;
    if (tp->t_cflag & CSTOPB)
        s |= STOP2;
    if (tp->t_cflag & PARENB)
        if ((tp->t_cflag & PARODD) == 0)
            s |= PEN|EPS;
    addr->te_lcr = s;

    /*
     * set modem control information
     */
    addr->te_mcr = DTR | RTS;

    /*
     * enable interrupts
     */
    addr->te_ier = ERBFI | ETBEI | ELSI | EDSSI;

    /*
     * reset pending interrupts
     */
    c = addr->te_rbr;
    c = addr->te_lsr;
    c = addr->te_msr;
    c = addr->te_ier;

    splx(oldpri);
}

/* VARARGS */
teintr(ap)
struct args *ap;
{
    register struct tedevice *addr;

```

```

register struct ccblock *cbp;
register struct tty *tp;
register int c, lcnt, flg, iir, lsr;
register char ctmp;
int i, any;
struct teidevice *iaddr;
int index, s;
char lbuf[3];

s = spl5();
index = te_remap[ap->a_dev];
iaddr = te_idevice[ap->a_dev];
c = 0;
iaddr->te_intr = c; /* reset master interrupt */
again:
any = 0;
for (i = index; i < index+4; i++) {
    addr = (struct teidevice *)te_ttptr[i].tt_addr;
restart:
    iir = addr->te_iir;
    if (iir & IRQ)
        continue;
    tp = te_ttptr[i].tt_tty;
    lsr = addr->te_lsr;
    if (iir & RID) {
        sysinfo.rcvint++;
        c = addr->te_rbr & 0xFF;
        if (tp->t_rbuf.c_ptr == NULL) {
            any++;
            goto restart;
        }
        if ((lsr & DATARDY) == 0)
            c = 0;
        if (tp->t_iflag & IXON) {
            ctmp = c & 0177;
            if (tp->t_state & TTSTOP) {
                if (ctmp == CSTART || tp->t_iflag & IXANY)
                    (*tp->t_proc)(tp, T_RESUME);
            } else {
                if (ctmp == CSTOP)
                    (*tp->t_proc)(tp, T_SUSPEND);
            }
            if (ctmp == CSTART || ctmp == CSTOP) {
                any++;
                goto restart;
            }
        }
        /*
        * Check for errors
        */
        lcnt = 1;
        flg = tp->t_iflag;
        if (lsr & (PE_ERR|FR_ERR|OV_ERR|BR_INT)) {
            if ((lsr & PE_ERR) && (flg & INPCK))
                c |= PERROR;
            if (lsr & OV_ERR)
                c |= OVERRUN;
            if (lsr & FR_ERR)
                c |= FERROR;
            if (lsr & BR_INT)
                c = OVERRUN; /* reset char on a break */
        }
        if (c & (FERROR|PERROR|OVERRUN)) {
            if ((c & 0377) == 0) {
                if (flg & IGNBRK) {
                    any++;
                    goto restart;
                }
                if (flg & BRKINT) {
                    signal(tp->t_pgrp, SIGINT);
                    ttyflush(tp, (FREAD|FWRITE));
                    any++;
                    goto restart;
                }
            } else {
                if (flg & IGNPAR) {
                    any++;
                    goto restart;
                }
            }
        }
        if (flg & PARMRK) {
            lbuf[2] = 0377;
            lbuf[1] = 0;
            lcnt = 3;
            sysinfo.rawch += 2;
        } else
            c = 0;
    } else {
        if (flg & ISTRIP)
            c &= 0177;
        else {
            if (c == 0377 && flg & PARMRK) {
                lbuf[1] = 0377;
                lcnt = 2;
            }
        }
    }
    /*
    * Stash character in r_buf
    */
    cbp = &tp->t_rbuf;
    if (cbp->c_ptr == NULL) {
        any++;
        goto restart;
    }
    if (lcnt != 1) {
        lbuf[0] = c;
        while (lcnt) {
            *cbp->c_ptr++ = lbuf[--lcnt];
            if (--cbp->c_count == 0) {
                cbp->c_ptr -= cbp->c_size;
                (*linesw[tp->t_line].l_input)(tp);
            }
            if (cbp->c_size != cbp->c_count) {
                cbp->c_ptr -= cbp->c_size - cbp->c_count;
                (*linesw[tp->t_line].l_input)(tp);
            }
        }
    } else {
        *cbp->c_ptr = c;
        cbp->c_count--;
        (*linesw[tp->t_line].l_input)(tp);
    }
    any++;
    goto restart;
}
if (iir & THE) {
    sysinfo.xmtint++;
    tp->t_state &= -BUSY;
    if (te_dparam[i]) {
        te_dparam[i] = 0;
        tparam(i);
    }
    tproc(tp, T_OUTPUT);
} else {
    /*
    * must be a modem transition interrupt
    */
    temodem(tp, addr->te_msr);
}
any++;
goto restart;
}
if (any != 0)
    goto again;
splx(s);
}
temodem(tp, msr)
register struct tty *tp;
{
    if ((tp->t_state & (ISOPEN|WOPEN)) == 0)

```

```
        return;
    if (msr & DSR || te_modem[tp->t_index] == 0) {
        if ((tp->t_state & CARR_ON) == 0) {
            tp->t_state |= CARR_ON;
            if (tp->t_state & WOPEN)
                wakeup((caddr_t)tp->t_rawq);
        }
    } else {
        if (tp->t_state & CARR_ON) {
            tp->t_state &= ~CARR_ON;
            if (tp->t_rbuf.c_ptr != NULL) {
                ttyflush(tp, FREAD|FWRITE);
                signal(tp->t_pgrp, SIGHUP);
            }
        }
    }
}
```

```

/* !(!)text.c 1.3 */
#include "sys/param.h"
#include "sys/config.h"
#include "sys/mmu.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/system.h"
#include "sys/map.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/proc.h"
#include "sys/context.h"
#include "sys/text.h"
#include "sys/inode.h"
#include "sys/buf.h"
#include "sys/seq.h"
#include "sys/var.h"
#include "sys/sysinfo.h"
#include "sys/scat.h"

typedef int mem_t;

/*
 * Swap out process p.
 * The ff flag causes its core to be freed--
 * it may be off when called to create an image for a
 * child process in newproc.
 * On a partial swap ff is the negative number of blocks to be swapped.
 * Os is the old size of the process,
 * and is supplied during core expansion swaps.
 */
xswap(p, ff, os)
register struct proc *p;
{
    register a, i, s, tos;
    int addr, sz;

    if (os == 0)
        os = p->p_size;
    p->p_flag |= (SLOCK|SNOMMU);
    xccdec(p->p_textp);
    cxrelse(p->p_context);
    a = malloc(swapmap, ctod(p->p_size));
    if (a == NULL) {
        /*
         * s = decreasing click size of total disk space needed
         * tos = decreasing click size of process being swapped
         */
        tos = os;
        s = p->p_size;
    }
#ifdef NONSCATLOAD
    addr = p->p_addr;
#else
    addr = p->p_scatter;
#endif
    for (i=0; i < NSCATSWAP; i++) {
        if ((a = dtoc(mallocl(swapmap))) == NULL)
            break;
        if (a > s)
            a = s;
        p->p_xaddr[i] = malloc(swapmap, (mem_t)ctod(a));
        p->p_xsize[i] = a;
        sz = MIN(a, tos);
        if (sz) {
#ifdef NONSCATLOAD
            swap(p->p_xaddr[i], addr, (mem_t)sz, B_WRITE);
#else
            addr = swap(p->p_xaddr[i], addr, (mem_t)sz,
                B_WRITE);
#endif
        }
        s -= a;
        if (s == 0)
            break;
        tos -= a;
    }
}

```

```

        if (tos < 0)
            tos = 0;
#ifdef NONSCATLOAD
        addr += a;
#endif
    }
    if (s != 0)
        panic("out of swap space");
    a = p->p_xaddr[0]; /* for /bin/ps */
} else {
#ifdef NONSCATLOAD
    swap((daddr_t)a, (int)p->p_addr, (mem_t)os, B_WRITE);
#else
    (void) swap((daddr_t)a, (int)p->p_scatter, (mem_t)os, B_WRITE);
#endif
}
p->p_flag &= ~SNOMMU;
if (ff) {
#ifdef NONSCATLOAD
    mfree(coremap, (mem_t)os, (mem_t)p->p_addr);
#else
    memfree(p->p_scatter);
    p->p_scatter = 0;
#endif
}
cxrelse(p->p_context);
p->p_dkaddr = a;
p->p_flag &= ~(SLOAD|SLOCK);
#ifdef NONSCATLOAD
if (p->p_flag & SSWAPIT) {
    p->p_flag &= ~SSWAPIT;
    p->p_flag |= SCONTIG;
    wakeup((caddr_t)scatmap);
}
#endif
p->p_time = 0;
if (runout) {
    runout = 0;
    wakeup((caddr_t)&runout);
}
/*
 * relinquish use of the shared text segment
 * of a process.
 */
xfree()
{
    register struct text *xp;
    register struct inode *ip;
    register struct proc *p = u.u_procp;

    if ((xp = p->p_textp) == NULL)
        return;
    xlock(xp);
    xp->x_flag &= ~XLOCK;
    p->p_textp = NULL;
    u.u_ptsize = 0;
    ip = xp->x_iptr;
    if (--xp->x_count==0 && ((ip->i_mode&ISVTX)==0 || xp->x_flag&XERROR)) {
        xmsave(xp);
        xp->x_iptr = NULL;
        if (xp->x_daddr)
            mfree(swapmap, ctod(xp->x_size), (int)xp->x_daddr);
        cctxfree(xp);
        ip->i_flag &= ~ITEXT;
        if (ip->i_flag&ILOCK)
            ip->i_count--;
        else
            iput(ip);
    } else
        xccdec(xp);
    cxrelse(u.u_procp->p_context);
}
/*

```

```

* Attach to a shared text segment.
* If there is no shared text, just return.
* If there is, hook up to it:
* if it is not currently being used, it has to be read
* in from the inode (ip); the written bit is set to force it
* to be written out as appropriate.
* If it is being used, but is not currently in core,
* a swap has to be done to get it back.
*/
xalloc(ip)
register struct inode *ip;
{
    register struct text *xp;
    register ts;
    register struct text *xpl;
    register struct user *up;

    up = &u;
    if (up->u_exdata.ux_tsize == 0)
        return;
    xpl = NULL;
loop:
    for (xp = &text[0]; xp < (struct text *)v.ve_text; xp++) {
        if (xp->x_iptr == NULL) {
            if (xpl == NULL)
                xpl = xp;
            continue;
        }
        if (xp->x_iptr == ip) {
            xlock(xp);
            xp->x_ccount++;
            up->u_procp->p_textp = xp;
            if (xp->x_ccount == 0)
                (void) xexpand(xp);
            else
                xp->x_ccount++;
            xunlock(xp);
            return;
        }
    }
    if ((xp=xpl) == NULL) {
        printf("out of text\n");
        syserr.textovf++;
        if (xumount(NODEV))
            goto loop;
        psignal(up->u_procp, SIGKILL);
        return;
    }
    xp->x_flag = XLOAD|XLOCK;
    xp->x_count = 1;
    xp->x_ccount = 0;
    xp->x_iptr = ip;
    ip->i_flag |= ITEXT;
    ip->i_count++;
    ts = btoc(up->u_exdata.ux_tsize);
    xp->x_size = ts;
    /* if ((xp->x_daddr = malloc(swapmap, ctod(ts))) == NULL) */
    /* panic("out of swap space"); */
    xp->x_daddr = 0; /* defer swap alloc til later */
    up->u_procp->p_textp = xp;
    if (xexpand(xp)) {
        (void) estabur((unsigned)ts, (unsigned)0, (unsigned)0, 0, RW);
        xp->x_flag = XWRIT;
        return;
    }
    (void) estabur((unsigned)ts, (unsigned)0, (unsigned)0, 0, RW);
    up->u_count = up->u_exdata.ux_tsize;
    up->u_offset = sizeof(up->u_exdata);
    /* up->u_offset = up->u_exdata.ux_tstart; */
    up->u_base = (caddr_t)v.v_ustart;
    /* up->u_base = 0; */
    up->u_segflg = 2;
    up->u_procp->p_flag |= SLOCK;
    readi(ip);
    up->u_procp->p_flag &= ~SLOCK;
    up->u_segflg = 0;
}

```

```

    if (up->u_error || up->u_count!=0)
        xp->x_flag = XERROR;
    else
        xp->x_flag = XWRIT;
}

/*
 * Assure core for text segment
 * Text must be locked to keep someone else from
 * freeing it in the meantime.
 * x_ccount must be 0.
 */
xexpand(xp)
register struct text *xp;
{
    if (xmlink(xp)) {
        xp->x_ccount++;
        xunlock(xp);
        return(1);
    }
#ifdef NONSCATLOAD
    if ((xp->x_caddr = malloc(coremap, xp->x_size)) != NULL) {
        if ((xp->x_flag&XLOAD)==0)
            swap(xp->x_daddr, (int)xp->x_caddr, xp->x_size, B_READ);
        xp->x_ccount++;
        xunlock(xp);
        return(0);
    }
#endif
    if ((xp->x_scatter = memalloc(xp->x_size)) != NULL) {
        if ((xp->x_flag&XLOAD)==0)
            (void) swap(xp->x_daddr, (int)xp->x_scatter,
                xp->x_size, B_READ);
        xp->x_ccount++;
        xunlock(xp);
        return(0);
    }
#endif
    if (save(u.u_ssav)) {
        cxtxfree(xp);
        sureg();
        return(0);
    }
    xswap(u.u_procp, 1, 0);
    xunlock(xp);
    u.u_procp->p_flag |= SSWAP;
    qswtch();
#ifdef lint
    return(0);
#endif
}

/*
 * Lock and unlock a text segment from swapping
 */
xlock(xp)
register struct text *xp;
{
    while(xp->x_flag&XLOCK) {
        xp->x_flag |= XWANT;
        (void) sleep((caddr_t)xp, PSWP);
    }
    xp->x_flag |= XLOCK;
}

xunlock(xp)
register struct text *xp;
{
    if (xp->x_flag&XWANT)
        wakeup((caddr_t)xp);
    xp->x_flag &= ~(XLOCK|XWANT);
}

```

```

* Decrement the in-core usage count of a shared text segment.
* When it drops to zero, free the core space.
*/
xccdec(xp)
register struct text *xp;
{
    int prevlock;

    if (xp==NULL || xp->x_ccount==0)
        return;
    xlock(xp);
    if (!(prevlock = {u.u_procp->p_flag & SLOCK}))
        u.u_procp->p_flag |= SLOCK;
    if (--xp->x_ccount==0) {
        if (xp->x_flag&XWRIT) {
            xp->x_flag &= ~XWRIT;
            if (xp->x_daddr == 0)
                xp->x_daddr = swalloc(ctod(xp->x_size), 1);
#ifdef NONSCATLOAD
                swap(xp->x_daddr, (int)xp->x_caddr, xp->x_size, B_WRITE);
#else
                (void) swap(xp->x_daddr,
                    (int)xp->x_scat, xp->x_size, B_WRITE);
#endif
        }
        xmsave(xp);
        cctxfree(xp);
    }
    if (!prevlock)
        u.u_procp->p_flag &= ~SLOCK;
    xunlock(xp);
}

/*
* free the swap image of all unused saved-text text segments
* which are from device dev (used by umount system call).
*/
xumount(dev)
register dev_t dev;
{
    register struct inode *ip;
    register struct text *xp;
    register count = 0;

    for (xp = &text[0]; xp < (struct text *)v.ve_text; xp++) {
        if ((ip = xp->x_iptr) == NULL)
            continue;
        if (dev != NODEV && dev != ip->i_dev)
            continue;
        if (xuntext(xp))
            count++;
    }
    return(count);
}

/*
* remove a shared text segment from the text table, if possible.
*/
xrele(ip)
register struct inode *ip;
{
    register struct text *xp;

    if ((ip->i_flag&ITEXT) == 0)
        return;
    for (xp = &text[0]; xp < (struct text *)v.ve_text; xp++)
        if (ip==xp->x_iptr)
            (void) xuntext(xp);
}

/*
* remove text image from the text table.
* the use count must be zero.
*/
xuntext(xp)
register struct text *xp;

```

```

{
    register struct inode *ip;

    xlock(xp);
    if (xp->x_count) {
        xunlock(xp);
        return(0);
    }
    ip = xp->x_iptr;
    xfree(ip);
    xp->x_flag &= ~XLOCK;
    xp->x_iptr = NULL;
    cctxfree(xp);
    if (xp->x_daddr)
        mfree(swapmap, ctod(xp->x_size), (int)xp->x_daddr);
    ip->i_flag &= ~ITEXT;
    if (ip->i_flag&ILOCK)
        ip->i_count--;
    else
        iput(ip);
    return(1);
}

/*
* allocate swap blocks, freeing and sleeping as necessary
*/
swalloc(size, sflg)
{
    register addr;

    for (;;) {
        if (addr = malloc(swapmap, size))
            return(addr);
        if (swapclu()) {
            printf("\nWARNING: swap space running out\n");
            printf("  needed %d blocks\n", size);
            continue;
        }
        printf("\nDANGER: out of swap space\n");
        printf("  needed %d blocks\n", size);
        if (sflg) {
            mapwant(swapmap)++;
            (void) sleep((caddr_t)swapmap, PSWP);
        } else
            return(0);
    }
}

/*
* clean up swap used by text
*/
swapclu()
{
    register struct text *xp;
    register ans = 0;

    for (xp = text; xp < (struct text *)v.ve_text; xp++) {
        if (xp->x_iptr == NULL)
            continue;
        if (xp->x_flag&XLOCK)
            continue;
        if (xp->x_daddr == 0)
            continue;
        if (xp->x_count) {
            if (xp->x_ccount) {
                mfree(swapmap, ctod(xp->x_size),
                    (int)xp->x_daddr);
                xp->x_flag |= XWRIT;
                xp->x_daddr = 0;
                ans++;
            }
        } else {
            (void) xuntext(xp);
            ans++;
        }
    }
}

```



```

        return(ans);
    }
/*
 * free the saved text area associated with an inode
 */
xmfree(ip)
register struct inode *ip;
{
    register struct svtext *svx;

    for (svx = &svtext[0]; svx < (struct svtext *)v.ve_svtext; svx++) {
        if (svx->x_svflag&XSVBUSY && ip->i_number==svx->x_svnumber &&
            ip->i_dev==svx->x_svdev) {
            svx->x_svflag &= ~XSVBUSY;
#ifdef NONSCATLOAD
            mfree(coremap, svx->x_svsize, (mem_t)svx->x_svcaddr);
#else
            memfree((mem_t)svx->x_svscat);
#endif
            break;
        }
    }
/*
 * link up to a text region already in memory
 */
xmmlink(xp)
register struct text *xp;
{
    register struct svtext *svx;
    register struct inode *ip;

    ip = xp->x_iptr;
    for (svx = &svtext[0]; svx < (struct svtext *)v.ve_svtext; svx++) {
        if (svx->x_svflag&XSVBUSY && ip->i_number==svx->x_svnumber &&
            ip->i_dev==svx->x_svdev) {
            svx->x_svflag &= ~XSVBUSY;
#ifdef NONSCATLOAD
            xp->x_caddr = svx->x_svcaddr;
#else
            xp->x_scat = svx->x_svscat;
#endif
#ifdef TEXTTRACE
            printf("linking to text caddr 0x%x\n", svx->x_svcaddr);
#endif
            return(1);
        }
    }
    return(0);
}
/*
 * Release a shared text segment in the text area space.
 */
xmrelease()
{
    register struct svtext *svx, *tsvx;
    register n;

    n = ((unsigned) -1) >> 1;
    tsvx = NULL;
    for (svx = &svtext[0]; svx < (struct svtext *)v.ve_svtext; svx++) {
        if (svx->x_svflag&XSVBUSY && svx->x_svsize<n) {
            n = svx->x_svsize;
            tsvx = svx;
            continue;
        }
    }
    if (tsvx == NULL)
        return(0);
#ifdef TEXTTRACE
    printf("freeing %d segments at text caddr 0x%x\n",
        tsvx->x_svsize, tsvx->x_svcaddr);
#endif
}

```

```

#ifdef NONSCATLOAD
    mfree(coremap, tsvx->x_svsize, (mem_t)tsvx->x_svcaddr);
#else
    memfree((mem_t)tsvx->x_svscat);
#endif
    tsvx->x_svflag &= ~XSVBUSY;
    return(1);
}
/*
 * Save the memory of a text region of a shared process
 */
xmsave(xp)
register struct text *xp;
{
    register struct svtext *svx, *tsvx;
    register struct inode *ip;

    tsvx = NULL;
    ip = xp->x_iptr;
    for (svx = &svtext[0]; svx < (struct svtext *)v.ve_svtext; svx++) {
        if ((svx->x_svflag&XSVBUSY) == 0) {
            if (tsvx == NULL)
                tsvx = svx;
            continue;
        }
        if (ip->i_number==svx->x_svnumber && ip->i_dev==svx->x_svdev) {
            printf("xmrelease:memory saved more than once\n");
            tsvx = NULL;
            break;
        }
    }
/*
 * No space left in table
 */
    if (xp->x_flag&XERROR || tsvx == NULL) {
#ifdef NONSCATLOAD
        mfree(coremap, xp->x_size, (mem_t)xp->x_caddr);
#else
        memfree((mem_t)xp->x_scat);
#endif
    } else {
        tsvx->x_svflag |= XSVBUSY;
        tsvx->x_svsize = xp->x_size;
#ifdef NONSCATLOAD
        tsvx->x_svcaddr = xp->x_caddr;
#else
        tsvx->x_svscat = xp->x_scat;
#endif
        tsvx->x_svdev = ip->i_dev;
        tsvx->x_svnumber = ip->i_number;
#ifdef TEXTTRACE
        printf("saving %d segments at text caddr 0x%x\n",
            tsvx->x_svsize, tsvx->x_svcaddr);
#endif
    }
}

```

```

/*#define HOWFAR*/
/*#define SYSCALLS */

/* @(#)trap.c 1.2 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/proc.h"
#include "sys/reg.h"
#include "sys/psl.h"
#include "sys/trap.h"
#ifdef mc68881
#include "sys/fptrap.h"
#endif mc68881
#include "sys/seq.h"
#include "sys/sysinfo.h"
#ifdef VIRTUAL451
#include "sys/buserr.h"
#endif

#define EBIT 1 /* user error bit in PS: C-bit */
#define USER 0x1000 /* user-mode flag added to number */
#define NSYSENT 128

#ifdef SYSCALLS
static char reserved[] = "reserved";

char *callnames[] = {
/* 0 */ "indir", "exit", "fork", "read",
/* 4 */ "write", "open", "close", "wait",
/* 8 */ "creat", "link", "unlink", "exec",
/* 12 */ "chdir", "time", "mknod", "chmod",
/* 16 */ "chown", "break", "stat", "seek",
/* 20 */ "getpid", "mount", "umount", "setuid",
/* 24 */ "getuid", "stime", "ptrace", "alarm",
/* 28 */ "fstat", "pause", "utime", "stty",
/* 32 */ "gtty", "access", "nice", "sleep",
/* 36 */ "sync", "kill", "csw", "setpgrp",
/* 40 */ "tell", "dup", "pipe", "times",
/* 44 */ "profil", "lock", "setgid", "getgid",
/* 48 */ "sig", "msgsys", reserved, "acct",
/* 52 */ "shmsys", "semsys", "ioctl", "phys",
/* 56 */ "locking", "utssys", reserved, "exece",
/* 60 */ "umask", "chroot", "fcntl", "ulimit",

/* 64 */ "reboot", reserved, reserved, reserved,
#ifdef UCB_NET
/* 68 */ reserved, reserved, "select", "gethostname",
/* 72 */ "sethostname", "socket", "accept", "connect",
/* 76 */ "receive", "send", "socketaddr", "netreset",
#else
/* 68 */ reserved, reserved, reserved, reserved,
/* 72 */ reserved, reserved, reserved, reserved,
/* 76 */ reserved, reserved, reserved, reserved,
#endif
/* 80 */ reserved, reserved, reserved, reserved,
/* 84 */ reserved, reserved, reserved, reserved,
/* 88 */ reserved, reserved, reserved, reserved,
/* 92 */ reserved, reserved, reserved, reserved,
/* 96 */ reserved, reserved, reserved, reserved,
/* 100 */ reserved, reserved, reserved, reserved,
/* 104 */ reserved, reserved, reserved, reserved,
/* 108 */ reserved, reserved, reserved, reserved,
/* 112 */ reserved, reserved, reserved, reserved,
/* 116 */ reserved, reserved, reserved, reserved,
/* 120 */ reserved, reserved, reserved, reserved,
/* 124 */ reserved, reserved, reserved, reserved
};
#endif

/*
 * Offsets of the user's registers relative to
 */

/* the saved r0. See reg.h
 */
char regloc[8+8+1+1] = {
R0, R1, R2, R3, R4, R5, R6, R7,
AR0, AR1, AR2, AR3, AR4, AR5, AR6, SP, PC,
RPS
};

/*
 * Called from the trap handler when a processor trap occurs.
 */
trap(number, regs)
short number;
{
register struct user *up;
extern int parityno;
register i;
time_t syst;
int retval;
int *oldar0;

#ifdef mc68881 /* MC68881 floating-point coprocessor */
extern short fp881; /* is there an MC68881? */
#endif mc68881

up = &u;
retval = 0;
syst = up->u_stime;
#ifdef FLOAT /* sky floating point board */
up->u_fpsaved = 0;
#endif

oldar0 = up->u_ar0;
up->u_ar0 = &regs;
if (USERMODE(up->u_ar0[RPS]))
number |= USER;

#ifdef HOWFAR
if (number != RESCHED && number != RESCHED+USER) {
printf("trap number=0x%x ps=0x%x\n", number,
up->u_ar0[RPS]&0xFFFF);
showregs(1);
}
#endif

/* Handle parity specially to make it processor independent
 */
if (number==parityno || number==(parityno|USER)) {
if ((i = parityerror()) == 0) {
logparity((paddr_t)&up->u_ar0[PC]);
goto out;
}
if (i > 0) {
number = i | (number & USER);
goto sw;
}
if (number & USER) {
logparity((paddr_t)&up->u_ar0[PC]);
i = SIGBUS;
} else {
if (nofault) {
up->u_ar0 = oldar0;
longjmp(nofault, -1);
}
showbus();
panic("kernel parity error");
}
} else {
sw:
switch(number) {

/*
 * Trap not expected.
 * Usually a kernel mode bus error.
 */
default:
if ((number & USER) == 0) {
panicstr = "trap"; /* fake it for printf's */
printf("\ntrap type %d\n", number);
}
}
}
}

```

```

        showregs(1);
        panic("unexpected kernel trap");
    }

    case CHK + USER:      /* CHK instruction */
    case TRAPV + USER:   /* TRAPV instruction */
    case PRIVVIO + USER: /* Privileged violation */
    case L1010 + USER:   /* Line 1010 emulator */
    case L1111 + USER:   /* Line 1111 emulator */
    case TRAP4 + USER:
    case TRAP5 + USER:
    case TRAP6 + USER:
    case TRAP7 + USER:
    case TRAP8 + USER:
    case TRAP9 + USER:
    case TRAP10 + USER:
    case TRAP11 + USER:
    case TRAP12 + USER:
    case TRAP13 + USER:
    case TRAP14 + USER:
    case TRAP15 + USER:
    case ILLINST + USER: /* illegal instruction */
        i = SIGILL;
        break;

    case DIVZERO + USER: /* zero divide */
        i = SIGFPE;
        break;

#ifdef mc68881 /* MC68881 floating-point coprocessor */
    case FPBSUN + USER: /* Branch or Set on Unordered Condition */
    case FPINEX + USER: /* Inexact Result */
    case FPDZ + USER:   /* Floating Point Divide by Zero */
    case FPUNFL + USER: /* Underflow */
    case FPOPERR + USER: /* Operand Error */
    case FFOVFL + USER: /* Overflow */
    case FPSNAN + USER: /* Signalling NAN (Not-A-Number) */
        up->u_fpxc = number & 0xFF;
        i = SIGFPE;
        break;
#endif

    case TRCTRAP: /* trace out of kernel mode - */
        up->u_ar0 = oldar0;
        return(retval); /* this is happens when a trap instruction */
                        /* is executed with the trace bit set */

    case TRCTRAP + USER: /* trace */
    case TRAP1 + USER:   /* bpt - trap #1 */
        i = SIGTRAP;
        up->u_ar0[RPS] &= ~PS_T;
        break;

    case TRAP2 + USER: /* iot - trap #2 */
        i = SIGIOT;
        break;

    case TRAP3 + USER: /* emt - trap #3 */
        i = SIGEMT;
        break;

    case SYSCALL + USER: /* sys call - trap #0 */
        panic("syscall\n");

    /*
     * Allow process switch
     */
    case RESCHED + USER:
    case RESCHED:
        qswtch();
        goto out;

#ifdef VIRTUAL451
    /*
     * If the user SP is below the stack segment
     * and within STACKGAP clicks of the bottom
     * of the stack segment, then grow the
     * stack automatically.
     */
    case ADDRERR + USER: /* bus error - address error */
        i = SIGBUS;
        retval = 1;
        trapaddr((struct buserr *)&regs);
        break;

    case BUSERR + USER: /* memory management error - bus error */
        if (pagein((int)((struct buserr *)&regs)->ber_faddr)) {
            up->u_ar0 = oldar0;
            return(0);
        }
        i = SIGSEGV;
        retval = 1;
        trapaddr((struct buserr *)&regs);
        break;

    case ADDRERR: /* bus error - address error */
        trapaddr((struct buserr *)&regs);
        printf("kernel address error\n");
        showbus();
        panic("kernel memory management error");

    case BUSERR: /* memory management error - bus error in kernel */
        if (nofault) {
            up->u_ar0 = oldar0;
            longjmp(nofault, -1);
        }
        trapaddr((struct buserr *)&regs);
        berdump((struct buserr *)&regs);
        printf("kernel bus error\n");
        showbus();
        panic("kernel memory management error");
#else
    /*
     * If the user SP is below the stack segment,
     * grow the stack automatically.
     * This relies on the ability of the hardware
     * to restart a half executed instruction.
     * On the 68000 this is not the case and
     * the routine machdep/backup() will fail.
     */
    case ADDRERR + USER: /* bus error - address error */
        i = SIGBUS;
        retval = 1;
        trapaddr((struct buserr *)&regs);
        break;

    case BUSERR + USER: /* memory management error - bus error */
        if (i = backup(up->u_ar0))
            if (grow((unsigned)(up->u_ar0[SP]+1)))
                goto out;
        i = SIGSEGV;
        retval = 1;
        trapaddr((struct buserr *)&regs);
        break;

    case ADDRERR: /* kernel address error */
    case BUSERR: /* kernel bus error */
        if (nofault)
            longjmp(nofault, -1);
        trapaddr((struct buserr *)&regs);
        showbus();
        panic("kernel memory management error\n");
#endif

#endif

    /*
     * Unused trap vectors generate this trap type.
     * Receipt of this trap is a
     * symptom of hardware problems and may
     * represent a real interrupt that got
     * sent to the wrong place. Watch out

```

```

    /* for hangs on disk completion if this message appears.
    */
case SPURINT:
case SPURINT + USER:
    printf("\nRandom interrupt ignored\n");
    up->u_ar0 = oldar0;
    return(retval);
}
/* end else ... */
psignal(up->u_procp, i);

out:
if (up->u_procp->p_sig && issig())
    psig();
if (up->u_prof.pr_scale)
    addupc((unsigned)up->u_ar0[PC], &up->u_prof, (int)(up->u_stime-syst));
#ifdef FLOAT /* sky floating point board */
if (up->u_fpinuse && up->u_fpsaved)
    restfp();
#endif
#ifdef mc68881 /* MC68881 floating-point coprocessor */
if (fp881)
    fprest();
#endif mc68881
up->u_ar0 = oldar0;
return(retval);
}

/*
 * process a system call
 */
syscall(regs)
{
    register struct proc *pp;
    register struct user *up;
    register *regp, *argp;
    register i;
    int *oldar0;

#ifdef mc68881 /* MC68881 floating-point coprocessor */
extern short fp881; /* is there an MC68881? */
#endif mc68881

    up = &u;
    sysinfo.syscall++;
    up->u_error = 0;
#ifdef FLOAT /* sky floating point board */
up->u_fpsaved = 0;
#endif
    oldar0 = up->u_ar0;
    up->u_ar0 = regp = &regs;
    up->u_ap = argp = up->u_arg;
    i = regp[R0] & 0377;
    if (i >= NSYSENT)
        i = 0;
    argp[0] = regp[AR0];
    argp[1] = regp[R1];
    argp[2] = regp[AR1];
    argp[3] = regp[R2];
    argp[4] = regp[AR2];
    argp[5] = regp[R3];

#ifdef SYSCALLS
printf("***** %s: %x %x %x %x\n", callnames[i],
    argp[0], argp[1], argp[2], argp[3], argp[4], argp[5]);
#endif
    up->u_dirp = (caddr_t)argp[0];
    up->u_rval1 = 0;
    up->u_rval2 = regp[R1];
    if (qsave(up->u_qsav)) {
        /*
         * restore registers not saved by qsave
         */
        up = &u;
        regp = &regs;
        argp = up->u_arg;
        if (up->u_error==0)

```

```

        up->u_error = EINTR;
    } else {
        (*(sysent[i].sy_call))();
    }
    if (up->u_error) {
        regp[R0] = up->u_error;
        regp[RPS] |= PS_C; /* carry bit */
        if (++up->u_errcnt > 16) {
            up->u_errcnt = 0;
            runrun++;
        }
#ifdef SYSCALLS
printf(" syscall error = %d, pc = 0x%x\n",
    up->u_error, regp[PC]);
#endif
    } else {
#ifdef SYSCALLS
printf(" syscall success, pc = 0x%x\n",
    regp[PC]);
#endif
        regp[RPS] &= ~PS_C; /* carry bit */
        regp[R0] = up->u_rval1;
        regp[R1] = up->u_rval2;
    }
    pp = up->u_procp;
    /*
     * Test if the trap instruction was executed with the
     * trace bit set (the trace trap was already ignored)
     * and set the trace signal to avoid missing the trace
     * on the trap instruction.
     */
    pp->p_pri = (pp->p_cpu>>1) + (PUSER - NZERO) + pp->p_nice;
    curpri = pp->p_pri;
    if (pp->p_sig && issig())
        psig();
    up->u_ar0 = oldar0;
    if (runrun)
        qswtch();
#ifdef FLOAT /* sky floating point board */
if (up->u_fpinuse && up->u_fpsaved)
    restfp();
#endif
#ifdef mc68881 /* MC68881 floating-point coprocessor */
if (fp881)
    fprest();
#endif mc68881
}

/*
 * nonexistent system call-- signal bad system call.
 */
nosys()
{
    psignal(u.u_procp, SIGSYS);
}

/*
 * Ignored system call
 */
nullsys()
{
}

stray(addr)
physadr addr;
{
    logstray(addr);
    printf("stray interrupt at %x\n", addr);
}

/*
 * trapaddr - Save the info from a 68010 bus or address error.
 */
trapaddr(ap)
register struct buserr *ap;
{

```

```

extern int cputype;

if (cputype == 0)
    return;
u.u_fcode = ap->ber_sstat;
u.u_aaddr = ap->ber_faddr;
u.u_ireg = ap->ber_iib;
}

/*
 * showbus - print out status on mmgt error
 */
showbus()
{
    register struct user *up;

    up = &u;
    printf("vaddr = 0x%x paddr = 0x%x ireg = 0x%x fcode = 0x%x\n",
        up->u_aaddr, vtop((caddr_t)up->u_aaddr), up->u_ireg&0xFFFF,
        up->u_fcode&0xF);
    showregs(1);
}

/*
 * Show a processes registers
 */
showregs(mmuflg)
int mmuflg;
{
    register struct user *up;
    register int i, j;
    char command[DIRSIZ+1];

    up = &u;
#ifdef HOWFAR
    if (mmuflg)
        dumpmm(-1);
#endif
#ifdef lint
    dumpmm(mmuflg);
#endif
    for (i=0; i<DIRSIZ; i++) {
        j = up->u_comm[i];
        if (j<='7' || j>=0x7F)
            break;
        command[i] = j;
    }
    command[i] = 0;
    /*
     * separate prints in case up or u_procp trashed
     */
    printf("pc = 0x%x sr = 0x%x up->u_procp = 0x%x",
        up->u_ar0[PC], up->u_ar0[RPS]&0xFFFF, up->u_procp);
    printf(" pid = %d exec = '%s'\n", up->u_procp->p_pid, command);
    for (i = 0; i < 16; i++) {
        printf("0x%x ", up->u_ar0[i]);
        if (i == 7 || i == 15) printf("\n");
    }
}

#ifdef DUMPSTK
#include <sys/var.h>
#ifdef VIRTUAL451
#include <sys/mmu.h>
#endif
#include <sys/sysmacros.h>
/*
 * dump the present contents of the stack
 */
dumpstack(ret)
{
    register unsigned short *ip;

    ip = (unsigned short *)&ret;
    ip -= 4;
    printf("\n%x ", ip);
}

```

```

while (ip < (unsigned short *)((int)&ctob(v.v_usize))) {
    if (((int)ip & 31) == 0)
        printf("\n%x ", ip);
    printf(" %x", *ip++);
}
printf("\n");
if (ret != 0)
    panic("**** ABORTING ****");
}

/*
 * dump the present contents of the user stack
 */
dumpustack(max)
unsigned max;
{
    register unsigned short *ip, *jp;
    register unsigned i, n;

    ip = (unsigned short *) (u.u_ar0[SP] - 20);
    jp = ip + 64;
    if (jp < (unsigned short *)max)
        jp = (unsigned short *)max;
    if (jp > (unsigned short *)v.v_uend)
        jp = (unsigned short *)v.v_uend;
    printf("\n%x ", ip);
    while (ip < jp) {
        i = (fuword(ip++) >> 16) & 0xFFFF;
        if (((int)ip & 31) == 0)
            printf("\n%x ", ip);
        printf(" %x", i);
    }
    printf("\n");
}
#endif

```

```

/* @(#)tt0.c 1.4 */
/*
 * Line discipline 0
 * Includes Terminal Handling
 */

#include "sys/param.h"
#include "sys/types.h"
#include "sys/system.h"
#include "sys/conf.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/proc.h"
#include "sys/file.h"
#include "sys/tty.h"
#include "sys/termio.h"
#include "sys/crtctl.h"
#include "sys/sysinfo.h"
#include "sys/var.h"

extern char partab[];

char colsave, rowsave;        /* temp save for high queue restore */
struct clist tempq;            /* temp for echo during high queue */

/*
 * routine called on first teletype open.
 * establishes a process group for distribution
 * of quits and interrupts from the tty.
 */
ttopen(tp)
register struct tty *tp;
{
    register struct proc *pp;

    pp = u.u_procp;
    if ((pp->p_pid == pp->p_pgrp)
        && (u.u_ttyp == NULL)
        && (tp->t_pgrp == 0)) {
        u.u_ttyp = &tp->t_pgrp;
        tp->t_pgrp = pp->p_pgrp;
    }
    ttioctl(tp, LDOPEN, 0, 0);
    tp->t_state &= ~WOPEX;
    tp->t_state |= ISOPEN;
}

ttclose(tp)
register struct tty *tp;
{
    if ((tp->t_state&ISOPEN) == 0)
        return;
    tp->t_state &= ~ISOPEN;
    tp->t_pgrp = 0;
    ttioctl(tp, LDCLOSE, 0, 0);
}

/*
 * Called from device's read routine after it has
 * calculated the tty-structure given as argument.
 */
ttread(tp)
register struct tty *tp;
{
    register struct user *up;
    register struct clist *tq;

    up = &u;
    tq = &tp->t_cang;
    if (tq->c_cc == 0)
        canon(tp);
    while (up->u_count!=0 && up->u_error==0) {
        if (up->u_count >= CLSIZE) {
            register n;

```

```

register struct cblock *cp;

if ((cp = getcb(tq)) == NULL)
    break;
n = min(up->u_count,
        (unsigned) (cp->c_last - cp->c_first));
if (copyout((caddr_t) &cp->c_data[cp->c_first],
            (caddr_t) up->u_base, n))
    up->u_error = EFAULT;
putcf((struct cblock *) cp);
up->u_base += n;
up->u_count -= n;
} else {
    register c;

    if ((c = getc(tq)) < 0)
        break;
    if (subyte(up->u_base++, c))
        up->u_error = EFAULT;
    up->u_count--;
}
}
if (tp->t_state&TBLOCK) {
    if (tp->t_rawq.c_cc<TTXOLO) {
        (*tp->t_proc)(tp, T_UNBLOCK);
    }
}
}

/*
 * Called from device's write routine after it has
 * calculated the tty-structure given as argument.
 */
ttwrite(tp)
register struct tty *tp;
{
    register struct user *up;
    int hqflag;
    int col, row;

    up = &u;

qwait:
    spltty();
    while (tp->t_tmflag & QLOCKB) {
        tp->t_state |= OASLP;
        (void) sleep((caddr_t) &tp->t_outq, TTOPRI);
    }
    SPL0();
    if (!(tp->t_state&CARR_ON))
        return;
    hqflag = 0;
    while (up->u_count) {
        if (tp->t_outq.c_cc > tthiwt[tp->t_cflag&CBAUD]) {
            if (hqflag) {
                col = tp->t_col;
                row = tp->t_row;
                hqrelse(tp);
            }
            spltty();
            (*tp->t_proc)(tp, T_OUTPUT);
            while (tp->t_outq.c_cc > tthiwt[tp->t_cflag&CBAUD]) {
                tp->t_state |= OASLP;
                if (sleep((caddr_t) &tp->t_outq,
                        hqflag ? PZERO : (TTOPRI|PCATCH))) {
                    SPL0();
                    goto out;
                }
            }
        }
        SPL0();
        if (hqflag) {
            tp->t_tmflag |= QLOCKI;
            colsave = tp->t_col;
            rowsave = tp->t_row;
            ttyctl(LCA, tp, col, row);
            continue;
        }
    }
}

```

```

        if (tp->t_tmflag & QLOCKB)
            goto qwait;
    }
    if (up->u_count >= (CLSIZE/2) && tp->t_term == 0) {
        register n;
        register struct cblock *cp;

        if ((cp = getcf()) == NULL)
            break;
        n = min(up->u_count, (unsigned)cp->c_last);
        if (copyin((caddr_t)up->u_base, (caddr_t)cp->c_data,
            n)) {
            up->u_error = EFAULT;
            putcf((struct cblock *)cp);
            break;
        }
        up->u_base += n;
        up->u_count -= n;
        cp->c_last = n;
        ttxput(tp, cp, n);
    } else {
        register c;

        c = fubyte(up->u_base++);
        if (c < 0) {
            up->u_error = EFAULT;
            break;
        }
        up->u_count--;
        if (c == ESC && tp->t_term) {
            switch (c = cpass()) {
                int col;

            case -1:
                continue;
            case ESC:
                goto norm;
            case HIQ:
                if (hqflag++)
                    continue;
                tp->t_tmflag |= QLOCKB|QLOCKI;
                tp->t_hqcnt++;
                colsave = tp->t_col;
                rowsave = tp->t_row;
                continue;
            case LCA:
            case SVID:
            case DVID:
            case CVID:
                col = cpass();
            default:
                ttyctl(c, tp, col, c==LCA ? cpass() : 0);
            }
        } else {
            ttxput(tp, c, 0);
        }
    }
}
if (hqflag) {
    hqrelse(tp);
    (void) putc(QESC, &tp->t_outq);
    (void) putc(HQEND, &tp->t_outq);
    spltty();
    if (tp->t_state & OASLP) {
        tp->t_state &= -OASLP;
        wakeup((caddr_t)&tp->t_outq);
    }
    SPL0();
}
out:
tp->t_tmflag &= ~(QLOCKB|QLOCKI);
spltty();
(*tp->t_proc)(tp, T_OUTPUT);
SPL0();
}

```

```

/*
 * Place a character on raw TTY input queue, putting in delimiters
 * and waking up top half as needed.
 * Also echo if required.
 */
#define LCLESC 0400

ttn(tp)
register struct tty *tp;
{
    register c;
    register flg;
    register char *cp;
    ushort nchar, nc;

    nchar = tp->t_rbuf.c_size - tp->t_rbuf.c_count;
    /* reinit rx control block */
    tp->t_rbuf.c_count = tp->t_rbuf.c_size;
    if (nchar==0)
        return;
    flg = tp->t_iflag;
    /* KMC does all but IXOFF */
    if (tp->t_state&EXTPROC)
        flg &= IXOFF;
    nc = nchar;
    cp = tp->t_rbuf.c_ptr;
    if (nc < cfreelist.c_size || (flg & (INLCR|IGNCR|ICRNL|IUCLC))
        || tp->t_term) {
        /* must do per character processing */
        for (; nc--; cp++) {
            c = *cp;
            if (tp->t_term) {
                c &= 0177;
                if ((c = (*termw[tp->t_term].t_input)(c, tp))
                    == -1)
                    continue;
                if (c & CPRES) {
                    (void) putc(ESC, &tp->t_rawq);
                    (void) putc(c, &tp->t_rawq);
                    continue;
                }
            }
            if (c == '\n' && flg&INLCR)
                *cp = c = '\r';
            else if (c == '\r')
                if (flg&IGNCR)
                    continue;
                else if (flg&ICRNL)
                    *cp = c = '\n';
            if (flg&IUCLC && 'A' <= c && c <= 'Z')
                c += 'a' - 'A';
            if (putc(c, &tp->t_rawq))
                continue;
            sysinfo.rawch++;
        }
    }
    cp = tp->t_rbuf.c_ptr;
} else {
    /* may do block processing */
    putcb(CMATCH((struct cblock *)cp), &tp->t_rawq);
    sysinfo.rawch += nc;
    /* allocate new rx buffer */
    if ((tp->t_rbuf.c_ptr = getcf()->c_data)
        == ((struct cblock *)NULL)->c_data) {
        tp->t_rbuf.c_ptr = NULL;
        return;
    }
    tp->t_rbuf.c_count = cfreelist.c_size;
    tp->t_rbuf.c_size = cfreelist.c_size;
}
}

if (tp->t_rawq.c_cc > TTXOHI) {
    if (flg&IXOFF && !(tp->t_state&TBLOCK))
        (*tp->t_proc)(tp, T_BLOCK);
    if (tp->t_rawq.c_cc > TTYHOG) {

```

```

        ttyflush(tp, FREAD);
        return;
    }
}
flg = lobyte(tp->t_lflag);
if (tp->t_outq.c_cc > (tthiwat(tp->t_cflag&CBAUD) + TTECHI))
    flg &= ~(ECHO|ECHOK|ECHONL|ECHOE);
if (flg) while (nchar-->0) {
    c = *cp++;
    if (flg&ISIG) {
        if (c == tp->t_cc[VINTR]) {
            signal(tp->t_pgrp, SIGINT);
            if (!(flg&NOFLSH) && tp->t_hqcnt==0)
                ttyflush(tp, (FREAD|FWRITE));
            continue;
        }
        if (c == tp->t_cc[VQUIT]) {
            signal(tp->t_pgrp, SIGQUIT);
            if (!(flg&NOFLSH) && tp->t_hqcnt==0)
                ttyflush(tp, (FREAD|FWRITE));
            continue;
        }
    }
    if (flg&ICANON) {
        if (tp->t_state&CLESC) {
            flg |= LCLESC;
            tp->t_state &= ~CLESC;
        }
        if (c == '\n') {
            if (flg&ECHONL)
                flg |= ECHO;
            tp->t_delct++;
        } else if (c == '\\') {
            tp->t_state |= CLESC;
            if (flg&XCASE) {
                c |= QESC;
                if (flg&LCLESC)
                    tp->t_state &= ~CLESC;
            }
        } else if (c == tp->t_cc[VEOL] || c == tp->t_cc[VEOL2])
            tp->t_delct++;
        else if (!(flg&LCLESC)) {
            if (c == tp->t_cc[VERASE] && flg&ECHOE) {
                if (flg&ECHO)
                    ttxputi(tp, '\b');
                flg |= ECHO;
                ttxputi(tp, ' ');
                c = '\b';
            } else if (c == tp->t_cc[VKILL] && flg&ECHOK) {
                if (flg&ECHO)
                    ttxputi(tp, c);
                flg |= ECHO;
                c = '\n';
            } else if (c == tp->t_cc[VEOF]) {
                flg &= ~ECHO;
                tp->t_delct++;
            }
        }
    }
    if (flg&ECHO) {
        ttxputi(tp, c);
        (*tp->t_proc)(tp, T_OUTPUT);
    }
}
if (!(flg&ICANON)) {
    tp->t_state &= ~RTO;
    if (tp->t_rawq.c_cc >= tp->t_cc[VMIN])
        tp->t_delct = 1;
    else if (tp->t_cc[VTIME]) {
        if (!(tp->t_state&TACT))
            ttimeo(tp);
    }
}
if (tp->t_delct && (tp->t_state&IASLP)) {
    tp->t_state &= ~IASLP;
    wakeup((caddr_t)&tp->t_rawq);
}

```

```

    }
}
/*
 * Interrupt interface to ttxput.
 * Checks for High Queue write in progress, and saves characters to be echoed.
 */
ttxputi(tp, c)
register struct tty *tp;
{
    if (tp->t_tmflag & QLOCKI) {
        (void) putc(c, &tempq);
        return;
    } else
        ttxput(tp, c, 0);
}
/*
 * Put character(s) on TTY output queue, adding delays,
 * expanding tabs, and handling the CR/NL bit.
 * It is called both from the base level for output, and from
 * interrupt level for echoing.
 */
/* VARARGS1 */
ttxput(tp, ucp, ncode)
register struct tty *tp;
union {
    struct ch { /* machine dependent union */
        char dum[3];
        unsigned char theaddr;
    } ch;
    int thechar;
    struct cblock *ptr;
} ucp;
{
    register c;
    register flg;
    register unsigned char *cp;
    register char *colp;
    int ctype;
    int cs;
    struct cblock *scf;

    /* KMC does all but XCASE, virt term needs CR info for t_col */
    if (tp->t_state&EXTPROC) {
        if (tp->t_term || tp->t_lflag&XCASE)
            flg = tp->t_oflag&(OPOST|OLCUC|ONLRET|ONLCR);
        else
            flg = 0;
    } else
        flg = tp->t_oflag;
    if (ncode == 0) {
        ncode++;
        if (!(flg&OPOST)) {
            sysinfo.outch++;
            (void) putc(ucp.thechar, &tp->t_outq);
            return;
        }
        cp = (unsigned char *)&ucp.ch.theaddr;
        scf = NULL;
    } else {
        if (!(flg&OPOST)) {
            sysinfo.outch += ncode;
            putcb(ucp.ptr, &tp->t_outq);
            return;
        }
        cp = (unsigned char *)&ucp.ptr->c_data[ucp.ptr->c_first];
        scf = ucp.ptr;
    }
    while (ncode-->0) {
        c = *cp++;
        if (c >= 0200) {
            /* spl5-0 */
            if (c == QESC && !(tp->t_state&EXTPROC))
                (void) putc(QESC, &tp->t_outq);
            sysinfo.outch++;
        }
    }
}

```



```

        (void) putc(c, &tp->t_outq);
        continue;
    }
    /*
    * Generate escapes for upper-case-only terminals.
    */
    if (tp->t_lflag&XCASE) {
        colp = "{}|!~'\\\\";
        while(*colp++)
            if (c == *colp++) {
                ttxput(tp, '\\'|0200, 0);
                c = colp[-2];
                break;
            }
        if ('A' <= c && c <= 'Z')
            ttxput(tp, '\\'|0200, 0);
    }
    if (flg&OLCUC && 'a' <= c && c <= 'z')
        c += 'A' - 'a';
    cs = c;
    /*
    * Calculate delays.
    * The numbers here represent clock ticks
    * and are not necessarily optimal for all terminals.
    * The delays are indicated by characters above 0200.
    */
    ctype = partab[c];
    colp = &tp->t_col;
    c = 0;
    switch (ctype&077) {
    case 0: /* ordinary */
        (*colp)++;

    case 1: /* non-printing */
        break;

    case 2: /* backspace */
        if (flg&BSDLY)
            c = 2;
        if (*colp)
            (*colp)--;
        break;

    case 3: /* line feed */
        if (tp->t_term) {
            if (tp->t_vrow && tp->t_row >= tp->t_lrow) {
                ttyctl(UVSCN, tp);
                continue;
            }
            if (tp->t_tmflag & SNL) {
                ttyctl(NL, tp);
                continue;
            }
        }
        if (flg&ONLRET)
            goto cr;
        if (tp->t_row < tp->t_lrow)
            tp->t_row++;
        if (flg&ONLCR) {
            if (!((tp->t_state&EXTPROC) &&
                !(flg&ONOCR && *colp==0))) {
                sysinfo.outch++;
                (void) putc('\r', &tp->t_outq);
            }
            goto cr;
        }
    }
    nl:
        if (flg&NLDLY)
            c = 5;
        break;

    case 4: /* tab */
        c = 8 - ((*colp)&07);
        *colp += c;

```

top:

```

        if (!(tp->t_state&EXTPROC)) {
            ctype = flg&TABDLY;
            if (ctype == TAB0) {
                c = 0;
            } else if (ctype == TAB1) {
                if (c < 5)
                    c = 0;
            } else if (ctype == TAB2) {
                c = 2;
            } else if (ctype == TAB3) {
                sysinfo.outch += c;
                do
                    (void) putc(' ', &tp->t_outq);
                while (--c);
                continue;
            }
        }
        break;

    case 5: /* vertical tab */
        if (flg&VTDLY)
            c = 0177;
        break;

    case 6: /* carriage return */
        if (flg&OCRNL) {
            cs = '\n';
            goto nl;
        }
        if (flg&ONOCR && *colp == 0)
            continue;

    cr:
        if (!(tp->t_state&EXTPROC)) {
            ctype = flg&CRDLY;
            if (ctype == CR1) {
                if (*colp)
                    c = max((unsigned)((*colp>>4)+3), 6);
            } else if (ctype == CR2) {
                c = 5;
            } else if (ctype == CR3) {
                c = 9;
            }
        }
        *colp = 0;
        break;

    case 7: /* form feed */
        if (flg&FFDLY)
            c = 0177;
        break;
    }
    sysinfo.outch++;
    if (tp->t_term && *colp >= 80 && tp->t_row >= tp->t_lrow
        && tp->t_tmflag & LCF) {
        ttyctl(VHOME, tp);
        ttyctl(DL, tp);
        ttyctl(LCA, tp, 79, tp->t_lrow-1);
        (*colp)++;
    }
    if (tp->t_term==0)
        (void) putc(cs, &tp->t_outq);
    else
        qputc(cs, &tp->t_outq);
    if (!(tp->t_state&EXTPROC)) {
        if (c) {
            if ((c < 32) && flg&OFILL) {
                if (flg&OFDEL)
                    cs = 0177;
                else
                    cs = 0;
                (void) putc(cs, &tp->t_outq);
                if (c > 3)
                    (void) putc(cs, &tp->t_outq);
            } else {
                (void) putc(QESC, &tp->t_outq);
                (void) putc(c|0200, &tp->t_outq);
            }
        }
    }

```

```

    }
}
if (*colp >= 80 && tp->t_term && tp->t_tmflag&ANL)
    if (tp->t_tmflag&LCF)
        ttyctl(LCA, tp, 0, tp->t_row+1);
    else {
        if ((flg&ONLCR) == 0)
            ttxputi(tp, '\r');
        cs = '\n';
        goto top;
    }
}
if (scf != NULL)
    putcf(scf);
}

/*
 * Get next packet from output queue.
 * Called from xmit interrupt complete.
 */

ttout(tp)
register struct tty *tp;
{
    register struct cblock *tbuf;
    register c;
    register char *cptr;
    register retval;

    extern ttrstrt();

    if (tp->t_state&TTIOW && tp->t_outq.c_cc==0) {
        tp->t_state &= ~TTIOW;
        wakeup((caddr_t)&tp->t_oflag);
    }

delay:
    tbuf = &tp->t_tbuf;
    if (hibyte(tp->t_lflag)) {
        if (tbuf->c_ptr) {
            putcf(CMATCH((struct cblock *)tbuf->c_ptr));
            tbuf->c_ptr = NULL;
        }
        tp->t_state |= TIMEOUT;
        timeout(ttrstrt, (caddr_t)tp,
            (int)((hibyte(tp->t_lflag)&0177)+6));
        hibyte(tp->t_lflag) = 0;
        return(0);
    }
    retval = 0;
    if (((tp->t_state&EXTPROC) || ((tp->t_oflag&OPOST))) &&
        tp->t_term==0) {
        if (tbuf->c_ptr)
            putcf(CMATCH((struct cblock *)tbuf->c_ptr));
        if ((tbuf->c_ptr = (char *)getcb(&tp->t_outq)) == NULL)
            return(0);
        tbuf->c_count = ((struct cblock *)tbuf->c_ptr)->c_last -
            ((struct cblock *)tbuf->c_ptr)->c_first;
        tbuf->c_size = tbuf->c_count;
        tbuf->c_ptr = &((struct cblock *)tbuf->c_ptr)->c_data
            [((struct cblock *)tbuf->c_ptr)->c_first];
        retval = CPRES;
    } else {
        /* watch for timing */
        if (tbuf->c_ptr == NULL) {
            if ((tbuf->c_ptr = getcf()->c_data)
                == ((struct cblock *)NULL)->c_data) {
                tbuf->c_ptr = NULL;
                return(0); /* Add restart? */
            }
        }
        tbuf->c_count = 0;
        cptr = tbuf->c_ptr;
        while ((c=getc(&tp->t_outq)) >= 0) {
            if (c == QESC) {
                if ((c = getc(&tp->t_outq)) < 0)
                    break;
            }
        }
    }
}

```

```

        if (c == HQEND) {
            if (tp->t_term)
                tp->t_hqcnt--;
            continue;
        }
        if (c > 0200 && !(tp->t_state&EXTPROC)) {
            hibyte(tp->t_lflag) = c;
            if (!retval)
                goto delay;
            break;
        }
    }
    retval = CPRES;
    *cptr++ = c;
    tbuf->c_count++;
    if (tbuf->c_count >= cfreelist.c_size)
        break;
}
tbuf->c_size = tbuf->c_count;
}

if (tp->t_state&OASLP &&
    tp->t_outq.c_cc<=ttlowat[tp->t_cflag&CBAUD]) {
    tp->t_state &= ~OASLP;
    wakeup((caddr_t)&tp->t_outq);
}
return(retval);
}

tttimeo(tp)
register struct tty *tp;
{
    tp->t_state &= ~TACT;
    if (tp->t_lflag&ICANON || tp->t_cc[VTIME] == 0)
        return;
    if (tp->t_rawq.c_cc == 0 && tp->t_cc[VMIN])
        return;
    if (tp->t_state&RTO) {
        tp->t_delct = 1;
        if (tp->t_state&IASLP) {
            tp->t_state &= ~IASLP;
            wakeup((caddr_t)&tp->t_rawq);
        }
    } else {
        tp->t_state |= RTO|TACT;
        timeout(tttimeo, (caddr_t)tp,
            (int)(tp->t_cc[VTIME]*(short)((short)v.v_hz/10)));
    }
}

/*
 * I/O control interface
 */
/* ARGSUSED */
ttioctl(tp, cmd, arg, mode)
register struct tty *tp;
{
    ushort chg;
    struct termcb termblk;
    register struct termcb *tbp;

    switch(cmd) {
    case LDOPEN:
        if (tp->t_rbuf.c_ptr == NULL) {
            /* allocate RX buffer */
            while((tp->t_rbuf.c_ptr = getcf()->c_data)
                == ((struct cblock *)NULL)->c_data) {
                tp->t_rbuf.c_ptr = NULL;
                cfreelist.c_flag = 1;
                (void) sleep((caddr_t)&cfreelist, TTOPRI);
            }
            tp->t_rbuf.c_count = cfreelist.c_size;
            tp->t_rbuf.c_size = cfreelist.c_size;
            (*tp->t_proc)(tp, T_INPUT);
        }
        break;
    }
}

```

```

case LDCLOSE:
    spltty();
    (*tp->t_proc)(tp, T_RESUME);
    SPL0();
    ttywait(tp);
    ttyflush(tp, (FREAD|FWRITE));
    if (tp->t_tbuf.c_ptr) {
        putchar(CMATCH((struct cblock *)tp->t_tbuf.c_ptr));
        tp->t_tbuf.c_ptr = NULL;
        tp->t_tbuf.c_count = 0;
        tp->t_tbuf.c_size = 0;
    }
    if (tp->t_rbuf.c_ptr) {
        putchar(CMATCH((struct cblock *)tp->t_rbuf.c_ptr));
        tp->t_rbuf.c_ptr = NULL;
        tp->t_rbuf.c_count = 0;
        tp->t_rbuf.c_size = 0;
    }
    tp->t_tmflag = 0;
    break;

case LDCHG:
    chg = tp->t_lflag*arg;
    if (!(chg&ICANON))
        break;
    spltty();
    if (tp->t_canq.c_cc) {
        if (tp->t_rawq.c_cc) {
            tp->t_canq.c_cc += tp->t_rawq.c_cc;
            tp->t_canq.c_cl->c_next = tp->t_rawq.c_cf;
            tp->t_canq.c_cl = tp->t_rawq.c_cl;
        }
        tp->t_rawq = tp->t_canq;
        tp->t_canq = ttnulq;
    }
    tp->t_delct = tp->t_rawq.c_cc;
    SPL0();
    break;

case LDGETT:
    tbp = &termblk;
    tbp->st_flg = tp->t_tmflag;
    tbp->st_term = tp->t_term;
    tbp->st_crow = tp->t_row;
    tbp->st_ccol = tp->t_col;
    tbp->st_vrow = tp->t_vrow;
    tbp->st_lrow = tp->t_lrow;
    if (copyout((caddr_t)tbp, (caddr_t)arg, sizeof(termblk)))
        u.u_error = EFAULT;
    break;

case LDSETT:
    tbp = &termblk;
    if (copyin((caddr_t)arg, (caddr_t)tbp, sizeof(termblk))) {
        u.u_error = EFAULT;
        break;
    }
    if ((unsigned)tbp->st_term >= termcnt) {
        u.u_error = ENXIO;
        break;
    }
    if (tbp->st_term) {
        (*termstwb(tbp->st_term).t_ioctl)
            (tp,
             tp->t_term==tbp->st_term ? LDCHG : LDOPEN,
             tbp->st_vrow);
        if (u.u_error)
            break;
        tp->t_vrow = tbp->st_vrow;
        tp->t_term = tbp->st_term;
        if (tbp->st_flg&TM_SET)
            tp->t_tmflag = tbp->st_flg & ~TM_SET;
    } else {
        tp->t_term = 0;
    }

        tp->t_state &= ~CLESC;
        break;
    default:
        break;
    }
}

/***** ADDITIONS FOR TERMINAL HANDLERS *****/

/*
 * release the high priority queue for interrupts.
 * copy over any received characters while queue was locked.
 */
hqrset(tp)
register struct tty *tp;
{
    register c;

    ttyctl(LCA, tp, colsave, rowsave);
    spltty();
    while((c = getc(&tempq)) >= 0)
        ttxput(tp, c, 0);
    tp->t_tmflag &= ~QLOCKI;
    SPL0();
}

/*
 * put a character on the output queue,
 * checking first to see if it is a ESC.
 */
qputc(c, qp)
register c;
struct clist *qp;
{
    if (c == ESC)
        (void) putc(c, qp);
    (void) putc(c, qp);
}

#ifndef notdef
/* simulate Up Variable SCREEN as common routine */
ttuvscn(tp)
register struct tty *tp;
{
    ttyctl(VHOME, tp);
    ttyctl(DL, tp);
    ttyctl(LCA, tp, 0, tp->t_lrow);
}

/* simulate Down Variable SCREEN as common routine */
ttdvscn(tp)
register struct tty *tp;
{
    ttyctl(VHOME, tp);
    ttyctl(IL, tp);
}
#endif

char colpres, rowpres;

/* VARARGS */
ttyctl(ac, tp, acol, arow)
register struct tty *tp;
{
    register char *colp;
    register c;
    int sps;

    c = ac;
    colp = &tp->t_col;
    sps = spltty();

    colpres = *colp;
    rowpres = tp->t_row;
    switch(c) {

```

```

case CUP:
case DSCRL:
    if (tp->t_row == 0)
        goto out;
    tp->t_row--;
    break;
case CDN:
case USCLR:
    if (tp->t_row >= tp->t_lrow)
        goto out;
    tp->t_row++;
    break;
case UVSCN:
    *colp = 0;
    tp->t_row = tp->t_lrow;
    break;
case DVSCN:
    *colp = 0;
    tp->t_row = tp->t_vrow;
    break;
case CRI:
case STB:
case SPB:
    if (*colp >= 79)
        goto out;
    (*colp)++;
    break;
case CLE:
    if (*colp == 0)
        goto out;
    (*colp)--;
    break;
case HOME:
case CS:
case CM:
    tp->t_row = 0;
case DL:
case IL:
    *colp = 0;
    break;
case VHOME:
    *colp = 0;
    tp->t_row = tp->t_vrow;
    c = LCA;
    break;
case LCA:
    *colp = acol;
    tp->t_row = arow;
    break;
case ASEG:
    tp->t_row = (short) (tp->t_row+24)%(short) (tp->t_lrow+1);
    break;
case NL:
    if (tp->t_row < tp->t_lrow)
        tp->t_row++;
case CRTN:
    *colp = 0;
    break;
case SVID:
    tp->t_dstat != acol;
    c = DVID;
    break;
case CVID:
    tp->t_dstat &= ~acol;
    c = DVID;
    break;
case DVID:
    tp->t_dstat = acol;
    break;
}
(*termw[tp->t_term].t_output)(c, tp);
out:
    splx(sps);

```

```

/* @(#)ttl.c 1.4 */
/*
 * Line discipline 0
 * No Virtual Terminal Handling
 */

#include "sys/param.h"
#include "sys/types.h"
#include "sys/system.h"
#include "sys/conf.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/proc.h"
#include "sys/file.h"
#include "sys/tty.h"
#include "sys/termio.h"
#include "sys/sysinfo.h"
#include "sys/var.h"
#include "sys/req.h"

extern char partab[];

/*
 * routine called on first teletype open.
 * establishes a process group for distribution
 * of quits and interrupts from the tty.
 */
ttopen(tp)
register struct tty *tp;
{
    register struct proc *pp;

    pp = u.u_procp;
    if ((pp->p_pid == pp->p_pgrp)
        && (u.u_ttyp == NULL)
        && (tp->t_pgrp == 0)) {
        u.u_ttyp = &tp->t_pgrp;
        tp->t_pgrp = pp->p_pgrp;
    }
    ttioctl(tp, LDOOPEN, 0, 0);
    tp->t_state &= ~WOPEN;
    tp->t_state |= ISOOPEN;
}

ttclose(tp)
register struct tty *tp;
{
    if ((tp->t_state&ISOOPEN) == 0)
        return;
    tp->t_state &= ~ISOOPEN;
    tp->t_pgrp = 0;
    ttioctl(tp, LDCLOSE, 0, 0);
}

/*
 * Called from device's read routine after it has
 * calculated the tty-structure given as argument.
 */
ttread(tp)
register struct tty *tp;
{
    register struct user *up;
    register struct clist *tq;

    up = &u;
    tq = &tp->t_canq;
    if (tq->c_cc == 0)
        canon(tp);
    while (up->u_count != 0 && up->u_error == 0) {
        if (up->u_count >= CLSIZE) {
            register n;
            register struct cblock *cp;

            if ((cp = getcb(tq)) == NULL)

```

```

                break;
            n = MIN(up->u_count,
                (unsigned) (cp->c_last - cp->c_first));
            if (copyout((caddr_t) &cp->c_data[cp->c_first],
                (caddr_t) up->u_base, n))
                up->u_error = EFAULT;
            putcb((struct cblock *) cp);
            up->u_base += n;
            up->u_count -= n;
        } else {
            register c;

            if ((c = getc(tq)) < 0)
                break;
            if (subyte(up->u_base++, c))
                up->u_error = EFAULT;
            up->u_count--;
        }
    }
    if (tp->t_state&TBLOCK) {
        if (tp->t_rawq.c_cc < TTXOLO) {
            (*tp->t_proc)(tp, T_UNBLOCK);
        }
    }
}

/*
 * Called from device's write routine after it has
 * calculated the tty-structure given as argument.
 */
ttwrite(tp)
register struct tty *tp;
{
    register struct cblock *cp;
    register struct user *up;
    register a, n;

    if (!(tp->t_state&CARR_ON))
        return;
    up = &u;
    a = tthwat[tp->t_cflag&CBAUD];
    if (up->u_count <= 4 && up->u_ar0[R0] & 0x80000000) {
        n = up->u_arg[3];
        SPL6();
        while (tp->t_outq.c_cc > a) {
            (*tp->t_proc)(tp, T_OUTPUT);
            /*
             * For non-interrupting output devices sleep
             * only when characters are still pending.
             */
            if (tp->t_state&(TIMEOUT|TTSTOP|BUSY)) {
                tp->t_state |= OASLP;
                (void) sleep((caddr_t) &tp->t_outq, TTOPRI);
            }
        }
        SPL0();
        while (up->u_count) {
            ttoutput(tp, n&0xFF, 0);
            up->u_base++;
            up->u_count--;
            n >>= 8;
        }
    } else
        while (up->u_count) {
            SPL6();
            while (tp->t_outq.c_cc > a) {
                (*tp->t_proc)(tp, T_OUTPUT);
                /*
                 * For non-interrupting output devices sleep
                 * only when characters are still pending.
                 */
                if (tp->t_state&(TIMEOUT|TTSTOP|BUSY)) {
                    tp->t_state |= OASLP;
                    (void) sleep((caddr_t) &tp->t_outq, TTOPRI);
                }
            }
        }
}

```

```

SPL0();
if (up->u_count >= (CLSIZE/4)) {
    if ((cp = getcf()) == NULL)
        break;
    n = MIN(up->u_count, (unsigned)cp->c_last);
    if (copyin((caddr_t)up->u_base, (caddr_t)cp->c_data,
              n)) {
        up->u_error = EFAULT;
        putcf((struct cblock *)cp);
        break;
    }
    /*
     * Put trailing '\n' in a separate cblock
     */
    if (n==up->u_count && cp->c_data[n-1]=='\n' && n>=3)
        n--;
    up->u_base += n;
    up->u_count -= n;
    cp->c_last = n;
    ttxput(tp, cp, n);
} else {
    n = fubyte(up->u_base++);
    if (n<0) {
        up->u_error = EFAULT;
        break;
    }
    up->u_count--;
    ttxput(tp, n, 0);
}
}
spltty();
if (!(tp->t_state&BUSY))
    (*tp->t_proc)(tp, T_OUTPUT);
SPL0();
}

/*
 * Place a character on raw TTY input queue, putting in delimiters
 * and waking up top half as needed.
 * Also echo if required.
 */
#define LCLESC 0400

ttin(tp)
register struct tty *tp;
{
    register c;
    register flg;
    register char *cp;
    ushort nchar, nc;

    nchar = tp->t_rbuf.c_size - tp->t_rbuf.c_count;
    /* reinit rx control block */
    tp->t_rbuf.c_count = tp->t_rbuf.c_size;
    if (nchar==0)
        return;
    flg = tp->t_iflag;
    nc = nchar;
    cp = tp->t_rbuf.c_ptr;
    if (nc < cfreelist.c_size || (flg & (INLCR|IGNCR|ICRNL|IUCLC))) {
        /* must do per character processing */
        for (; nc--; cp++) {
            c = *cp;
            if (c == '\n' && flg&INLCR)
                *cp = c = '\r';
            else if (c == '\r')
                if (flg&IGNCR)
                    continue;
                else if (flg&ICRNL)
                    *cp = c = '\n';
            if (flg&IUCLC && 'A' <= c && c <= 'Z')
                c += 'a' - 'A';
            if (putc(c, &tp->t_rawq))
                continue;
            sysinfo.rawch++;
        }
    }

    cp = tp->t_rbuf.c_ptr;
} else {
    /* may do block processing */
    putcb(CMATCH((struct cblock *)cp), &tp->t_rawq);
    sysinfo.rawch += nc;
    /* allocate new rx buffer */
    if ((tp->t_rbuf.c_ptr = getcf()->c_data)
        == ((struct cblock *)NULL)->c_data) {
        tp->t_rbuf.c_ptr = NULL;
        return;
    }
    tp->t_rbuf.c_count = cfreelist.c_size;
    tp->t_rbuf.c_size = cfreelist.c_size;
}

if (tp->t_rawq.c_cc > TTXOHI) {
    if (flg&IXOFF && !(tp->t_state&TBLOCK))
        (*tp->t_proc)(tp, T_BLOCK);
    if (tp->t_rawq.c_cc > TTYHOG) {
        ttyflush(tp, FREAD);
        return;
    }
}
flg = lobyte(tp->t_lflag);
if (tp->t_outq.c_cc > (tthiwat[tp->t_cflag&CBAUD] + TTECHI))
    flg &= ~(ECHO|ECHOK|ECHONL|ECHOE);
if (flg) while (nchar-->0) {
    c = *cp++;
    if (flg&ISIG) {
        if (c == tp->t_cc[VINTR]) {
            signal(tp->t_pgrp, SIGINT);
            if (!(flg&NOFLSH))
                ttyflush(tp, (FREAD|FWRITE));
            continue;
        }
        if (c == tp->t_cc[VQUIT]) {
            signal(tp->t_pgrp, SIGQUIT);
            if (!(flg&NOFLSH))
                ttyflush(tp, (FREAD|FWRITE));
            continue;
        }
    }
}
}

#ifdef notdef
if (flg&ICANON) {
    if (tp->t_state&CLESC) {
        flg |= LCLESC;
        tp->t_state &= ~CLESC;
    }
    if (c == '\n') {
        if (flg&ECHONL)
            flg |= ECHO;
        tp->t_delct++;
    } else if (c == '\\') {
        tp->t_state |= CLESC;
        if (flg&XCASE) {
            c |= QESC;
            if (flg&LCLESC)
                tp->t_state &= ~CLESC;
        }
    }
} else if (c == tp->t_cc[VEOL] || c == tp->t_cc[VEOL2])
    tp->t_delct++;
else if (!(flg&LCLESC)) {
    if (c == tp->t_cc[VERASE] && flg&ECHOE) {
        if (flg&ECHO)
            ttxput(tp, '\b', 0);
        flg |= ECHO;
        ttxput(tp, ' ', 0);
        c = '\b';
    } else if (c == tp->t_cc[VKILL] && flg&ECHOK) {
        if (flg&ECHO)
            ttxput(tp, c, 0);
        flg |= ECHO;
        c = '\n';
    } else if (c == tp->t_cc[VEOF]) {
        flg &= ~ECHO;
    }
}
}
#endif

```

```

        tp->t_delct++;
    }
}
if (flg&ECHO) {
    ttxput(tp, c, 0);
    (*tp->t_proc)(tp, T_OUTPUT);
}
#else
if (flg&ICANON) {
    if (c == '\n') {
        if (flg&ECHONL)
            flg |= ECHO;
        tp->t_delct++;
    } else if (c == tp->t_cc[VEOL] || c == tp->t_cc[VEOL2])
        tp->t_delct++;
    if (!(tp->t_state&CLESC)) {
        if (c == '\\')
            tp->t_state |= CLESC;
        if (c == tp->t_cc[VERASE] && flg&ECHOE) {
            if (flg&ECHO)
                ttxput(tp, '\b', 0);
            flg |= ECHO;
            ttxput(tp, ' ', 0);
            c = '\b';
        } else if (c == tp->t_cc[VKILL] && flg&ECHO) {
            if (flg&ECHO)
                ttxput(tp, c, 0);
            flg |= ECHO;
            c = '\n';
        } else if (c == tp->t_cc[VEOF]) {
            flg &= ~ECHO;
            tp->t_delct++;
        }
    } else {
        if (c != '\\') {
            if (flg&XCASE)
                tp->t_state &= ~CLESC;
        }
    }
}
if (flg&ECHO) {
    ttxput(tp, c, 0);
    (*tp->t_proc)(tp, T_OUTPUT);
}
#endif
}
if (!(flg&ICANON)) {
    tp->t_state &= ~RTO;
    if (tp->t_rawq.c_cc >= tp->t_cc[VMIN])
        tp->t_delct = 1;
    else if (tp->t_cc[VTIME]) {
        if (!(tp->t_state&TACT))
            ttimeo(tp);
    }
}
if (tp->t_delct && (tp->t_state&IASLP)) {
    tp->t_state &= ~IASLP;
    wakeup((caddr_t)&tp->t_rawq);
}
}
/*
 * Scan a list of characters and assure that they require no
 * post processing
 */
ttxchk(ncode, cp)
register short ncode;
register unsigned char *cp;
{
    register c, n;

    n = 0;
    ncode--;
    do {
        c = *cp++;
        if (c & 0200)
            return(-1);
    }
}
        c = partab[c] & 077;
        if (c == 0)
            n++;
        else if (c != 1)
            return(-1);
    } while (--ncode != -1);
    return(n);
}
/*
 * Put character(s) on TTY output queue, adding delays,
 * expanding tabs, and handling the CR/NL bit.
 * It is called both from the base level for output, and from
 * interrupt level for echoing.
 */
/* VARARGS1 */
ttxput(tp, ucp, ncode)
register struct tty *tp;
register ncode;
union {
    struct ch {
        /* machine dependent union */
        char dum[3];
        unsigned char theaddr;
    } ch;
    int thechar;
    struct cblock *ptr;
} ucp;
{
    register struct clist *outqp;
    register unsigned char *cp;
    register char *colp;
    struct cblock *scf;
    int cs;

    flg = tp->t_oflag;
    outqp = &tp->t_outq;
    if (ncode == 0) {
        if (!(flg&OPOST)) {
            sysinfo.outch++;
            (void) putc(ucp.thechar, outqp);
            return;
        }
        ncode++;
        cp = (unsigned char *)&ucp.ch.theaddr;
        scf = NULL;
    } else {
        if (!(flg&OPOST)) {
            sysinfo.outch += ncode;
            putcb(ucp.ptr, outqp);
            return;
        }
        cp = (unsigned char *)&ucp.ptr->c_data[ucp.ptr->c_first];
        scf = ucp.ptr;
    }
    if ((tp->t_lflag&XCASE)==0 && (flg&OLCUC)==0) {
        colp = &tp->t_col;
        if (ncode > 1 && (c = ttxchk(ncode, cp)) >= 0) {
            (*colp) += c;
            sysinfo.outch += ncode;
            putcb(ucp.ptr, outqp);
            return;
        }
    }
    while (ncode-- > 0) {
        ctype = partab[c = *cp++] & 077;
        if (ctype==0) {
            (*colp)++;
            sysinfo.outch++;
            (void) putc(c, outqp);
            continue;
        }
        else if (ctype==1) {
            sysinfo.outch++;
            (void) putc(c, outqp);
            continue;
        }
    }
}

```

```

if (c >= 0200) {
    if (c == QESC)
        (void) putc(QESC, outqp);
    sysinfo.outch++;
    (void) putc(c, outqp);
    continue;
}
cs = c;
/*
 * Calculate delays.
 * The numbers here represent clock ticks
 * and are not necessarily optimal for all terminals.
 * The delays are indicated by characters above 0200.
 */
c = 0;
switch (ctype) {

case 0: /* ordinary */
    (*colp)++;

case 1: /* non-printing */
    break;

case 2: /* backspace */
    if (flg&BSDLY)
        c = 2;
    if (*colp)
        (*colp)--;
    break;

case 3: /* line feed */
    if (flg&ONLRET)
        goto qcr;
    if (flg&ONLCR) {
        if (!(flg&ONOCR && *colp==0)) {
            sysinfo.outch++;
            (void) putc('\r', outqp);
        }
        goto qcr;
    }

qnl:
    if (flg&NLDLY)
        c = 5;
    break;

case 4: /* tab */
    c = 8 - ((*colp)&07);
    *colp += c;
    ctype = flg&TABDLY;
    if (ctype == TAB0) {
        c = 0;
    } else if (ctype == TAB1) {
        if (c < 5)
            c = 0;
    } else if (ctype == TAB2) {
        c = 2;
    } else if (ctype == TAB3) {
        sysinfo.outch += c;
        do
            (void) putc(' ', outqp);
        while (--c);
        continue;
    }
    break;

case 5: /* vertical tab */
    if (flg&VTDLY)
        c = 0177;
    break;

case 6: /* carriage return */
    if (flg&OCRNL) {
        cs = '\n';
        goto qnl;
    }
    if (flg&ONOCR && *colp == 0)

```

```

        continue;
qcr:
    ctype = flg&CRDLY;
    if (ctype == CR1) {
        if (*colp)
            c = max((unsigned)((*colp>>4) + 3), 6);
    } else if (ctype == CR2) {
        c = 5;
    } else if (ctype == CR3) {
        c = 9;
    }
    *colp = 0;
    break;

case 7: /* form feed */
    if (flg&FFDLY)
        c = 0177;
    break;
}
sysinfo.outch++;
(void) putc(cs, outqp);
if (c) {
    if ((c < 32) && flg&OFILL) {
        if (flg&OFDEL)
            cs = 0177;
        else
            cs = 0;
        (void) putc(cs, outqp);
        if (c > 3)
            (void) putc(cs, outqp);
    } else {
        (void) putc(QESC, outqp);
        (void) putc(c|0200, outqp);
    }
}
} else
while (ncode--) {
    c = *cpt++;
    if (c >= 0200) {
/* spl5-0 */
        if (c == QESC)
            (void) putc(QESC, outqp);
        sysinfo.outch++;
        (void) putc(c, outqp);
        continue;
    }
/*
 * Generate escapes for upper-case-only terminals.
 */
if (tp->t_lflag&XCASE) {
    colp = "{}|!|^~'\\";
    while(*colp++)
        if (c == *colp++) {
            ttxput(tp, '\\'|0200, 0);
            c = colp[-2];
            break;
        }
    if ('A' <= c && c <= 'Z')
        ttxput(tp, '\\'|0200, 0);
}
if (flg&OLCUC && 'a' <= c && c <= 'z')
    c += 'A' - 'a';
cs = c;
/*
 * Calculate delays.
 * The numbers here represent clock ticks
 * and are not necessarily optimal for all terminals.
 * The delays are indicated by characters above 0200.
 */
ctype = partab[c];
colp = &tp->t_col;
c = 0;
switch (ctype&077) {

case 0: /* ordinary */

```



```

        (*colp)++;
case 1: /* non-printing */
        break;
case 2: /* backspace */
        if (flg&BSDLY)
            c = 2;
        if (*colp)
            (*colp)--;
        break;
case 3: /* line feed */
        if (flg&ONLRET)
            goto cr;
        if (flg&ONLCR) {
            if (!(flg&ONOCR && *colp==0)) {
                sysinfo.outch++;
                (void) putc('\r', outqp);
            }
            goto cr;
        }
nl:
        if (flg&NLDLY)
            c = 5;
        break;
case 4: /* tab */
        c = 8 - ((*colp)&07);
        *colp += c;
        ctype = flg&TABDLY;
        if (ctype == TAB0) {
            c = 0;
        } else if (ctype == TAB1) {
            if (c < 5)
                c = 0;
        } else if (ctype == TAB2) {
            c = 2;
        } else if (ctype == TAB3) {
            sysinfo.outch += c;
            do
                (void) putc(' ', outqp);
            while (--c);
            continue;
        }
        break;
case 5: /* vertical tab */
        if (flg&VTDLY)
            c = 0177;
        break;
case 6: /* carriage return */
        if (flg&OCRNL) {
            cs = '\n';
            goto nl;
        }
        if (flg&ONOCR && *colp == 0)
            continue;
cr:
        ctype = flg&CRDLY;
        if (ctype == CR1) {
            if (*colp)
                c = max((unsigned)((*colp)>>4) + 3), 6);
        } else if (ctype == CR2) {
            c = 5;
        } else if (ctype == CR3) {
            c = 9;
        }
        *colp = 0;
        break;
case 7: /* form feed */
        if (flg&FFDLY)
            c = 0177;
        break;

```

```

        }
        sysinfo.outch++;
        (void) putc(cs, outqp);
        if (c) {
            if ((c < 32) && flg&OFILL) {
                if (flg&OFDEL)
                    cs = 0177;
                else
                    cs = 0;
                (void) putc(cs, outqp);
                if (c > 3)
                    (void) putc(cs, outqp);
            } else {
                (void) putc(QESC, outqp);
                (void) putc(c|0200, outqp);
            }
        }
    }
    if (scf != NULL)
        putcf(scf);
}
/*
 * Get next packet from output queue.
 * Called from xmit interrupt complete.
 */
ttout(tp)
register struct tty *tp;
{
    register struct cblock *tbuf;
    register c;
    register char *cptr;
    register retval;
    register struct clist *outqp;
    extern ttrstrt();

    outqp = &tp->t_outq;
    if (tp->t_state&TTIOW && outqp->c_cc==0) {
        tp->t_state &= ~TTIOW;
        wakeup((caddr_t)&tp->t_oflag);
    }
delay:
    tbuf = &tp->t_tbuf;
    if (hibyte(tp->t_lflag)) {
        if (tbuf->c_ptr) {
            putcf(CMATCH((struct cblock *)tbuf->c_ptr));
            tbuf->c_ptr = NULL;
        }
        tp->t_state |= TIMEOUT;
        timeout(ttrstrt, (caddr_t)tp,
            (int){(hibyte(tp->t_lflag)&0177)+6});
        hibyte(tp->t_lflag) = 0;
        return(0);
    }
    retval = 0;
    if (!(tp->t_oflag&OPOST)) {
        if (tbuf->c_ptr)
            putcf(CMATCH((struct cblock *)tbuf->c_ptr));
        if ((tbuf->c_ptr = (char *)getc(outqp)) == NULL)
            goto out;
        /* return(0); */
        tbuf->c_count = ((struct cblock *)tbuf->c_ptr)->c_last -
            ((struct cblock *)tbuf->c_ptr)->c_first;
        tbuf->c_size = tbuf->c_count;
        tbuf->c_ptr = &((struct cblock *)tbuf->c_ptr)->c_data
            [((struct cblock *)tbuf->c_ptr)->c_first];
        retval = CPRES;
    } else {
        /* watch for timing */
        if (tbuf->c_ptr == NULL) {
            if ((tbuf->c_ptr = getcf()->c_data)
                == ((struct cblock *)NULL)->c_data) {
                tbuf->c_ptr = NULL;
                goto out;
            }
            /* return(0); /* Add restart? */

```

```

    }
}
tbuf->c_count = 0;
cptr = tbuf->c_ptr;
while ((c = getc(outqp)) >= 0) {
    if (c == QESC) {
        if ((c = getc(outqp)) < 0)
            break;
        if (c > 0200) {
            hbyte(tp->t_lflag) = c;
            if (!retval)
                goto delay;
            break;
        }
    }
    retval = CPRES;
    *cptr++ = c;
    tbuf->c_count++;
    if (tbuf->c_count >= cfreelist.c_size)
        break;
}
tbuf->c_size = tbuf->c_count;
}
out:
if (tp->t_state & OASLP) {
    outqp->c_cc <- tlowat[tp->t_cflag & CBAUD];
    tp->t_state &= ~OASLP;
    wakeup((caddr_t)outqp);
}
return(retval);
}

tttimeo(tp)
register struct tty *tp;
{
    tp->t_state &= ~TACT;
    if (tp->t_lflag & ICANON || tp->t_cc[VTIME] == 0)
        return;
    if (tp->t_rawq.c_cc == 0 && tp->t_cc[VMIN])
        return;
    if (tp->t_state & RTO) {
        tp->t_delct = 1;
        if (tp->t_state & IASLP) {
            tp->t_state &= ~IASLP;
            wakeup((caddr_t)&tp->t_rawq);
        }
    } else {
        tp->t_state |= RTO | TACT;
        timeout(tttimeo, (caddr_t)tp,
            (int)(tp->t_cc[VTIME] * (short)((short)v_v_hz/10)));
    }
}

/*
 * I/O control interface
 */
/* ARGSUSED */
ttioctl(tp, cmd, arg, mode)
register struct tty *tp;
{
    ushort chg;

    switch(cmd) {
    case LDOPEN:
        if (tp->t_rbuf.c_ptr == NULL) {
            /* allocate RX buffer */
            while((tp->t_rbuf.c_ptr = getcf()->c_data)
                == ((struct cblock *)NULL)->c_data) {
                tp->t_rbuf.c_ptr = NULL;
                cfreelist.c_flag = 1;
                (void) sleep((caddr_t)&cfreelist, TTOPRI);
            }
            tp->t_rbuf.c_count = cfreelist.c_size;
            tp->t_rbuf.c_size = cfreelist.c_size;
            (*tp->t_proc)(tp, T_INPUT);
        }
}

```

```

        break;
}
case LDCLOSE:
    spltty();
    (*tp->t_proc)(tp, T_RESUME);
    SPL0();
    ttywait(tp);
    ttyflush(tp, (FREAD|FWRITE));
    if (tp->t_tbuf.c_ptr) {
        putcf(CMATCH((struct cblock *)tp->t_tbuf.c_ptr));
        tp->t_tbuf.c_ptr = NULL;
        tp->t_tbuf.c_count = 0;
        tp->t_tbuf.c_size = 0;
    }
    if (tp->t_rbuf.c_ptr) {
        putcf(CMATCH((struct cblock *)tp->t_rbuf.c_ptr));
        tp->t_rbuf.c_ptr = NULL;
        tp->t_rbuf.c_count = 0;
        tp->t_rbuf.c_size = 0;
    }
    tp->t_tmflag = 0;
    break;
}
case LDCHG:
    chg = tp->t_lflag ^ arg;
    if (!(chg & ICANON))
        break;
    spltty();
    if (tp->t_canq.c_cc) {
        if (tp->t_rawq.c_cc) {
            tp->t_canq.c_cc += tp->t_rawq.c_cc;
            tp->t_canq.c_cl->c_next = tp->t_rawq.c_cf;
            tp->t_canq.c_cl = tp->t_rawq.c_cl;
        }
        tp->t_rawq = tp->t_canq;
        tp->t_canq = ttнулq;
    }
    tp->t_delct = tp->t_rawq.c_cc;
    SPL0();
    break;
}
default:
    break;
}

```

```

/* @(#)tty.c 1.2 */
/*
 * general TTY subroutines
 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/system.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/tty.h"
#include "sys/ttold.h"
#include "sys/proc.h"
#include "sys/file.h"
#include "sys/conf.h"
#include "sys/termio.h"
#include "sys/sysinfo.h"
#include "sys/var.h"

extern int speed;
extern int tthwat[];
extern int tllwat[];
extern char ttchar[];

/* null clist header */
struct clist ttnulq;

/* canon buffer */
char canonb[CANBSIZ];

/*
 * Input mapping table-- if an entry is non-zero, when the
 * corresponding character is typed preceded by "\" the escape
 * sequence is replaced by the table value. Mostly used for
 * upper-case only terminals.
 */
char maptab[] = {
    000,000,000,000,000,000,000,000,000,000,
    000,000,000,000,000,000,000,000,000,000,
    000,000,000,000,000,000,000,000,000,000,
    000,000,000,000,000,000,000,000,000,000,
    000,'|',000,000,000,000,000,000,'\'',
    '\'',000,000,000,000,000,000,000,000,000,
    000,000,000,000,000,000,000,000,000,000,
    000,000,000,000,000,000,000,000,000,000,
    000,000,000,000,000,000,000,000,000,000,
    000,000,000,000,000,000,000,000,000,000,
    000,000,000,000,000,000,000,000,000,000,
    000,000,000,000,000,000,000,000,000,000,
    000,'A','B','C','D','E','F','G',
    'H','I','J','K','L','M','N','O',
    'P','Q','R','S','T','U','V','W',
    'X','Y','Z',000,000,000,000,000,000,
};

/*
 * common ioctl tty code
 */
ttiocom(tp, cmd, arg, mode)
register struct tty *tp;
{
    register struct user *up;
    register short flag;
    register struct sgttyb *tbp;
    register struct termio *cbp;
    struct termio cb;
    struct sgttyb tb;

    up = &u;
    switch(cmd) {
    case IOCTYPE:
        up->u_rvall = TIOC;
        break;

    case TCSETAW:

```

```

    case TCSETAF:
        ttywait(tp);
        if (cmd == TCSETAF)
            ttyflush(tp, (FREAD|FWRITE));

    case TCSETA:
        cbp = &cb;
        if (copyin((caddr_t)arg, (caddr_t)cbp, sizeof cb)) {
            up->u_error = EFAULT;
            break;
        }
        if (tp->t_line != cbp->c_line) {
            if (cbp->c_line < 0 || cbp->c_line >= linecnt) {
                up->u_error = EINVAL;
                break;
            }
            (*linesw[tp->t_line].l_ioctl)(tp, LDCLOSE, 0, mode);
        }
        flag = tp->t_lflag;
        tp->t_iflag = cbp->c_iflag;
        tp->t_oflag = cbp->c_oflag;
        tp->t_cflag = cbp->c_cflag;
        tp->t_lflag = cbp->c_lflag;
        bcopy((caddr_t)cbp->c_cc, (caddr_t)tp->t_cc, NCC);
        if (tp->t_line != cbp->c_line) {
            tp->t_line = cbp->c_line;
            (*linesw[tp->t_line].l_ioctl)(tp, LDOPEN, 0, mode);
        } else if (tp->t_lflag != flag) {
            (*linesw[tp->t_line].l_ioctl)(tp, LDCHG, flag, mode);
        }
        return(1);

    case TCGETA:
        cbp = &cb;
        cbp->c_iflag = tp->t_iflag;
        cbp->c_oflag = tp->t_oflag;
        cbp->c_cflag = tp->t_cflag;
        cbp->c_lflag = tp->t_lflag;
        cbp->c_line = tp->t_line;
        bcopy((caddr_t)tp->t_cc, (caddr_t)cbp->c_cc, NCC);
        if (copyout((caddr_t)cbp, (caddr_t)arg, sizeof cb))
            up->u_error = EFAULT;
        break;

    case TCSSBRK:
        ttywait(tp);
        if (arg == 0)
            (*tp->t_proc)(tp, T_BREAK);
        break;

    case TCXONC:
        switch (arg) {
        case 0:
            (*tp->t_proc)(tp, T_SUSPEND);
            break;
        case 1:
            (*tp->t_proc)(tp, T_RESUME);
            break;
        case 2:
            (*tp->t_proc)(tp, T_BLOCK);
            break;
        case 3:
            (*tp->t_proc)(tp, T_UNBLOCK);
            break;
        default:
            up->u_error = EINVAL;
        }
        break;

    case TCPLSH:
        switch (arg) {
        case 0:
        case 1:
        case 2:
            ttyflush(tp, (arg - FOPEN) & (FREAD|FWRITE));
            break;

```

```

default:
    up->u_error = EINVAL;
}
break;

/* conversion aide only */
case TIOCGSTP:
    tbp = &tb;
    ttywait(tp);
    ttyflush(tp, (FREAD|FWRITE));
    if (copyin((caddr_t)arg, (caddr_t)tbp, sizeof(tb))) {
        up->u_error = EFAULT;
        break;
    }
    tp->t_iflag = 0;
    tp->t_oflag = 0;
    tp->t_lflag = 0;
    tp->t_cflag = (tbp->sg_ispeed&CBAUD)|CREAD;
    if ((tbp->sg_ispeed&CBAUD) == B110)
        tp->t_cflag |= CSTOPB;
    tp->t_cc[VERASE] = tbp->sg_erase;
    tp->t_cc[VKILL] = tbp->sg_kill;
    flag = tbp->sg_flags;
    if (flag&O_HUPCL)
        tp->t_cflag |= HUPCL;
    if (flag&O_XTABS)
        tp->t_oflag |= TAB3;
    else if (flag&O_TDELAY)
        tp->t_oflag |= TAB1;
    if (flag&O_LCASE) {
        tp->t_iflag |= IUCLC;
        tp->t_oflag |= OLCUC;
        tp->t_lflag |= XCASE;
    }
    if (flag&O_ECHO)
        tp->t_lflag |= ECHO;
    if (!(flag&O_NOAL))
        tp->t_lflag |= ECHOK;
    if (flag&O_CRMOD) {
        tp->t_iflag |= ICRNL;
        tp->t_oflag |= ONLCR;
        if (flag&O_CR1)
            tp->t_oflag |= CR1;
        if (flag&O_CR2)
            tp->t_oflag |= ONOCR|CR2;
    } else {
        tp->t_oflag |= ONLRET;
        if (flag&O_NL1)
            tp->t_oflag |= CR1;
        if (flag&O_NL2)
            tp->t_oflag |= CR2;
    }
    if (flag&O_RAW) {
        tp->t_cc[VTIME] = 1;
        tp->t_cc[VMIN] = 6;
        tp->t_iflag &= ~(ICRNL|IUCLC);
        tp->t_cflag |= CS8;
    } else {
        tp->t_cc[VEOF] = CEOF;
        tp->t_cc[VEOL] = 0;
        tp->t_cc[VEOL2] = 0;
        tp->t_iflag |= BRKINT|IGNPAR|ISTRIP|IXON|IXANY;
        tp->t_oflag |= OPOST;
        tp->t_cflag |= CS7|PARENB;
        tp->t_lflag |= ICANON|ISIG;
    }
    tp->t_iflag |= INPCK;
    if (flag&O_ODDP)
        if (flag&O_EVENP)
            tp->t_iflag &= ~INPCK;
        else
            tp->t_cflag |= PARODD;
    if (flag&O_VTDELAY)
        tp->t_oflag |= FFDLY;
    if (flag&O_BSDLY)
        tp->t_oflag |= BSDLY;

```

```

return(1);

case TIOCGSTP:
    tbp = &tb;
    tbp->sg_ispeed = tp->t_cflag&CBAUD;
    tbp->sg_ospeed = tbp->sg_ispeed;
    tbp->sg_erase = tp->t_cc[VERASE];
    tbp->sg_kill = tp->t_cc[VKILL];
    flag = 0;
    if (tp->t_cflag&HUPCL)
        flag |= O_HUPCL;
    if (!(tp->t_lflag&ICANON))
        flag |= O_RAW;
    if (tp->t_lflag&XCASE)
        flag |= O_LCASE;
    if (tp->t_lflag&ECHO)
        flag |= O_ECHO;
    if (!(tp->t_lflag&ECHOK))
        flag |= O_NOAL;
    if (tp->t_cflag&PARODD)
        flag |= O_ODDP;
    else if (tp->t_iflag&INPCK)
        flag |= O_EVENP;
    else
        flag |= O_ODDP|O_EVENP;
    if (tp->t_oflag&ONLCR) {
        flag |= O_CRMOD;
        if (tp->t_oflag&CR1)
            flag |= O_CR1;
        if (tp->t_oflag&CR2)
            flag |= O_CR2;
    } else {
        if (tp->t_oflag&CR1)
            flag |= O_NL1;
        if (tp->t_oflag&CR2)
            flag |= O_NL2;
    }
    if ((tp->t_oflag&TABDLY) == TAB3)
        flag |= O_XTABS;
    else if (tp->t_oflag&TAB1)
        flag |= O_TDELAY;
    if (tp->t_oflag&FFDLY)
        flag |= O_VTDELAY;
    if (tp->t_oflag&BSDLY)
        flag |= O_BSDLY;
    tbp->sg_flags = flag;
    if (copyout((caddr_t)tbp, (caddr_t)arg, sizeof(tb)))
        up->u_error = EFAULT;
    break;

/*
 * The following ioctl's were added by UniSoft
 */

/*
 * Return number of characters immediately available.
 */
case FIONREAD: {
    off_t nread;

    SPL6();
    while (tp->t_rawq.c_cc && tp->t_delct)
        canon(tp);
    SPL0();
    nread = tp->t_canq.c_cc;
    if (!(tp->t_lflag & ICANON))
        nread += tp->t_rawq.c_cc;
    if (copyout((caddr_t)&nread, (caddr_t)arg, sizeof(off_t)))
        up->u_error = EFAULT;
    break;
}

default:
    if ((cmd&IOCTYPE) == LDIOC)
        (*linesw[tp->t_line].l_ioctl)(tp, cmd, arg, mode);
    else

```

```

        up->u_error = EINVAL;
        break;
    }
    return(0);
}

ttinit(tp)
register struct tty *tp;
{
    tp->t_line = 0;
    tp->t_iflag = 0;
    tp->t_oflag = 0;
    tp->t_cflag = cspeed|CS8|CREAD|HUPCL;
    tp->t_lflag = 0;
    bcopy((caddr_t)ttochar, (caddr_t)tp->t_cc, NCC);
}

ttywait(tp)
register struct tty *tp;
{
    spltty();
    while (tp->t_outq.c_cc || (tp->t_state&(BUSY|TIMEOUT))) {
        tp->t_state |= TTIOW;
        (void) sleep((caddr_t)&tp->t_oflag, TTOPRI);
    }
    SPL0();
    delay(v.v_hz>>4);
}

/*
 * flush TTY queues
 */
ttyflush(tp, cmd)
register struct tty *tp;
{
    register struct cblock *cp;
    register s;

    if (cmd&FWRITE) {
        while ((cp = getcb(&tp->t_outq)) != NULL)
            putcf(cp);
        (*tp->t_proc)(tp, T_WFLUSH);
        if (tp->t_state&OASLP) {
            tp->t_state &= ~OASLP;
            wakeup((caddr_t)&tp->t_outq);
        }
        if (tp->t_state&TTIOW) {
            tp->t_state &= ~TTIOW;
            wakeup((caddr_t)&tp->t_oflag);
        }
    }
    if (cmd&FREAD) {
        while ((cp = getcb(&tp->t_canq)) != NULL)
            putcf(cp);
        s = spltty();
        while ((cp = getcb(&tp->t_rawq)) != NULL)
            putcf(cp);
        tp->t_delct = 0;
        splx(s);
        (*tp->t_proc)(tp, T_RFLUSH);
        if (tp->t_state&IASLP) {
            tp->t_state &= ~IASLP;
            wakeup((caddr_t)&tp->t_rawq);
        }
    }
}

/*
 * Transfer raw input list to canonical list,
 * doing erase-kill processing and handling escapes.
 */
canon(tp)
register struct tty *tp;
{
    register char *bp;
    register c, esc;

```

```

    spltty();
    if (tp->t_rawq.c_cc == 0)
        tp->t_delct = 0;
    while (tp->t_delct == 0) {
        if (!(tp->t_state&CARR_ON) || (u.u_fmode&FNDELAY)) {
            SPL0();
            return;
        }
        if (!(tp->t_lflag&ICANON) && tp->t_cc[VMIN]==0) {
            if (tp->t_cc[VTIME]==0)
                break;
            tp->t_state &= -RTO;
            if (!(tp->t_state&TACT))
                tttimeo(tp);
        }
        tp->t_state |= IASLP;
        (void) sleep((caddr_t)&tp->t_rawq, TTIPRI);
    }
    if (!(tp->t_lflag&ICANON)) {
        if (tp->t_canq.c_cc == 0) {
            tp->t_canq = tp->t_rawq;
            tp->t_rawq = ttнулq;
        } else
            while (tp->t_rawq.c_cc)
                (void) putc(getc(&tp->t_rawq), &tp->t_canq);
        tp->t_delct = 0;
        SPL0();
        return;
    }
    SPL0();
    bp = canonb;
    esc = 0;
    while ((c=getc(&tp->t_rawq)) >= 0) {
        if (!esc) {
            if (c == '\\') {
                esc++;
            } else if (c == tp->t_cc[VERASE]) {
                if (bp > canonb)
                    bp--;
                continue;
            } else if (c == tp->t_cc[VKILL]) {
                bp = canonb;
                continue;
            } else if (c == tp->t_cc[VEOF]) {
                break;
            }
        } else {
            esc = 0;
            if (c == tp->t_cc[VERASE] ||
                c == tp->t_cc[VKILL] ||
                c == tp->t_cc[VEOF])
                bp--;
            else if (tp->t_lflag&XCASE) {
                if ((c < 0200) && maptab[c]) {
                    bp--;
                    c = maptab[c];
                } else if (c == '\\')
                    continue;
            } else if (c == '\\')
                esc++;
        }
        *bp++ = c;
        if (c == '\n' || c == tp->t_cc[VEOL] || c == tp->t_cc[VEOL2])
            break;
        if (bp >= &canonb[CANBSIZ])
            bp--;
    }
    tp->t_delct--;
    c = bp - canonb;
    sysinfo.canch += c;
    bp = canonb;
    /* faster copy ? */
    while (c--)
        (void) putc(*bp++, &tp->t_canq);
    return;
}

```

```

}
/*
 * Restart typewriter output following a delay timeout.
 * The name of the routine is passed to the timeout
 * subroutine and it is called during a clock interrupt.
 */
ttrstrt(tp)
register struct tty *tp;
{
    (*tp->t_proc)(tp, T_TIME);
}

```

```

/*      udp_usrreq.c      4.45      83/02/16      */

#include "sys/param.h"
#include "sys/config.h"
#include "sys/errno.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/systm.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "net/misc.h"
#include "net/mbuf.h"
#include "net/protosw.h"
#include "net/socket.h"
#include "net/socketvar.h"
#include "net/in.h"
#include "net/if.h"
#include "net/route.h"
#include "net/in_pcb.h"
#include "net/in_systm.h"
#include "net/ip.h"
#include "net/ip_var.h"
#include "net/ip_icmp.h"
#include "net/udp.h"
#include "net/udp_var.h"
#include "errno.h"

/*
 * UDP protocol implementation.
 * Per RFC 768, August, 1980.
 */
udp_init()
{
    udb.inp_next = udb.inp_prev = &udb;
}

int      udpcksum;
struct   sockaddr_in udp_in = { AF_INET };

udp_input(m0)
    struct mbuf *m0;
{
    register struct udpiphdr *ui;
    register struct inpcb *inp;
    register struct mbuf *m;
    int len;

    /*
     * Get IP and UDP header together in first mbuf.
     */
    m = m0;
    if ((m->m_off > MMAXOFF || m->m_len < sizeof (struct udpiphdr)) &&
        (m = m_pullup(m, sizeof (struct udpiphdr))) == 0) {
        udpstat.udps_hdrops++;
        return;
    }
    ui = mtod(m, struct udpiphdr *);
    if (((struct ip *)ui)->ip_hl > (sizeof (struct ip) >> 2))
        ip_stripoptions((struct ip *)ui, (struct mbuf *)0);

    /*
     * Make mbuf data length reflect UDP length.
     * If not enough data to reflect UDP length, drop.
     */
    len = ntohs((u_short)ui->ui_ulen);
    if (((struct ip *)ui)->ip_len != len) {
        if (len > ((struct ip *)ui)->ip_len) {
            udpstat.udps_badlen++;
            goto bad;
        }
        m_adj(m, ((struct ip *)ui)->ip_len - len);
        /* (struct ip *)ui->ip_len = len; */
    }

    }
}

/*
 * Checksum extended UDP header and data.
 */
if (udpcksum && ui->ui_sum) {
    register u_short csum = ui->ui_sum;

    if (csum == 0xffff)
        csum = 0;
    ui->ui_next = ui->ui_prev = 0;
    ui->ui_x1 = 0;
    ui->ui_len = htons((u_short)len);
    ui->ui_sum = 0;
    if (csum != in_cksum(m, len + sizeof (struct ip))) {
        udpstat.udps_badsum++;
        m_freem(m);
        return;
    }
}

/*
 * Locate pcb for datagram.
 */
inp = in_pcblookup(&udb,
    ui->ui_src, ui->ui_sport, ui->ui_dst, ui->ui_dport,
    INPLOOKUP_WILDCARD);
if (inp == 0) {
    /* don't send ICMP response for broadcast packet */
    if (in_lnaof(ui->ui_dst) == INADDR_ANY)
        goto bad;
    icmp_error((struct ip *)ui, ICMP_UNREACH, ICMP_UNREACH_PORT);
    return;
}

/*
 * Construct sockaddr format source address.
 * Stuff source address and datagram in user buffer.
 */
udp_in.sin_port = ui->ui_sport;
udp_in.sin_addr = ui->ui_src;
m->m_len -= sizeof (struct udpiphdr);
m->m_off += sizeof (struct udpiphdr);
if (sbappendaddr(&inp->inp_socket->so_rcv, (struct sockaddr *)&udp_in, m) == 0)
    goto bad;
sorwakeup(inp->inp_socket);
return;

bad:
    m_freem(m);
}

udp_abort(inp)
    struct inpcb *inp;
{
    struct socket *so = inp->inp_socket;

    in_pcbdisconnect(inp);
    soisdisconnected(so);
}

udp_ctlinput(cmd, arg)
    int cmd;
    caddr_t arg;
{
    struct in_addr *sin;
    extern u_char inetctlerrmap[];

    if (cmd < 0 || cmd > PRC_NCMLS)
        return;
    switch (cmd) {

    case PRC_ROUTEDEAD:
        break;

    case PRC_QUENCH:
        break;
    }
}

```

```

/* these are handled by ip */
case PRC_IFDOWN:
case PRC_HOSTDEAD:
case PRC_HOSTUNREACH:
    break;

default:
    sin = &((struct icmp *)arg)->icmp_ip.ip_dst;
    in_pcbnotify(&udb, sin, (int)inetctlerrmap[cmd], udp_abort);
}
}

udp_output(inp, m0)
struct inpcb *inp;
struct mbuf *m0;
{
    register struct mbuf *m;
    register struct udpiphdr *ui;
    register struct socket *so;
    register int len = 0;

    /*
     * Calculate data length and get a mbuf
     * for UDP and IP headers.
     */
    for (m = m0; m; m = m->m_next)
        len += m->m_len;
    /* we don't have MT_HEADER's (yet?)
     * m = m_get(M_DONTWAIT, MT_HEADER);
     */
    m = m_get(M_DONTWAIT);
    if (m == 0) {
        m_freem(m0);
        return (ENOBUFS);
    }

    /*
     * Fill in mbuf with extended UDP header
     * and addresses and length put into network format.
     */
    m->m_off = MMAXOFF - sizeof (struct udpiphdr);
    m->m_len = sizeof (struct udpiphdr);
    m->m_next = m0;
    ui = mtod(m, struct udpiphdr *);
    ui->ui_next = ui->ui_prev = 0;
    ui->ui_xl = 0;
    ui->ui_pr = IPPROTO_UDP;
    ui->ui_len = len + sizeof (struct udphdr);
    ui->ui_src = inp->inp_laddr;
    ui->ui_dst = inp->inp_faddr;
    ui->ui_sport = inp->inp_lport;
    ui->ui_dport = inp->inp_fport;
    ui->ui_ulen = htons((u_short)ui->ui_len);
    ui->ui_len = ui->ui_ulen;

    /*
     * Stuff checksum and output datagram.
     */
    ui->ui_sum = 0;
    if (udpcksum) {
        ui->ui_sum = in_cksum(m, sizeof (struct udpiphdr) + len);
        if (ui->ui_sum == 0)
            ui->ui_sum = 0xffff;
    }
    ((struct ip *)ui)->ip_len = sizeof (struct udpiphdr) + len;
    ((struct ip *)ui)->ip_ttl = MAXTTL;
    so = inp->inp_socket;
    return (ip_output(m, (struct mbuf *)0,
        (so->so_options & SO_DONTROUTE) ? &routetoif : (struct route *)0,
        so->so_state & SS_PRIV));
}

/*ARGSUSED*/
udp_usrreq(so, req, m, nam)
    struct socket *so;

```

```

int req;
struct mbuf *m, *nam;

struct inpcb *inp = sotoinpcb(so);
int error = 0;

if (inp == NULL && req != PRU_ATTACH) {
    error = EINVAL;
    goto release;
}

switch (req) {

case PRU_ATTACH:
    if (inp != NULL) {
        error = EINVAL;
        break;
    }
    /* billn. for now, attach the old 4.1a way
     error = in_pcballoc(so, &udb);
     if (error)
         break;
     error = soreserve(so, 2048, 2048);
     */
    error = in_pcbattach(so, &udb, 2048, 2048,
        (struct sockaddr_in *)nam);
    if (error)
        break;
    break;

case PRU_DETACH:
    if (inp == NULL) {
        error = ENOTCONN;
        break;
    }
    in_pcbdetach(inp);
    break;

/* "bind" is not in the system (yet?)
case PRU_BIND:
    error = in_pcbbind(inp, nam);
    break;

/* neither is listen
case PRU_LISTEN:
    error = EOPNOTSUPP;
    break;

*/
case PRU_CONNECT:
    if (inp->inp_faddr.s_addr != INADDR_ANY) {
        error = EISCONN;
        break;
    }
    error = in_pcbconnect(inp, (struct sockaddr_in *)nam);
    if (error == 0)
        soisconnected(so);
    break;

case PRU_ACCEPT:
    error = EOPNOTSUPP;
    break;

case PRU_DISCONNECT:
    if (inp->inp_faddr.s_addr == INADDR_ANY) {
        error = ENOTCONN;
        break;
    }
    in_pcbdisconnect(inp);
    soisdisconnected(so);
    break;

case PRU_SHUTDOWN:
    socantsendmore(so);
    break;

case PRU_SEND: {

```



```
    struct in_addr laddr;

    if (nam) {
        laddr = inp->inp_laddr;
        if (inp->inp_faddr.s_addr != INADDR_ANY) {
            error = EISCONN;
            break;
        }
        error = in_pcbconnect(inp, (struct sockaddr_in *)nam);
        if (error)
            break;
    } else {
        if (inp->inp_faddr.s_addr == INADDR_ANY) {
            error = ENOTCONN;
            break;
        }
    }
    error = udp_output(inp, m);
    m = NULL;
    if (nam) {
        in_pcbdisconnect(inp);
        inp->inp_laddr = laddr;
    }
    break;

case PRU_ABORT:
    in_pcbdetach(inp);
    sofree(so);
    soisdisconnected(so);
    break;

case PRU_CONTROL:
    error = EOPNOTSUPP;
    break;

case PRU_SOCKADDR:
    in_setsockaddr((struct sockaddr_in *)nam, (struct inpcb *)inp);
    break;

default:
    panic("udp_usrreq");
}
release:
if (m != NULL)
    m_freem(m);
return (error);
}
```

```
/* @(#)utssys.c 1.3 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/buf.h"
#include "sys/filsys.h"
#include "sys/mount.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/var.h"
#include "sys/utsname.h"

utssys()
{
    register i;
    register struct a {
        char    *cbuf;
        int     mv;
        int     type;
    } *uap;
    struct {
        daddr_t f_tfree;
        inc_t   f_tinode;
        char    f_fname[6];
        char    f_fpack[6];
    } ust;
    register struct user *up;

    up = &u;
    uap = (struct a *)up->u_ap;
    switch(uap->type) {

case 0:        /* uname */
    if (copyout((caddr_t)&utsname, uap->cbuf, sizeof(struct utsname)))
        up->u_error = EFAULT;
    return;

/* case 1 was umask */

case 2:        /* ustat */
    for(i=0; i<v.v_mount; i++) {
        register struct mount *mp;

        mp = &mount[(short)i];
        if (mp->m_flags==MINUSE && brdev(mp->m_dev)==brdev(uap->mv)) {
            register struct filsys *fp;

            fp = mp->m_bufp->b_un.b_filsys;
            ust.f_tfree = PsLTOP(mp->m_dev, fp->s_tfree);
            ust.f_tinode = fp->s_tinode;
            bcopy(fp->s_fname, ust.f_fname, sizeof(ust.f_fname));
            bcopy(fp->s_fpack, ust.f_fpack, sizeof(ust.f_fpack));
            if (copyout((caddr_t)&ust, uap->cbuf, 18))
                up->u_error = EFAULT;
            return;
        }
    }
    up->u_error = EINVAL;
    return;

case 33:       /* uvar */
    if (copyout((caddr_t)&v, uap->cbuf, sizeof(struct var)))
        up->u_error = EFAULT;
    return;

default:
    up->u_error = EFAULT;
}
}
```

```

/*
 * Copyright 1982 UniSoft Corporation
 * Use of this material is subject to your disclosure agreement with
 * AT&T, Western Electric and UniSoft Corporation.
 *
 * VT100 emulator
 * Called with each character destined for the console.
 * Processes control sequences and keeps track of terminal state.
 * Calls into bitmap.c actually do the I/O
 */

/* SENSESCRN resets the contrast on screen output (via the console device
 * driver) as if a key was hit.
 */

#include "sys/types.h"
#include "sys/l2.h"

#define FAST
#define SENSESCRN

#define MAXVT_N 255 /* maximum value of any numeric parameter */
#define MAXPARAMS 10 /* maximum number of numeric parameters */

short vt_n[MAXPARAMS]; /* numeric parameters of \E[ commands */
char vt_mparam;
char vt_tabset[88] = { 0 };
extern char bmbck, bmcolor, bmnorm;
extern char *bmscrn;
extern char kb_altkp;
short vt_maxrow = 38;
short vt_maxcol = 88;
short vt_row, vt_col; /* cursor location (0-vt_maxrow, 0-vt_maxcol) */
short vtrow_ofs = 1;
short vtcol_ofs = 1; /* row and column offsets */
short vt_winscr1 = 1; /* lines to scroll each time */

/* This routine interprets the characters destined for the console and
 * performs like a VT100 . It is implemented in terms of primitives
 * defined in bitmap.c
 */

#define CBKSP 1
#define CCR 2
#define CLF 3
#define CHTAB 4
#define CESC 5
#define CBELL 6
#define CGARB 7
#define CCHAR 8

char vt_keytype[] = {
    CGARB, CGARB, CGARB, CGARB, CGARB, CGARB, CGARB, CBELL,
    CBKSP, CHTAB, CLF, CLF, CLF, CCR, CGARB, CGARB,
    CGARB, CGARB, CGARB, CGARB, CGARB, CGARB, CGARB,
    CGARB, CGARB, CGARB, CESC, CGARB, CGARB, CGARB, CGARB,

    CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CCHAR,
    CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CCHAR,
    CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CCHAR,
    CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CCHAR,
    CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CCHAR,
    CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CCHAR, CGARB,
};

vt_putc (c)
register char c;
{
extern int (*te_putc)();
int vt_esc();
int x;

extern time_t lbolt;

l2_dtrape = lbolt + l2_dtime;
if (l2_dimmed) l2undim();

if (c >= ' ') {
    bputc(vt_row+vtrow_ofs,vt_col+vtcol_ofs,c);
    vt_advance();
    bminvert (vt_row+vtrow_ofs, vt_col+vtcol_ofs);
    return;
}
bminvert (vt_row+vtrow_ofs, vt_col+vtcol_ofs);
switch (vt_keytype[c]) {
    case CBKSP:
        if (vt_col > 0)
            vt_col--;
        break;
    case CCR:
        vt_col = 0;
        break;
    case CLF:
        if (++vt_row >= vt_maxrow) {
            blt(bmscrn+90*9,bmscrn+90*9*2,9*90*38);
            vt_row -= vt_winscr1;
        }
        break;
    case CHTAB:
        for (x = vt_col+1; x < vt_maxcol-1; x++)
            if (vt_tabset[x] == 1)
                break;
        vt_col = x;
        break;
    case CESC:
        te_putc = vt_esc;
        break;
    case CBELL:
        beep();
        break;
    case CGARB:
        break;
}
bminvert(vt_row+vtrow_ofs,vt_col+vtcol_ofs);
}

/*
 * Process the escape sequences to the terminal
 */
vt_esc(c) /* after ESC key hit */
register char c;
{
extern int (*te_putc)();
int vt_brck(), vt_putc();

switch (c) {
    case '[':
        te_putc = vt_brck; /* check E[ sequence */
        return;
    case '>': /* disable alternate keypad */
        kb_altkp = 0;
        break;
    case '=': /* enable alternate keypad */
        kb_altkp = 1;
        break;
    case 'M': /* reverse scroll */
        bminvert(vt_row+vtrow_ofs,vt_col+vtcol_ofs);
        bmscr1();
        bminvert(vt_row+vtrow_ofs,vt_col+vtcol_ofs);

        break;
    case 'H':
        vt_tabset[vt_col]=1;
}
}

```

```

        break;
    }
    te_putc = vt_putc;
    return;
}

vt_brck(c)                                /* \E[ sequence checked here */
register char c;
{
    extern int (*te_putc)();
    int vt_param(), vt_attrb();

    vt_mparam = 0;
    vt_n[0] = vt_n[1] = 0;
    if (c == ';') { /* missing 1st number - look for 2nd */
        te_putc = vt_param;
        vt_mparam++;
        return;
    }
    if (c >= '0' && c <= '9' ) {
        vt_n[0] = c - '0';
        te_putc = vt_param;
        return;
    }
    if (c == '?') {
        te_putc = vt_attrb;
        return;
    }
    vt_cmd(c);
}

vt_param(c)
register char c;
{
    register tmp;
    int vt_putc();

    if (c >= '0' && c <= '9' ) {
        tmp = (vt_n[vt_mparam] * 10) + (c - '0');
        vt_n[vt_mparam] = (tmp > MAXVT_N) ? MAXVT_N : tmp;
        return;
    }
    if (c == ';') {
        if (++vt_mparam >= MAXPARAMS) { /* too many parameters */
            te_putc = vt_putc;
            return;
        }
        vt_n[vt_mparam] = 0;
        return;
    }
    vt_cmd(c);
}

vt_cmd(c)                                /* now have last char of esc sequence */
register char c;
{
    extern int (*te_putc)();
    register vt_n1 = vt_n[0];
    register vt_n2 = vt_n[1];
    register int x, y;
    int vt_putc();

    if (c == 'f') c = 'H';
    if ((c >= 'A') && (c <= 'Z')) {
        bminvert(vt_row+vtrow_ofs, vt_col+vtcol_ofs);
        switch (c) {
            case 'A': /* move cursor up */
                if (vt_n1 == 0)
                    vt_n1 = 1;
                vt_row = (vt_n1 < vt_row) ? (vt_row - vt_n1) : 0;
                break;
            case 'B': /* move cursor down */
                if (vt_n1 == 0)
                    vt_n1 = 1;
                y = vt_row + vt_n1;
                vt_row = (y < vt_maxrow) ? y : vt_maxrow-1;

```

```

                break;
            case 'C': /* move cursor right */
                if (vt_n1 == 0)
                    vt_n1 = 1;
                y = vt_col + vt_n1;
                vt_col = (y < vt_maxcol) ? y : vt_maxcol-1;
                break;
            case 'D': /* move cursor left */
                if (vt_n1 == 0)
                    vt_n1 = 1;
                vt_col = (vt_n1 < vt_col) ? (vt_col - vt_n1) : 0;
                break;
            case 'H': /* move cursor home */
                if (vt_n1 == 0)
                    vt_row = 0;
                else if ((vt_row = vt_n1-1) >= vt_maxrow)
                    vt_row = vt_maxrow - 1;
                if (vt_n2 == 0)
                    vt_col = 0;
                else if ((vt_col = vt_n2-1) >= vt_maxcol)
                    vt_col = vt_maxcol - 1;
                break;
            case 'J': /* clear screen */
                if (vt_n1 == 0)
                    if ((vt_row == 0) && (vt_col == 0))
                        vt_n1 = 2;
                switch (vt_n1) {
                    case 0: /* clear from cursor to end */
                        for (y = vt_col ; y < vt_maxcol ; y++)
                            bputc(vt_row+vtrow_ofs,y+vtcol_ofs,' ');
                        for (x = vt_row+1; x < vt_maxrow ; x++)
                            bmbblank(x+vtrow_ofs);
                        break;
                    case 1: /* clear from beginning to cursor */
                        for (x = 0; x < vt_row ; x++)
                            bmbblank(x+vtrow_ofs);
                        for (y = 0; y <= vt_col ; y++)
                            bputc(vt_row+vtrow_ofs,y+vtcol_ofs,' ');
                        break;
                    case 2: /* clear entire screen */
                        bmclean();
                }
                break;
            case 'K': /* clear line */
                switch (vt_n1) {
                    case 0: /* clear from cursor to end */
                        for (y = vt_col ; y < vt_maxcol ; y++)
                            bputc(vt_row+vtrow_ofs,y+vtcol_ofs,' ');
                        break;
                    case 1: /* clear from beginning to cursor */
                        for (y = 0 ; y <= vt_col ; y++)
                            bputc(vt_row+vtrow_ofs,y+vtcol_ofs,' ');
                        break;
                    case 2: /* clear entire line */
                        bmbblank(vt_row+vtrow_ofs);
                }
                break;
            case 'L': /* insert line(s) */
                if (vt_n1 == 0)
                    vt_n1 = 1;
                if (vt_n1 > vt_maxrow - vt_row)
                    vt_n1 = vt_maxrow - vt_row;
                for (x=vt_maxrow-1, y=vt_maxrow-vt_n1-1; y >= vt_row; x--, y--)
                    bmcpl(x+vtrow_ofs, y+vtrow_ofs);
                for ( ; x >= vt_row ; x-- )
                    bmbblank(x+vtrow_ofs);
                break;
            case 'M': /* delete line(s) */
                if (vt_n1 == 0)
                    vt_n1 = 1;
                if (vt_n1 > vt_maxrow - vt_row)
                    vt_n1 = vt_maxrow - vt_row;
                for (x=vt_row, y=vt_row+vt_n1; y < vt_maxrow; x++, y++)
                    bmcpl(x+vtrow_ofs, y+vtrow_ofs);
                for ( ; x < vt_maxrow ; x++ )
                    bmbblank(x+vtrow_ofs);

```

```

        break;
    case 'P': /* delete character(s) */
        if (vt_n1 == 0)
            vt_n1 = 1;
        if (vt_n1 > vt_maxcol - vt_col)
            vt_n1 = vt_maxcol - vt_col;
        for (x=vt_col, y=vt_col+vt_n1; y < vt_maxcol; x++, y++)
            bmmvc(vt_row+vtrow_ofs, x+vtcol_ofs,
                 vt_row+vtrow_ofs, y+vtcol_ofs);
        for ( ; x < vt_maxcol ; x++ )
            bmputc(vt_row+vtrow_ofs,x+vtcol_ofs,' ');
        break;
    }
    bminvert (vt_row+vtrow_ofs, vt_col+vtcol_ofs);
} else {
    switch (c) {
    case 'g':
        if (vt_n1 == 0)
            vt_tabset[vt_col] = 0;
        else if (vt_n1 == 3)
            for (x = 0 ; x < vt_maxcol ; x++ )
                vt_tabset[x] = 0;
        break;
    case 'm': /* set normal display or reverse video */
        for (x = 0; x <= vt_mparam; x++)
            switch (vt_n[x]) {
            case 0: /* turn underline off, set normal background */
                if (bmbck != bmnormal) {
                    bmswitch(); /* invert font table */
                    bmbck = bmnormal;
                }
                bmcOLOR = bmbck; /* underline off */
                break;
            case 1: case 7: /* set reverse image */
                if (bmbck == bmnormal) {
                    bmswitch(); /* invert font table */
                    bmbck = bmnormal ? 0 : -1;
                }
                break;
            case 4: /* set underline */
                bmcOLOR = bmbck ? 0 : -1;
            }
        break;
    }
}
te_putc = vt_putc;
}

vt_attrb(c) /* \E[? */
char c;
{
    int vt_attrb(), vt_putc();

    if (c >= '0' && c <= '9')
        te_putc = vt_attrb;
    else
        te_putc = vt_putc;
}

vt_attrb()
{
    int vt_putc();

    te_putc = vt_putc;
}

vt_advance()
{
    if (++vt_col >= vt_maxcol) { /* wraps around */
        vt_col = 0;
        if (++vt_row >= vt_maxrow) { /* on last line */
            b1t(bmscrn+90*9,bmscrn+90*9*2,9*90*38);
            vt_row -- vt_winscr1;
        }
    }
}
}

```

```

/*
 * Copyright 1982 UniSoft Corporation
 * Use of this material is subject to your disclosure agreement with
 * AT&T, Western Electric and UniSoft Corporation.
 *
 * Bitmap Display Driver
 *
 * The screen is 720 by 364 raster units. A comfortable pixel size is
 * 9 by 10 giving 80 columns (9 wide) and 36 lines (10 deep). The
 * extra 4 raster rows are unused. Unfortunately this arrangement is
 * very expensive in processor time since pixels lie across byte boundaries.
 *
 * For the time being the pixel size used will be 8 by 9 providing a
 * screen area of 90 columns, and 40 lines.
 *
 * This is the low level display driver intended to simplify the task
 * of writing higher level screen emulators by simulating just the common
 * aspects of all crts in the following functions:
 *
 *      bminit ()          clears entire display
 *      bminvert (row, col) invert character at row, column
 *      bmputc (r, c, char) put the character at row, column
 *      bmwin (cmd, t, b, l, r) modifies window (top,bot,left,right);
 *                          cmd == 1 means scroll up a line
 *                          cmd == 0 means clear that area
 */
#define FAST
#define NODEBUG
#include <sys/mmu.h>
#include <sys/types.h>
#include <sys/local.h>
#include <sys/bmfont.h>

char *bmscrn;          /* pointer to screen -- initialized in bminit */
char bmcolor;         /* underline color; same as bmback for no underlining */
char bmback;          /* current background color; 0 for white, -1 for black */
char bmnormal;        /* normal background color; 0 for white, -1 for black */
char bmfirst = 1;     /* is this the first boot? (not true for a restart) */

bminit ()              /* one time screen initialization (called from mch.s) */
{
    register short i;          /* amount of screen in longs */
    register long *p;
    extern long bblank[];

    *MEMEND == SCRNSIZE;      /* lop off mem for screen */
    bmscrn = *MEMEND;         /* logical screen address */
    VIDADDR = (int)(bmscrn+MEMBASE)>>15); /* init screen addr latch */

    i = ((SCRNSIZE - (MAXCOL * MAXROW * V_RESO)) >> 2) - 1;
    p = (long *) (bmscrn + (MAXCOL * MAXROW * V_RESO));

    do {
        /* so retrace lines won't show */
        *p++ = -1;
    } while (--i != -1);
    bmcolor = 0; /* bmcolor switches in bminit if bmback is -1 */
    bmback = -1;
    bmnormal = -1;
    bmclear();
    if (bmfirst) {
        bmfirst = 0;
        bmswitch();
        bmclear();
    } else {
        bmcolor = -1;
        for (i=0; i<16; i++)
            bblank[i] = -1;
        bmclear();
    }
}

bmswitch()
{
    bmcolor = (bmcolor ? 0 : -1);
    bmifont();
}

bmifont()              /* invert the font table (set or reset rev. video) */
{
    register short i;          /* amount of screen in longs */
    register char *f, g;
    extern long bblank[];

    i = sizeof(bmfont) - 1;
    f = bmfont;
    do {
        g = ~*f;
        *f++ = g;
    } while (--i != -1);
    for (i=0; i<16; i++)
        bblank[i] = ( bblank[i] ? 0 : -1);
}

bmsinv()              /* invert the entire screen */
{
    register short i;          /* amount of screen in chars */
    register char *p, q;

    i = MAXCOL * MAXROW * V_RESO;
    p = bmscrn;

    do {
        q = ~*p;
        *p++ = q;
    } while (--i != 0);
}

#endif FAST
/* Modify a section of the display
 * Cmd is 0 to clear the area, positive to scroll up that many lines
 * or negative to scroll down that many.
 */
bmwin (cmd, tr, br, lc, rc)
    short cmd, br, lc, rc;
    register short tr;
{
    register short wrap, j, i;
    short cols, rows;
    register char *pc, *ec;
    register char color = bmcolor;

    pc = bmscrn;

#endif NODEBUG
    if (tr<0||tr>=MAXROW||br>=MAXCOL||br<tr
        ||lc<0||lc>=MAXCOL||rc>=MAXCOL||rc<=lc) {
        printf("bmwin(%s,%d,%d,%d,%d) is illegal\n",
            cmd?"scroll":"clear", tr, br, lc, rc);
        return;
    }
#endif

/* Calculate number of logical lines to operate on */
    rows = br - tr + 1;

/* Calculate number of columns (bytes) accross screen */
    cols = rc - lc + 1;

/* Wrap is an offset to be added to scan line pointers at the end of
 * each line to bring them around to the start of the next line.
 * It is the number of columns not being scrolled (usually zero)
 */
    wrap = BPL - cols; /* wrap around offset */

/* Pc points at the upper area of the screen (ie, that which will
 * receive the scroll data).
 */
    pc += tr * (short) (BPL * V_RESO) + lc;

/* I is the number of scan lines to operate on. ie. the number of
 * lines in the window minus the number of lines to scroll.

```

```

*/
i = (short)(rows - cmd) * (short)(V_RESO) - 1; /* scan lines */

/* Here we do the scrolling if it is indicated */
if (cmd) {
    /* scroll command lines */

/* Ec is is set to point at the first line to scroll up */
ec = pc + cmd * ((short)(BPL) * (short)(V_RESO));

do {
    /* scroll scan line at a time */
    j = cols - 1; /* column counter */
    do *pc++ = *ec++; /* optimizes to dbra loop */
    while (--j != -1);
    pc += wrap; /* skip to next scan line */
    ec += wrap;
} while (--i != -1);

i = (short)(V_RESO) * cmd - 1; /* reset for clear */
}
do {
    /* clear */
    j = cols - 1;
    do *pc++ = color;
    while (--j != -1);
    pc += wrap;
} while (--i != -1);
return;
}
#endif

bminvert (r, c) /* reverse video char under cursor */
register short r, c;
{
    register char *p = bmscrn;
    register short i;

    p += (r * (BPL * V_RESO)) + c;
    i = V_RESO;
    do {
        *p = ~*p;
        p += BPL; /* 1 scan line */
    } while (--i != 0);
}

bmmvc (dr, dc, sr, sc) /* copy char at (sr,sc) to (dr,dc) */
register short dr, dc;
register short sr, sc;
{
    register char *dest, *src;
    register short i;

    dest = bmscrn + (dr * (BPL * V_RESO)) + dc;
    src = bmscrn + (sr * (BPL * V_RESO)) + sc;
    i = V_RESO;
    do {
        *dest = *src;
        dest += BPL; /* 1 scan line */
        src += BPL; /* 1 scan line */
    } while (--i != 0);
}

#endif
#endif FAST
bmcpl(dl, sl) /* copy line sl to dl */
register dl, sl;
{
    register char *dest, *src;
    register short i;

    dest = bmscrn + (dl * (BPL * V_RESO));
    src = bmscrn + (sl * (BPL * V_RESO));
    i = BPL * V_RESO;
    do {
        *dest = *src;
        dest++; /* 1 scan line */
        src++; /* 1 scan line */
    } while (--i != 0);
}

```

```

#else FAST
bmcpl(dl, sl) /* copy line sl to dl */
register dl, sl;
{
    register char *dest, *src;

    dest = bmscrn + (dl * (BPL * V_RESO)); /* register a5 */
    src = bmscrn + (sl * (BPL * V_RESO)); /* register a4 */

/* Use the 13 registers a6, a3-a0 and d7-d0 to copy 52 bytes at a time. */
asm(" moveml #0xFFFF,sp@- "); /* save all the registers */
asm(" moveml a4@,#0x4FFF "); /* move 52 bytes */
asm(" moveml #0x4FFF,a5@ ");
asm(" moveml a4@(52),#0x4FFF "); /* move 52 bytes */
asm(" moveml #0x4FFF,a5@(52) ");
asm(" moveml a4@(104),#0x4FFF "); /* move 52 bytes */
asm(" moveml #0x4FFF,a5@(104) ");
asm(" moveml a4@(156),#0x4FFF "); /* move 52 bytes */
asm(" moveml #0x4FFF,a5@(156) ");
asm(" moveml a4@(208),#0x4FFF "); /* move 52 bytes */
asm(" moveml #0x4FFF,a5@(208) ");
asm(" moveml a4@(260),#0x4FFF "); /* move 52 bytes */
asm(" moveml #0x4FFF,a5@(260) ");
asm(" moveml a4@(312),#0x4FFF "); /* move 52 bytes */
asm(" moveml #0x4FFF,a5@(312) ");
asm(" moveml a4@(364),#0x4FFF "); /* move 52 bytes */
asm(" moveml #0x4FFF,a5@(364) ");
asm(" moveml a4@(416),#0x4FFF "); /* move 52 bytes */
asm(" moveml #0x4FFF,a5@(416) ");
asm(" moveml a4@(468),#0x4FFF "); /* move 52 bytes */
asm(" moveml #0x4FFF,a5@(468) ");
asm(" moveml a4@(520),#0x4FFF "); /* move 52 bytes */
asm(" moveml #0x4FFF,a5@(520) ");
asm(" moveml a4@(572),#0x4FFF "); /* move 52 bytes */
asm(" moveml #0x4FFF,a5@(572) ");
asm(" moveml a4@(624),#0x4FFF "); /* move 52 bytes */
asm(" moveml #0x4FFF,a5@(624) ");
asm(" moveml a4@(676),#0x4FFF "); /* move 52 bytes */
asm(" moveml #0x4FFF,a5@(676) ");
asm(" moveml a4@(728),#0x4FFF "); /* move 52 bytes */
asm(" moveml #0x4FFF,a5@(728) ");
asm(" moveml a4@(780),#0x7F "); /* move 28 more bytes */
asm(" moveml #0x7F,a5@(780) ");
asm(" movm a4@(808),a5@(808) "); /* move last 2 bytes */
asm(" moveml sp@+,#0xFFFF ");

#ifdef lint
if (dest)
    return;
else if (src)
    return;
#endif
}
#endif FAST

#endif
bmbank(dl) /* blank line dl */
register dl;
{
    register char *dest;
    register short i;

    dest = bmscrn + (dl * (BPL * V_RESO));
    i = BPL * V_RESO;
    do {
        *dest = bmnorm;
        dest++; /* 1 scan line */
    } while (--i != 0);
}

#else FAST
bmbank(dl) /* blank line dl */
register dl;
{
    register char *dest, *a4; /* NOTUSED */
    extern long bblank[];

    dest = bmscrn + (dl * (BPL * V_RESO)); /* register a5 */
}

```

