

```
/* A small utility to append an a.out header to an arbitrary file. This allows
 * inclusion of arbitrary data in the boot image, so that it is magically
 * loaded as a RAM disk. The a.out header is structured as follows:
 *
 *      a_flags:          A_IMG to indicate this is not an executable
 *
 * Created:      April 2005, Jorrit N. Herder
 */
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <a.out.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include <unistd.h>

#define INPUT_FILE      1
#define OUTPUT_FILE     2

/* Report problems. */
void report(char *problem, char *message)
{
    fprintf(stderr, "%s:\n", problem);
    fprintf(stderr, "  %s\n\n", message);
}

int copy_data(int srcfd, int dstfd)
{
    char buf[8192];
    ssize_t n;
    int total=0;

    /* Copy the little bytes themselves. (Source from cp.c). */
    while ((n= read(srcfd, buf, sizeof(buf))) > 0) {
        char *bp = buf;
        ssize_t r;

        while (n > 0 && (r= write(dstfd, bp, n)) > 0) {
            bp += r;
            n -= r;
            total += r;
        }
        if (r <= 0) {
            if (r == 0) {
                fprintf(stderr, "Warning: EOF writing to output file.\n");
                return(-1);
            }
        }
    }
    return(total);
}

/* Main program. */
int main(int argc, char **argv)
{
    struct exec aout;
    struct stat stin;
    int fdin, fdout;
    char * bp;
    int n,r;
    int total_size=0;
    int result;

    /* Check if command line arguments are present, or print usage. */
    if (argc!=3) {
        printf("Invalid arguments. Usage:\n");
        printf("  %s <input_file> <output_file>\n", argv[0]);
        return(1);
    }
}
```

```
/* Check if we can open the input and output file. */
if (stat(argv[INPUT_FILE], &stin) != 0) {
    report("Couldn't get status of input file", strerror(errno));
    return(1);
}
if ((fdin = open(argv[INPUT_FILE], O_RDONLY)) < 0) {
    report("Couldn't open input file", strerror(errno));
    return(1);
}
if ((fdout = open(argv[OUTPUT_FILE], O_WRONLY|O_CREAT|O_TRUNC,
    stin.st_mode & 0777)) < 0) {
    report("Couldn't open output file", strerror(errno));
    return(1);
}

/* Copy input file to output file, but leave space for a.out header. */
lseek(fdout, sizeof(aout), SEEK_SET);
total_size = copy_data(fdin, fdout);
if (total_size < 0) {
    report("Aborted", "Output file may be truncated.");
    return(1);
} else if (total_size == 0) {
    report("Aborted without prepending header", "No data in input file.");
    return(1);
}

/* Build a.out header and write to output file. */
memset(&aout, 0, sizeof(struct exec));
aout.a_magic[0] = A_MAGIC0;
aout.a_magic[1] = A_MAGIC1;
aout.a_flags |= A_IMG;
aout.a_hdrlen = sizeof(aout);
aout.a_text = 0;
aout.a_data = total_size;
aout.a_bss = 0;
aout.a_total = aout.a_hdrlen + aout.a_data;

bp = (char *) &aout;
n = sizeof(aout);
lseek(fdout, 0, SEEK_SET);
while (n > 0 && (r = write(fdout, bp, n)) > 0) {
    bp += r;
    n -= r;
}

printf("Prepended data file (%u bytes) with a.out header (%u bytes).\n",
    total_size, sizeof(aout));
printf("Done.\n");

return(0);
}
```

```
#!/bin/sh
#
#      a.out2com - Minix a.out to DOS .COM      Author: Kees J. Bot
#                                              17 Jun 1995
# Transform a Minix a.out to the COM format of MS-DOS,
# the executable must be common I&D with 256 scratch bytes at the start of
# the text segment to make space for the Program Segment Prefix. The Minix
# a.out header and these 256 bytes are removed to make a COM file.

case $# in
2)      aout="$1"
        com="$2"
        ;;
*)      echo "Usage: $0 <a.out> <dos.com>" >&2
        exit 1
esac

size "$aout" >/dev/null || exit
set `size "$aout" | sed ld`
count=`expr \( $1 + $2 - 256 + 31 \) / 32`

exec dd if="$aout" of="$com" bs=32 skip=9 count=$count conv=silent

#
# $PchId: a.out2com,v 1.3 1998/08/01 09:13:01 philip Exp $
```

```

!      Bootblock 1.5 - Minix boot block.                                Author: Kees J. Bot
!                                                                           21 Dec 1991
!
! When the PC is powered on, it will try to read the first sector of floppy
! disk 0 at address 0x7C00. If this fails due to the absence of flexible
! magnetic media, it will read the master boot record from the first sector
! of the hard disk. This sector not only contains executable code, but also
! the partition table of the hard disk. When executed, it will select the
! active partition and load the first sector of that at address 0x7C00.
! This file contains the code that is eventually read from either the floppy
! disk, or the hard disk partition. It is just smart enough to load /boot
! from the boot device into memory at address 0x10000 and execute that. The
! disk addresses for /boot are patched into this code by installboot as 24-bit
! sector numbers and 8-bit sector counts above enddata upwards. /boot is in
! turn smart enough to load the different parts of the Minix kernel into
! memory and execute them to finally get Minix started.
!

        LOADOFF    =    0x7C00    ! 0x0000:LOADOFF is where this code is loaded
        BOOTSEG    =    0x1000    ! Secondary boot code segment.
        BOOTOFF    =    0x0030    ! Offset into /boot above header
        BUFFER     =    0x0600    ! First free memory
        LOWSEC     =          8    ! Offset of logical first sector in partition
                                   ! table

        ! Variables addressed using bp register
        device     =          0    ! The boot device
        lowsec     =          2    ! Offset of boot partition within drive
        secpcyl    =          6    ! Sectors per cylinder = heads * sectors

.text

! Start boot procedure.

boot:
        xor        ax, ax          ! ax = 0x0000, the vector segment
        mov        ds, ax
        cli        ! Ignore interrupts while setting stack
        mov        ss, ax          ! ss = ds = vector segment
        mov        sp, #LOADOFF    ! Usual place for a bootstrap stack
        sti

        push       ax
        push       ax              ! Push a zero lowsec(bp)

        push       dx              ! Boot device in dl will be device(bp)
        mov        bp, sp          ! Using var(bp) is one byte cheaper then var.

        push       es
        push       si              ! es:si = partition table entry if hard disk

        mov        di, #LOADOFF+sectors    ! char *di = sectors;

        testb     dl, dl           ! Winchester disks if dl ≥ 0x80
        jge        floppy

winchester:

! Get the offset of the first sector of the boot partition from the partition
! table. The table is found at es:si, the lowsec parameter at offset LOWSEC.

        eseg
        les        ax, LOWSEC(si)  ! es:ax = LOWSEC+2(si):LOWSEC(si)
        mov        lowsec+0(bp), ax ! Low 16 bits of partition's first sector
        mov        lowsec+2(bp), es ! High 16 bits of partition's first sector

! Get the drive parameters, the number of sectors is bluntly written into the
! floppy disk sectors/track array.

        movb       ah, #0x08       ! Code for drive parameters
        int        0x13            ! dl still contains drive
        andb       cl, #0x3F       ! cl = max sector number (1-origin)
        movb       (di), cl        ! Number of sectors per track
        incb       dh              ! dh = 1 + max head number (0-origin)

```

```

        jmp     loadboot

! Floppy:
! Execute three read tests to determine the drive type.  Test for each floppy
! type by reading the last sector on the first track.  If it fails, try a type
! that has less sectors.  Therefore we start with 1.44M (18 sectors) then 1.2M
! (15 sectors) ending with 720K/360K (both 9 sectors).

next:   inc     di                ! Next number of sectors per track

floppy: xorb    ah, ah            ! Reset drive
        int     0x13

        movb    cl, (di)         ! cl = number of last sector on track

        cmpb    cl, #9           ! No need to do the last 720K/360K test
        je      success

! Try to read the last sector on track 0

        mov     es, lowsec(bp)    ! es = vector segment (lowsec = 0)
        mov     bx, #BUFFER      ! es:bx buffer = 0x0000:0x0600
        mov     ax, #0x0201      ! Read sector, #sectors = 1
        xorb    ch, ch           ! Track 0, last sector
        xorb    dh, dh           ! Drive dl, head 0
        int     0x13
        jc      next            ! Error, try the next floppy type

success: movb    dh, #2           ! Load number of heads for multiply

loadboot:
! Load /boot from the boot device

        movb    al, (di)         ! al = (di) = sectors per track
        mulb    dh               ! dh = heads, ax = heads * sectors
        mov     secpcyl(bp), ax  ! Sectors per cylinder = heads * sectors

        mov     ax, #BOOTSEG     ! Segment to load /boot into
        mov     es, ax
        xor     bx, bx           ! Load first sector at es:bx = BOOTSEG:0x0000
        mov     si, #LOADOFF+addresses ! Start of the boot code addresses

load:
        mov     ax, 1(si)        ! Get next sector number: low 16 bits
        movb    dl, 3(si)        ! Bits 16-23 for your up to 8GB partition
        xorb    dh, dh           ! dx:ax = sector within partition
        add     ax, lowsec+0(bp)
        adc     dx, lowsec+2(bp) ! dx:ax = sector within drive
        cmp     dx, #[1024*255*63-255]>>16 ! Near 8G limit?
        jae     bigdisk
        div     secpcyl(bp)      ! ax = cylinder, dx = sector within cylinder
        xchg    ax, dx           ! ax = sector within cylinder, dx = cylinder
        movb    ch, dl           ! ch = low 8 bits of cylinder
        divb    (di)             ! al = head, ah = sector (0-origin)
        xorb    dl, dl           ! About to shift bits 8-9 of cylinder into dl
        shr     dx, #1
        shr     dx, #1           ! dl[6..7] = high cylinder
        orb     dl, ah           ! dl[0..5] = sector (0-origin)
        movb    cl, dl           ! cl[0..5] = sector, cl[6..7] = high cyl
        incb    cl               ! cl[0..5] = sector (1-origin)
        movb    dh, al           ! dh = al = head
        movb    dl, device(bp)   ! dl = device to read
        movb    al, (di)         ! Sectors per track - Sector number (0-origin)
        subb    al, ah           ! = Sectors left on this track
        cmpb    al, (si)         ! Compare with # sectors to read
        jbe     read            ! Can't read past the end of a cylinder?
        movb    al, (si)         ! (si) < sectors left on this track
read:   push     ax              ! Save al = sectors to read
        movb    ah, #0x02        ! Code for disk read (all registers in use now!)
        int     0x13             ! Call the BIOS for a read
        pop     cx               ! Restore al in cl
        jmp     rdeval

bigdisk:
        movb    cl, (si)         ! Number of sectors to read
        push     si              ! Save si

```

```

    mov     si, #LOADOFF+ext_rw ! si = extended read/write parameter packet
    movb    2(si), cl           ! Fill in # blocks to transfer
    mov     4(si), bx           ! Buffer address
    mov     8(si), ax           ! Starting block number = dx:ax
    mov     10(si), dx
    movb    dl, device(bp)     ! dl = device to read
    movb    ah, #0x42          ! Extended read
    int     0x13
    pop     si                  ! Restore si to point to the addresses array
    ! jmp    rdeval

rdeval:
    jc      error              ! Jump on disk read error
    movb    al, cl             ! Restore al = sectors read
    addb    bh, al             ! bx += 2 * al * 256 (add bytes read)
    addb    bh, al             ! es:bx = where next sector must be read
    add     1(si), ax          ! Update address by sectors read
    adcb    3(si), ah          ! Don't forget bits 16-23 (add ah = 0)
    subb    (si), al           ! Decrement sector count by sectors read
    jnz     load               ! Not all sectors have been read
    add     si, #4              ! Next (address, count) pair
    cmpb    ah, (si)           ! Done when no sectors to read
    jnz     load               ! Read next chunk of /boot

done:

! Call /boot, assuming a long a.out header (48 bytes). The a.out header is
! usually short (32 bytes), but to be sure /boot has two entry points:
! One at offset 0 for the long, and one at offset 16 for the short header.
! Parameters passed in registers are:
!
!     dl      = Boot-device.
!     es:si    = Partition table entry if hard disk.
!
    pop     si                  ! Restore es:si = partition table entry
    pop     es                  ! dl is still loaded
    jmpf    BOOTOFF, BOOTSEG    ! jmp to sec. boot (skipping header).

! Read error: print message, hang forever
error:
    mov     si, #LOADOFF+errno+1
prnum:    movb    al, ah         ! Error number in ah
    andb    al, #0x0F          ! Low 4 bits
    cmpb    al, #10            ! A-F?
    jb      digit              ! 0-9!
    addb    al, #7              ! 'A' - ':'
digit:    addb    (si), al       ! Modify '0' in string
    dec     si
    movb    cl, #4              ! Next 4 bits
    shrb    ah, cl
    jnz     prnum              ! Again if digit > 0

    mov     si, #LOADOFF+rderr  ! String to print
print:    lodsb                 ! al = *si++ is char to be printed
    testb   al, al              ! Null byte marks end
hang:     jz      hang           ! Hang forever waiting for CTRL-ALT-DEL
    movb    ah, #0x0E          ! Print character in teletype mode
    mov     bx, #0x0001         ! Page 0, foreground color
    int     0x10                ! Call BIOS VIDEO_IO
    jmp     print

.data
rderr:    .ascii  "Read error "
errno:    .ascii  "00\0"
errend:

! Floppy disk sectors per track for the 1.44M, 1.2M and 360K/720K types:
sectors:
    .data1  18, 15, 9

! Extended read/write commands require a parameter packet.
ext_rw:
    .data1  0x10                ! Length of extended r/w packet
    .data1  0                  ! Reserved
    .data2  0                  ! Blocks to transfer (to be filled in)

```

```
.data2 0          ! Buffer address offset (tbfi)
.data2 BOOTSEG    ! Buffer address segment
.data4 0          ! Starting block number low 32 bits (tbfi)
.data4 0          ! Starting block number high 32 bits

.align 2
```

addresses:

! The space below this is **for** disk addresses **for** a 38K /boot program (worst case, i.e. file is completely fragmented). It should be enough.

```

/*      boot.c - Load and start Minix.                                Author: Kees J. Bot
*                                                              27 Dec 1991
*/

char version[] = "2.20";

#define BIOS      (!UNIX)      /* Either uses BIOS or UNIX syscalls. */

#define nil 0
#define _POSIX_SOURCE 1
#define _MINIX 1
#include <stddef.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <string.h>
#include <errno.h>
#include <ibm/partition.h>
#include <minix/config.h>
#include <minix/type.h>
#include <minix/com.h>
#include <minix/dmap.h>
#include <minix/const.h>
#include <minix/minlib.h>
#include <minix/syslib.h>
#if BIOS
#include <kernel/const.h>
#include <kernel/type.h>
#endif
#if UNIX
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <termios.h>
#endif
#include "rawfs.h"
#undef EXTERN
#define EXTERN /* Empty */
#include "boot.h"

#define arraysize(a)      (sizeof(a) / sizeof((a)[0]))
#define arraylimit(a)     ((a) + arraysize(a))
#define between(a, c, z)  ((unsigned) ((c) - (a)) <= ((z) - (a)))

int fsok = -1;      /* File system state.  Initially unknown. */

static int block_size;

#if BIOS

/* this data is reserved for BIOS int 0x13 to put the 'specification packet'
 * in. It has a structure of course, but we don't define a struct because
 * of compiler padding. We fiddle out the bytes ourselves later.
 */
unsigned char boot_spec[24];

char *bios_err(int err)
/* Translate BIOS error code to a readable string. (This is a rare trait
 * known as error checking and reporting. Take a good look at it, you won't
 * see it often.)
 */
{
    static struct errlist {
        int      err;
        char     *what;
    } errlist[] = {
#if !DOS
        { 0x00, "No error" },
        { 0x01, "Invalid command" },
        { 0x02, "Address mark not found" },

```



```

        { 0x03, "Disk write-protected" },
        { 0x04, "Sector not found" },
        { 0x05, "Reset failed" },
        { 0x06, "Floppy disk removed" },
        { 0x07, "Bad parameter table" },
        { 0x08, "DMA overrun" },
        { 0x09, "DMA crossed 64 KB boundary" },
        { 0x0A, "Bad sector flag" },
        { 0x0B, "Bad track flag" },
        { 0x0C, "Media type not found" },
        { 0x0D, "Invalid number of sectors on format" },
        { 0x0E, "Control data address mark detected" },
        { 0x0F, "DMA arbitration level out of range" },
        { 0x10, "Uncorrectable CRC or ECC data error" },
        { 0x11, "ECC corrected data error" },
        { 0x20, "Controller failed" },
        { 0x40, "Seek failed" },
        { 0x80, "Disk timed-out" },
        { 0xAA, "Drive not ready" },
        { 0xBB, "Undefined error" },
        { 0xCC, "Write fault" },
        { 0xE0, "Status register error" },
        { 0xFF, "Sense operation failed" }
    }
#else /* DOS */
    { 0x00, "No error" },
    { 0x01, "Function number invalid" },
    { 0x02, "File not found" },
    { 0x03, "Path not found" },
    { 0x04, "Too many open files" },
    { 0x05, "Access denied" },
    { 0x06, "Invalid handle" },
    { 0x0C, "Access code invalid" },
#endif /* DOS */
};
struct errlist *errp;

for (errp= errlist; errp < arraylimit(errlist); errp++) {
    if (errp->err == err) return errp->what;
}
return "Unknown error";
}

char *unix_err(int err)
/* Translate the few errors rawfs can give. */
{
    switch (err) {
        case ENOENT: return "No such file or directory";
        case ENOTDIR: return "Not a directory";
        default: return "Unknown error";
    }
}

void rwerr(char *rw, off_t sec, int err)
{
    printf("\n%s error 0x%02x (%s) at sector %ld absolute\n",
        rw, err, bios_err(err), sec);
}

void readerr(off_t sec, int err) { rwerr("Read", sec, err); }
void writerr(off_t sec, int err) { rwerr("Write", sec, err); }

void readblock(off_t blk, char *buf, int block_size)
/* Read blocks for the rawfs package. */
{
    int r;
    u32_t sec= lowsec + blk * RATIO(block_size);

    if(!block_size) {
        printf("block_size 0\n");
        exit(1);
    }

    if ((r= readsectors(mon2abs(buf), sec, 1 * RATIO(block_size))) != 0) {
        readerr(sec, r); exit(1);
    }
}

```

```

    }
}

#define istty          (1)
#define alarm(n)       (0)

#endif /* BIOS */

#if UNIX

/* The Minix boot block must start with these bytes: */
char boot_magic[] = { 0x31, 0xC0, 0x8E, 0xD8, 0xFA, 0x8E, 0xD0, 0xBC };

struct biosdev {
    char *name;          /* Name of device. */
    int device;          /* Device to edit parameters. */
} bootdev;

struct termios termbuf;
int istty;

void quit(int status)
{
    if (istty) (void) tcsetattr(0, TCSANOW, &termbuf);
    exit(status);
}

#define exit(s) quit(s)

void report(char *label)
/* edparams: label: No such file or directory */
{
    fprintf(stderr, "edparams: %s: %s\n", label, strerror(errno));
}

void fatal(char *label)
{
    report(label);
    exit(1);
}

void *alloc(void *m, size_t n)
{
    m = m == nil ? malloc(n) : realloc(m, n);
    if (m == nil) fatal("");
    return m;
}

#define malloc(n)      alloc(nil, n)
#define realloc(m, n)  alloc(m, n)

#define mon2abs(addr)  ((void *) (addr))

int rwsectors(int rw, void *addr, u32_t sec, int nsec)
{
    ssize_t r;
    size_t len = nsec * SECTOR_SIZE;

    if (lseek(bootdev.device, sec * SECTOR_SIZE, SEEK_SET) == -1)
        return errno;

    if (rw == 0) {
        r = read(bootdev.device, (char *) addr, len);
    } else {
        r = write(bootdev.device, (char *) addr, len);
    }
    if (r == -1) return errno;
    if (r != len) return EIO;
    return 0;
}

#define readsectors(a, s, n)    rwsectors(0, (a), (s), (n))
#define writesectors(a, s, n)  rwsectors(1, (a), (s), (n))
#define readerr(sec, err)      (errno = (err), report(bootdev.name))

```

```

#define writerr(sec, err)      (errno= (err), report(bootdev.name))
#define putch(c)              putchar(c)
#define unix_err(err)         strerror(err)

void readblock(off_t blk, char *buf, int block_size)
/* Read blocks for the rawfs package. */
{
    if(!block_size) fatal("block_size 0");
    errno= EIO;
    if (lseek(bootdev.device, blk * block_size, SEEK_SET) == -1
        || read(bootdev.device, buf, block_size) != block_size)
    {
        fatal(bootdev.name);
    }
}

sig_atomic_t trapsig;

void trap(int sig)
{
    trapsig= sig;
    signal(sig, trap);
}

int escape(void)
{
    if (trapsig == SIGINT) {
        trapsig= 0;
        return 1;
    }
    return 0;
}

static unsigned char unchar;

int getch(void)
{
    unsigned char c;

    fflush(stdout);

    if (unchar != 0) {
        c= unchar;
        unchar= 0;
        return c;
    }

    switch (read(0, &c, 1)) {
    case -1:
        if (errno != EINTR) fatal("");
        return(ESC);
    case 0:
        if (istty) putch('\n');
        exit(0);
    default:
        if (istty && c == termbuf.c_cc[VEOF]) {
            putch('\n');
            exit(0);
        }
        return c;
    }
}

#define ungetch(c)            ((void) (unchar = (c)))

#define get_tick()            ((u32_t) time(nil))
#define clear_screen()        printf("[clear]")
#define boot_device(device)   printf("[boot %s]\n", device)
#define ctty(line)            printf("[ctty %s]\n", line)
#define bootminix()            (run_trailer() && printf("[boot]\n"))
#define off()                  printf("[off]")

#endif /* UNIX */

```

```

char *getline(void)
/* Read a line including a newline with echoing. */
{
    char *line;
    size_t i, z;
    int c;

    i= 0;
    z= 20;
    line= malloc(z * sizeof(char));

    do {
        c= getch();

        if (strchr("\b\177\25\30", c) != nil) {
            /* Backspace, DEL, ctrl-U, or ctrl-X. */
            do {
                if (i == 0) break;
                printf("\b\b");
                i--;
            } while (c == '\25' || c == '\30');
        } else
        if (c < ' ' && c != '\n') {
            putchar('\7');
        } else {
            putchar(c);
            line[i++]= c;
            if (i == z) {
                z*= 2;
                line= realloc(line, z * sizeof(char));
            }
        }
    } while (c != '\n');
    line[i]= 0;
    return line;
}

int sugar(char *tok)
/* Recognize special tokens. */
{
    return strchr("=(){};\n", tok[0]) != nil;
}

char *onetoken(char **aline)
/* Returns a string with one token for tokenize. */
{
    char *line= *aline;
    size_t n;
    char *tok;

    /* Skip spaces and runs of newlines. */
    while (*line == ' ' || (*line == '\n' && line[1] == '\n')) line++;

    *aline= line;

    /* Don't do odd junk (nor the terminating 0!). */
    if ((unsigned) *line < ' ' && *line != '\n') return nil;

    if (*line == '(') {
        /* Function argument, anything goes but () must match. */
        int depth= 0;

        while ((unsigned) *line >= ' ') {
            if (*line == '(') depth++;
            if (*line++ == ') ' && --depth == 0) break;
        }
    } else
    if (sugar(line)) {
        /* Single character token. */
        line++;
    } else {
        /* Multicharacter token. */
        do line++; while ((unsigned) *line > ' ' && !sugar(line));
    }
}

```

```

    n= line - *aline;
    tok= malloc((n + 1) * sizeof(char));
    memcpy(tok, *aline, n);
    tok[n]= 0;
    if (tok[0] == '\n') tok[0]= ';';          /* ';' same as '\n' */

    *aline= line;
    return tok;
}

/* Typed commands form strings of tokens. */

typedef struct token {
    struct token    *next; /* Next in a command chain. */
    char            *token;
} token;

token **tokenize(token **acmds, char *line)
/* Takes a line apart to form tokens. The tokens are inserted into a command
 * chain at *acmds. Tokenize returns a reference to where another line could
 * be added. Tokenize looks at spaces as token separators, and recognizes only
 * ';', '=', '{', '}', and '\n' as single character tokens. One token is
 * formed from '(' and ')' with anything in between as long as more () match.
 */
{
    char *tok;
    token *newcmd;

    while ((tok= onetoken(&line)) != nil) {
        newcmd= malloc(sizeof(*newcmd));
        newcmd->token= tok;
        newcmd->next= *acmds;
        *acmds= newcmd;
        acmds= &newcmd->next;
    }
    return acmds;
}

token *cmds;          /* String of commands to execute. */
int err;              /* Set on an error. */

char *poptoken(void)
/* Pop one token off the command chain. */
{
    token *cmd= cmds;
    char *tok= cmd->token;

    cmds= cmd->next;
    free(cmd);

    return tok;
}

void voidtoken(void)
/* Remove one token from the command chain. */
{
    free(poptoken());
}

void parse_code(char *code)
/* Tokenize a string of monitor code, making sure there is a delimiter. It is
 * to be executed next. (Prepended to the current input.)
 */
{
    if (cmds != nil && cmds->token[0] != ';') (void) tokenize(&cmds, ";");
    (void) tokenize(&cmds, code);
}

int interrupt(void)
/* Clean up after an ESC has been typed. */
{
    if (escape()) {
        printf("[ESC]\n");
        err= 1;
    }
}

```

```

        return 1;
    }
    return 0;
}

#if BIOS

int activate;

struct biosdev {
    char name[8];
    int device, primary, secondary;
} bootdev, tmpdev;

int get_master(char *master, struct part_entry **table, u32_t pos)
/* Read a master boot sector and its partition table. */
{
    int r, n;
    struct part_entry *pe, **pt;

    if ((r= readsectors(mon2abs(master), pos, 1)) != 0) return r;

    pe= (struct part_entry *) (master + PART_TABLE_OFF);
    for (pt= table; pt < table + NR_PARTITIONS; pt++) *pt= pe++;

    /* DOS has the misguided idea that partition tables must be sorted. */
    if (pos != 0) return 0;          /* But only the primary. */

    n= NR_PARTITIONS;
    do {
        for (pt= table; pt < table + NR_PARTITIONS-1; pt++) {
            if (pt[0]->sysind == NO_PART
                || pt[0]->lowsec > pt[1]->lowsec) {
                pe= pt[0]; pt[0]= pt[1]; pt[1]= pe;
            }
        }
    } while (--n > 0);
    return 0;
}

void initialize(void)
{
    char master[SECTOR_SIZE];
    struct part_entry *table[NR_PARTITIONS];
    int r, p;
    u32_t masterpos;
    char *argp;

    /* Copy the boot program to the far end of low memory, this must be
     * done to get out of the way of Minix, and to put the data area
     * cleanly inside a 64K chunk if using BIOS I/O (no DMA problems).
     */
    u32_t oldaddr= caddr;
    u32_t memend= mem[0].base + mem[0].size;
    u32_t newaddr= (memend - runsize) & ~0x0000FL;

#if !DOS
    u32_t dma64k= (memend - 1) & ~0x0FFFFL;

    /* Check if data segment crosses a 64K boundary. */
    if (newaddr + (daddr - caddr) < dma64k) {
        newaddr= (dma64k - runsize) & ~0x0000FL;
    }

#endif

    /* Set the new caddr for relocate. */
    caddr= newaddr;

    /* Copy code and data. */
    raw_copy(newaddr, oldaddr, runsize);

    /* Make the copy running. */
    relocate();
}

```

```
#if !DOS
```

```

/* Take the monitor out of the memory map if we have memory to spare,
 * and also keep the BIOS data area safe (1.5K), plus a bit extra for
 * where we may have to put a.out headers for older kernels.
 */
if (mon_return = (mem[1].size > 512*1024L)) mem[0].size = newaddr;
mem[0].base += 2048;
mem[0].size -= 2048;

/* Find out what the boot device and partition was. */
bootdev.name[0]= 0;
bootdev.device= device;
bootdev.primary= -1;
bootdev.secondary= -1;

if (device < 0x80) {
    /* Floppy. */
    strcpy(bootdev.name, "fd0");
    bootdev.name[2] += bootdev.device;
    return;
}

/* Disk: Get the partition table from the very first sector, and
 * determine the partition we booted from using the information from
 * the booted partition entry as passed on by the bootstrap (rem_part).
 * All we need from it is the partition offset.
 */
raw_copy(mon2abs(&lowsec),
        vec2abs(&rem_part) + offsetof(struct part_entry, lowsec),
        sizeof(lowsec));

masterpos= 0; /* Master bootsector position. */

for (;;) {
    /* Extract the partition table from the master boot sector. */
    if ((r= get_master(master, table, masterpos)) != 0) {
        readerr(masterpos, r); exit(1);
    }

    /* See if you can find "lowsec" back. */
    for (p= 0; p < NR_PARTITIONS; p++) {
        if (lowsec - table[p]->lowsec < table[p]->size) break;
    }

    if (lowsec == table[p]->lowsec) { /* Found! */
        if (bootdev.primary < 0)
            bootdev.primary= p;
        else
            bootdev.secondary= p;
        break;
    }

    if (p == NR_PARTITIONS || bootdev.primary >= 0
        || table[p]->sysind != MINIX_PART) {
        /* The boot partition cannot be named, this only means
         * that "bootdev" doesn't work.
         */
        bootdev.device= -1;
        return;
    }

    /* See if the primary partition is subpartitioned. */
    bootdev.primary= p;
    masterpos= table[p]->lowsec;
}
strcpy(bootdev.name, "d0p0");
bootdev.name[1] += (device - 0x80);
bootdev.name[3] += bootdev.primary;
if (bootdev.secondary >= 0) {
    strcat(bootdev.name, "s0");
    bootdev.name[5] += bootdev.secondary;
}

```

```

#else /* DOS */
    /* Take the monitor out of the memory map if we have memory to spare,
     * note that only half our PSP is needed at the new place, the first
     * half is to be kept in its place.
     */
    if (mem[1].size > 0) mem[0].size = newaddr + 0x80 - mem[0].base;

    /* Parse the command line. */
    argp= PSP + 0x81;
    argp[PSP[0x80]]= 0;
    while (between('\l', *argp, ' ')) argp++;
    vdisk= argp;
    while (!between('\0', *argp, ' ')) argp++;
    while (between('\l', *argp, ' ')) *argp++= 0;
    if (*vdisk == 0) {
        printf("\nUsage: boot <vdisk> [commands ...]\n");
        exit(1);
    }
    drun= *argp == 0 ? "main" : argp;

    if ((r= dev_open()) != 0) {
        printf("\n%s: Error %02x (%s)\n", vdisk, r, bios_err(r));
        exit(1);
    }

    /* Find the active partition on the virtual disk. */
    if ((r= get_master(master, table, 0)) != 0) {
        readerr(0, r); exit(1);
    }

    strcpy(bootdev.name, "d0");
    bootdev.primary= -1;
    for (p= 0; p < NR_PARTITIONS; p++) {
        if (table[p]->bootind != 0 && table[p]->sysind == MINIX_PART) {
            bootdev.primary= p;
            strcat(bootdev.name, "p0");
            bootdev.name[3] += p;
            lowsec= table[p]->lowsec;
            break;
        }
    }
#endif /* DOS */
}

#endif /* BIOS */

/* Reserved names: */
enum resnames {
    R_NULL, R_BOOT, R_CTTY, R_DELAY, R_ECHO, R_EXIT, R_HELP,
    R_LS, R_MENU, R_OFF, R_SAVE, R_SET, R_TRAP, R_UNSET
};

char resnames[][6] = {
    "", "boot", "ctty", "delay", "echo", "exit", "help",
    "ls", "menu", "off", "save", "set", "trap", "unset",
};

/* Using this for all null strings saves a lot of memory. */
#define null (resnames[0])

int reserved(char *s)
/* Recognize reserved strings. */
{
    int r;

    for (r= R_BOOT; r <= R_UNSET; r++) {
        if (strcmp(s, resnames[r]) == 0) return r;
    }
    return R_NULL;
}

void sfree(char *s)
/* Free a non-null string. */
{

```



```

    if (s != nil && s != null) free(s);
}

char *copystr(char *s)
/* Copy a non-null string using malloc. */
{
    char *c;

    if (*s == 0) return null;
    c = malloc((strlen(s) + 1) * sizeof(char));
    strcpy(c, s);
    return c;
}

int is_default(environment *e)
{
    return (e->flags & E_SPECIAL) && e->defval == nil;
}

environment **searchenv(char *name)
{
    environment **aenv = &env;

    while (*aenv != nil && strcmp((*aenv)->name, name) != 0) {
        aenv = &(*aenv)->next;
    }

    return aenv;
}

#define b_getenv(name) (*searchenv(name))
/* Return the environment *structure* belonging to name, or nil if not found. */

char *b_value(char *name)
/* The value of a variable. */
{
    environment *e = b_getenv(name);

    return e == nil || !(e->flags & E_VAR) ? nil : e->value;
}

char *b_body(char *name)
/* The value of a function. */
{
    environment *e = b_getenv(name);

    return e == nil || !(e->flags & E_FUNCTION) ? nil : e->value;
}

int b_setenv(int flags, char *name, char *arg, char *value)
/* Change the value of an environment variable. Returns the flags of the
 * variable if you are not allowed to change it, 0 otherwise.
 */
{
    environment **aenv, *e;

    if ((*aenv = searchenv(name)) == nil) {
        if (reserved(name)) return E_RESERVED;
        e = malloc(sizeof(*e));
        e->name = copystr(name);
        e->flags = flags;
        e->defval = nil;
        e->next = nil;
        *aenv = e;
    } else {
        e = *aenv;

        /* Don't change special variables to functions or vv. */
        if (e->flags & E_SPECIAL
            && (e->flags & E_FUNCTION) != (flags & E_FUNCTION))
            return e->flags;

        e->flags = (e->flags & E_STICKY) | flags;
        if (is_default(e)) {

```

```

        e->defval= e->value;
    } else {
        sfree(e->value);
    }
    sfree(e->arg);
}
e->arg= copystr(arg);
e->value= copystr(value);

return 0;
}

int b_setvar(int flags, char *name, char *value)
/* Set variable or simple function. */
{
    int r;

    if((r=b_setenv(flags, name, null, value))) {
        return r;
    }

    return r;
}

void b_unset(char *name)
/* Remove a variable from the environment. A special variable is reset to
 * its default value.
 */
{
    environment **aenv, *e;

    if ((e= *(aenv= searchenv(name))) == nil) return;

    if (e->flags & E_SPECIAL) {
        if (e->defval != nil) {
            sfree(e->arg);
            e->arg= null;
            sfree(e->value);
            e->value= e->defval;
            e->defval= nil;
        }
    } else {
        sfree(e->name);
        sfree(e->arg);
        sfree(e->value);
        *aenv= e->next;
        free(e);
    }
}

long a2l(char *a)
/* Cheap atol(). */
{
    int sign= 1;
    long n= 0;

    if (*a == '-') { sign= -1; a++; }

    while (between('0', *a, '9')) n= n * 10 + (*a++ - '0');

    return sign * n;
}

char *ul2a(u32_t n, unsigned b)
/* Transform a long number to ascii at base b, (b >= 8). */
{
    static char num[(CHAR_BIT * sizeof(n) + 2) / 3 + 1];
    char *a= arraylimit(num) - 1;
    static char hex[16] = "0123456789ABCDEF";

    do *--a = hex[(int) (n % b)]; while ((n/= b) > 0);
    return a;
}

```

```

char *ul2a10(u32_t n)
/* Transform a long number to ascii at base 10. */
{
    return ul2a(n, 10);
}

unsigned a2x(char *a)
/* Ascii to hex. */
{
    unsigned n= 0;
    int c;

    for (;;) {
        c= *a;
        if (between('0', c, '9')) c= c - '0' + 0x0;
        else
        if (between('A', c, 'F')) c= c - 'A' + 0xA;
        else
        if (between('a', c, 'f')) c= c - 'a' + 0xa;
        else
            break;
        n= (n<<4) | c;
        a++;
    }
    return n;
}

void get_parameters(void)
{
    char params[SECTOR_SIZE + 1];
    token **acmds;
    int r, bus, processor;
    memory *mp;
    static char bus_type[][4] = {
        "xt", "at", "mca"
    };
    static char vid_type[][4] = {
        "mda", "cga", "ega", "ega", "vga", "vga"
    };
    static char vid_chrome[][6] = {
        "mono", "color"
    };

    /* Variables that Minix needs: */
    b_setvar(E_SPECIAL|E_VAR|E_DEV, "rootdev", "ram");
    b_setvar(E_SPECIAL|E_VAR|E_DEV, "ramimagedev", "bootdev");
    b_setvar(E_SPECIAL|E_VAR, "ramsize", "0");
#if BIOS
    processor = getprocessor();
    if(processor == 1586) processor = 686;
    b_setvar(E_SPECIAL|E_VAR, "processor", ul2a10(processor));
    b_setvar(E_SPECIAL|E_VAR, "bus", bus_type[get_bus()]);
    b_setvar(E_SPECIAL|E_VAR, "video", vid_type[get_video()]);
    b_setvar(E_SPECIAL|E_VAR, "chrome", vid_chrome[get_video() & 1]);
    params[0]= 0;
    for (mp= mem; mp < arraylimit(mem); mp++) {
        if (mp->size == 0) continue;
        if (params[0] != 0) strcat(params, ",");
        strcat(params, ul2a(mp->base, 0x10));
        strcat(params, ":");
        strcat(params, ul2a(mp->size, 0x10));
    }
    b_setvar(E_SPECIAL|E_VAR, "memory", params);
#endif
#if DOS
    b_setvar(E_SPECIAL|E_VAR, "dosfile-d0", vdisk);
#endif
#endif
#if UNIX
    b_setvar(E_SPECIAL|E_VAR, "processor", "?");
    b_setvar(E_SPECIAL|E_VAR, "bus", "?");
    b_setvar(E_SPECIAL|E_VAR, "video", "?");
    b_setvar(E_SPECIAL|E_VAR, "chrome", "?");

```

```

b_setvar(E_SPECIAL|E_VAR, "memory", "?");
b_setvar(E_SPECIAL|E_VAR, "c0", "?");
#endif

/* Variables boot needs: */
b_setvar(E_SPECIAL|E_VAR, "image", "boot/image");
b_setvar(E_SPECIAL|E_FUNCTION, "leader",
    "echo --- Welcome to MINIX 3. This is the boot monitor. ---\\n");
b_setvar(E_SPECIAL|E_FUNCTION, "main", "menu");
b_setvar(E_SPECIAL|E_FUNCTION, "trailer", "");

/* Default hidden menu function: */
b_setenv(E_RESERVED|E_FUNCTION, null, "=", "Start MINIX", "boot");

/* Tokenize bootparams sector. */
if ((r= readsectors(mon2abs(params), lowsec+PARAMSEC, 1)) != 0) {
    readerr(lowsec+PARAMSEC, r);
    exit(1);
}
params[SECTOR_SIZE]= 0;
acmds= tokenize(&cmds, params);

/* Stuff the default action into the command chain. */
#if UNIX
    (void) tokenize(acmds, "::");
#elif DOS
    (void) tokenize(tokenize(acmds, "::;leader;"), drun);
#else /* BIOS */
    (void) tokenize(acmds, "::;leader;main");
#endif
}

char *addptr;

void addparm(char *n)
{
    while (*n != 0 && *addptr != 0) *addptr++ = *n++;
}

void save_parameters(void)
/* Save nondefault environment variables to the bootparams sector. */
{
    environment *e;
    char params[SECTOR_SIZE + 1];
    int r;

    /* Default filling: */
    memset(params, '\\n', SECTOR_SIZE);

    /* Don't touch the 0! */
    params[SECTOR_SIZE]= 0;
    addptr= params;

    for (e= env; e != nil; e= e->next) {
        if (e->flags & E_RESERVED || is_default(e)) continue;

        addparm(e->name);
        if (e->flags & E_FUNCTION) {
            addparm("(");
            addparm(e->arg);
            addparm(")");
        } else {
            addparm((e->flags & (E_DEV|E_SPECIAL)) != E_DEV
                ? "=" : "=d ");
        }
        addparm(e->value);
        if (*addptr == 0) {
            printf("The environment is too big\\n");
            return;
        }
        *addptr++= '\\n';
    }

    /* Save the parameters on disk. */

```

```

        if ((r= writesectors(mon2abs(params), lowsec+PARAMSEC, 1)) != 0) {
            writerr(lowsec+PARAMSEC, r);
            printf("Can't save environment\n");
        }
    }

void show_env(void)
/* Show the environment settings. */
{
    environment *e;
    unsigned more= 0;
    int c;

    for (e= env; e != nil; e= e->next) {
        if (e->flags & E_RESERVED) continue;
        if (!istty && is_default(e)) continue;

        if (e->flags & E_FUNCTION) {
            printf("%s(%s) %s\n", e->name, e->arg, e->value);
        } else {
            printf(is_default(e) ? "%s=(%s)\n" : "%s=%s\n",
                e->name, e->value);
        }

        if (e->next != nil && istty && ++more % 20 == 0) {
            printf("More? ");
            c= getch();
            if (c == ESC || c > ' ') {
                putchar('\n');
                if (c > ' ') ungetch(c);
                break;
            }
            printf("\b\b\b\b\b\b\b\b");
        }
    }
}

int numprefix(char *s, char **ps)
/* True iff s is a string of digits. *ps will be set to the first nondigit
 * if non-nil, otherwise the string should end.
 */
{
    char *n= s;

    while (between('0', *n, '9')) n++;

    if (n == s) return 0;

    if (ps == nil) return *n == 0;

    *ps= n;
    return 1;
}

int numeric(char *s)
{
    return numprefix(s, (char **) nil);
}

#if BIOS

/* Device numbers of standard MINIX devices. */
#define DEV_FD0 0x0200
static dev_t dev_cNd0[] = { 0x0300, 0x0800, 0x0A00, 0x0C00, 0x1000 };
#define minor_p0s0 128

static int block_size;

dev_t name2dev(char *name)
/* Translate, say, /dev/c0d0p2 to a device number. If the name can't be
 * found on the boot device, then do some guesswork. The global structure
 * "tmpdev" will be filled in based on the name, so that "boot dlp0" knows
 * what device to boot without interpreting device numbers.
 */

```

```

{
    dev_t dev;
    ino_t ino;
    int drive;
    struct stat st;
    char *n, *s;

    /* "boot *d0p2" means: make partition 2 active before you boot it. */
    if ((activate= (name[0] == '*')) name++;

    /* The special name "bootdev" must be translated to the boot device. */
    if (strcmp(name, "bootdev") == 0) {
        if (bootdev.device == -1) {
            printf("The boot device could not be named\n");
            errno= 0;
            return -1;
        }
        name= bootdev.name;
    }

    /* If our boot device doesn't have a file system, or we want to know
     * what a name means for the BIOS, then we need to interpret the
     * device name ourselves: "fd" = floppy, "c0d0" = hard disk, etc.
     */
    tmpdev.device= tmpdev.primary= tmpdev.secondary= -1;
    dev= -1;
    n= name;
    if (strncmp(n, "/dev/", 5) == 0) n+= 5;

    if (strcmp(n, "ram") == 0) {
        dev= DEV_RAM;
    } else
    if (strcmp(n, "boot") == 0) {
        dev= DEV_BOOT;
    } else
    if (n[0] == 'f' && n[1] == 'd' && numeric(n+2)) {
        /* Floppy. */
        tmpdev.device= a2l(n+2);
        dev= DEV_FD0 + tmpdev.device;
    } else
    if ((n[0] == 'h' || n[0] == 's') && n[1] == 'd' && numprefix(n+2, &s)
        && (*s == 0 || (between('a', *s, 'd') && s[1] == 0))
    ) {
        /* Old style hard disk (backwards compatibility.) */
        dev= a2l(n+2);
        tmpdev.device= dev / (1 + NR_PARTITIONS);
        tmpdev.primary= (dev % (1 + NR_PARTITIONS)) - 1;
        if (*s != 0) {
            /* Subpartition. */
            tmpdev.secondary= *s - 'a';
            dev= minor_p0s0
                + (tmpdev.device * NR_PARTITIONS
                    + tmpdev.primary) * NR_PARTITIONS
                + tmpdev.secondary;
        }
        tmpdev.device+= 0x80;
        dev+= n[0] == 'h' ? dev_cNd0[0] : dev_cNd0[2];
    } else {
        /* Hard disk. */
        int ctrlr= 0;

        if (n[0] == 'c' && between('0', n[1], '4')) {
            ctrlr= (n[1] - '0');
            tmpdev.device= 0;
            n+= 2;
        }

        if (n[0] == 'd' && between('0', n[1], '7')) {
            tmpdev.device= (n[1] - '0');
            n+= 2;
            if (n[0] == 'p' && between('0', n[1], '3')) {
                tmpdev.primary= (n[1] - '0');
                n+= 2;
                if (n[0] == 's' && between('0', n[1], '3')) {
                    tmpdev.secondary= (n[1] - '0');
                }
            }
        }
    }
}

```

```

        n+= 2;
    }
}
}
if (*n == 0) {
    dev= dev_cNd0[ctrlr];
    if (tmpdev.secondary < 0) {
        dev += tmpdev.device * (NR_PARTITIONS+1)
            + (tmpdev.primary + 1);
    } else {
        dev += minor_p0s0
            + (tmpdev.device * NR_PARTITIONS
                + tmpdev.primary) * NR_PARTITIONS
            + tmpdev.secondary;
    }
    tmpdev.device+= 0x80;
}
}

/* Look the name up on the boot device for the UNIX device number. */
if (fsok == -1) fsok= r_super(&block_size) != 0;
if (fsok) {
    /* The current working directory is "/dev". */
    ino= r_lookup(r_lookup(ROOT_INO, "dev"), name);

    if (ino != 0) {
        /* Name has been found, extract the device number. */
        r_stat(ino, &st);
        if (!S_ISBLK(st.st_mode)) {
            printf("%s is not a block device\n", name);
            errno= 0;
            return (dev_t) -1;
        }
        dev= st.st_rdev;
    }
}

if (tmpdev.primary < 0) activate= 0;    /* Careful now! */

if (dev == -1) {
    printf("Can't recognize '%s' as a device\n", name);
    errno= 0;
}
return dev;
}

#ifdef DEBUG
static void apm_perror(char *label, u16_t ax)
{
    unsigned ah;
    char *str;

    ah= (ax >> 8);
    switch(ah)
    {
        case 0x01: str= "APM functionality disabled"; break;
        case 0x03: str= "interface not connected"; break;
        case 0x09: str= "unrecognized device ID"; break;
        case 0x0A: str= "parameter value out of range"; break;
        case 0x0B: str= "interface not engaged"; break;
        case 0x60: str= "unable to enter requested state"; break;
        case 0x86: str= "APM not present"; break;
        default: printf("%s: error 0x%02x\n", label, ah); return;
    }
    printf("%s: %s\n", label, str);
}

#define apm_printf printf
#else
#define apm_perror(label, ax) ((void)0)
#define apm_printf
#endif

static void off(void)

```

```

{
    bios_env_t be;
    unsigned al, ah;

    /* Try to switch off the system. Print diagnostic information
     * that can be useful if the operation fails.
     */

    be.ax= 0x5300; /* APM, Installation check */
    be.bx= 0;      /* Device, APM BIOS */
    int15(&be);
    if (be.flags & FL_CARRY)
    {
        apm_perror("APM installation check failed", be.ax);
        return;
    }
    if (be.bx != (('P' << 8) | 'M'))
    {
        apm_printf("APM signature not found (got 0x%04x)\n", be.bx);
        return;
    }

    ah= be.ax >> 8;
    if (ah > 9)
        ah= (ah >> 4)*10 + (ah & 0xf);
    al= be.ax & 0xff;
    if (al > 9)
        al= (al >> 4)*10 + (al & 0xf);
    apm_printf("APM version %u.%u%s%s%s%s\n",
        ah, al,
        (be.cx & 0x1) ? ", 16-bit PM" : "",
        (be.cx & 0x2) ? ", 32-bit PM" : "",
        (be.cx & 0x4) ? ", CPU-Idle" : "",
        (be.cx & 0x8) ? ", APM-disabled" : "",
        (be.cx & 0x10) ? ", APM-disengaged" : "");

    /* Connect */
    be.ax= 0x5301; /* APM, Real mode interface connect */
    be.bx= 0x0000; /* APM BIOS */
    int15(&be);
    if (be.flags & FL_CARRY)
    {
        apm_perror("APM real mode connect failed", be.ax);
        return;
    }

    /* Ask for a seat upgrade */
    be.ax= 0x530e; /* APM, Driver Version */
    be.bx= 0x0000; /* BIOS */
    be.cx= 0x0102; /* version 1.2 */
    int15(&be);
    if (be.flags & FL_CARRY)
    {
        apm_perror("Set driver version failed", be.ax);
        goto disco;
    }

    /* Is this version really worth reporting. Well, if the system
     * does switch off, you won't see it anyway.
     */
    ah= be.ax >> 8;
    if (ah > 9)
        ah= (ah >> 4)*10 + (ah & 0xf);
    al= be.ax & 0xff;
    if (al > 9)
        al= (al >> 4)*10 + (al & 0xf);
    apm_printf("Got APM connection version %u.%u\n", ah, al);

    /* Enable */
    be.ax= 0x5308; /* APM, Enable/disable power management */
    be.bx= 0x0001; /* All device managed by APM BIOS */

#if 0
    /* For old APM 1.0 systems, we need 0xffff. Assume that those
     * systems do not exist.

```



```

        */
        be.bx= 0xffff; /* All device managed by APM BIOS (compat) */
#endif
        be.cx= 0x0001; /* Enable power management */
        int15(&be);
        if (be.flags & FL_CARRY)
        {
            apm_perror("Enable power management failed", be.ax);
            goto disco;
        }

        /* Off */
        be.ax= 0x5307; /* APM, Set Power State */
        be.bx= 0x0001; /* All devices managed by APM */
        be.cx= 0x0003; /* Off */
        int15(&be);
        if (be.flags & FL_CARRY)
        {
            apm_perror("Set power state failed", be.ax);
            goto disco;
        }

        apm_printf("Power off sequence successfully completed.\n\n");
        apm_printf("Ha, ha, just kidding!\n");

disco:
        /* Disconnect */
        be.ax= 0x5304; /* APM, interface disconnect */
        be.bx= 0x0000; /* APM BIOS */
        int15(&be);
        if (be.flags & FL_CARRY)
        {
            apm_perror("APM interface disconnect failed", be.ax);
            return;
        }
    }

#ifdef !DOS
#define B_NOSIG          -1          /* "No signature" error code. */

int exec_bootstrap(void)
/* Load boot sector from the disk or floppy described by tmpdev and execute it.
   */
{
    int r, n, dirty= 0;
    char master[SECTOR_SIZE];
    struct part_entry *table[NR_PARTITIONS], dummy, *active= &dummy;
    u32_t masterpos;

    active->lowsec= 0;

    /* Select a partition table entry. */
    while (tmpdev.primary >= 0) {
        masterpos= active->lowsec;

        if ((r= get_master(master, table, masterpos)) != 0) return r;

        active= table[tmpdev.primary];

        /* How does one check a partition table entry? */
        if (active->sysind == NO_PART) return B_NOSIG;

        tmpdev.primary= tmpdev.secondary;
        tmpdev.secondary= -1;
    }

    if (activate && !active->bootind) {
        for (n= 0; n < NR_PARTITIONS; n++) table[n]->bootind= 0;
        active->bootind= ACTIVE_FLAG;
        dirty= 1;
    }

    /* Read the boot sector. */
    if ((r= readsectors(BOOTPOS, active->lowsec, 1)) != 0) return r;

```

```

    /* Check signature word. */
    if (get_word(BOOTPOS+SIGNATOFF) != SIGNATURE) return B_NOSIG;

    /* Write the partition table if a member must be made active. */
    if (dirty && (r= writesectors(mon2abs(master), masterpos, 1)) != 0)
        return r;

    bootstrap(device, active);
}

void boot_device(char *devname)
/* Boot the device named by devname. */
{
    dev_t dev= name2dev(devname);
    int save_dev= device;
    int r;
    char *err;

    if (tmpdev.device < 0) {
        if (dev != -1) printf("Can't boot from %s\n", devname);
        return;
    }

    /* Change current device and try to load and execute its bootstrap. */
    device= tmpdev.device;

    if ((r= dev_open()) == 0) r= exec_bootstrap();

    err= r == B_NOSIG ? "Not bootable" : bios_err(r);
    printf("Can't boot %s: %s\n", devname, err);

    /* Restore boot device setting. */
    device= save_dev;
    (void) dev_open();
}

void cty(char *line)
{
    if (between('0', line[0], '3') && line[1] == 0) {
        serial_init(line[0] - '0');
    } else {
        printf("Bad serial line number: %s\n", line);
    }
}

#else /* DOS */

void boot_device(char *devname)
/* No booting of other devices under DOS. */
{
    printf("Can't boot devices under DOS\n");
}

void cty(char *line)
/* Don't know how to handle serial lines under DOS. */
{
    printf("No serial line support under DOS\n");
}

#endif /* DOS */
#endif /* BIOS */

void ls(char *dir)
/* List the contents of a directory. */
{
    ino_t ino;
    struct stat st;
    char name[NAME_MAX+1];

    if (fsok == -1) fsok= r_super(&block_size) != 0;
    if (!fsok) return;

    /* (,) construct because r_stat returns void */

```

```

    if ((ino= r_lookup(ROOT_INO, dir)) == 0 ||
        (r_stat(ino, &st), r_readdir(name)) == -1)
    {
        printf("ls: %s: %s\n", dir, unix_err(errno));
        return;
    }
    (void) r_readdir(name); /* Skip "." too. */

    while ((ino= r_readdir(name)) != 0) printf("%s/%s\n", dir, name);
}

u32_t milli_time(void)
{
    return get_tick() * MSEC_PER_TICK;
}

u32_t milli_since(u32_t base)
{
    return (milli_time() + (TICKS_PER_DAY*MSEC_PER_TICK) - base)
           % (TICKS_PER_DAY*MSEC_PER_TICK);
}

char *Thandler;
u32_t Tbase, Tcount;

void unschedule(void)
/* Invalidate a waiting command. */
{
    alarm(0);

    if (Thandler != nil) {
        free(Thandler);
        Thandler= nil;
    }
}

void schedule(long msec, char *cmd)
/* Schedule command at a certain time from now. */
{
    unschedule();
    Thandler= cmd;
    Tbase= milli_time();
    Tcount= msec;
    alarm(1);
}

int expired(void)
/* Check if the timer expired for getch(). */
{
    return (Thandler != nil && milli_since(Tbase) >= Tcount);
}

void delay(char *msec)
/* Delay for a given time. */
{
    u32_t base, count;

    if ((count= a2l(msec)) == 0) return;
    base= milli_time();

    alarm(1);

    do {
        pause();
    } while (!interrupt() && !expired() && milli_since(base) < count);
}

enum whatfun { NOFUN, SELECT, DEFFUN, USERFUN } menufun(environment *e)
{
    if (!(e->flags & E_FUNCTION) || e->arg[0] == 0) return NOFUN;
    if (e->arg[1] != ',') return SELECT;
    return e->flags & E_RESERVED ? DEFFUN : USERFUN;
}

```

```

void menu(void)
/* By default: Show a simple menu.
 * Multiple kernels/images: Show extra selection options.
 * User defined function: Kill the defaults and show these.
 * Wait for a keypress and execute the given function.
 */
{
    int c, def= 1;
    char *choice= nil;
    environment *e;

    /* Just a default menu? */
    for (e= env; e != nil; e= e->next) if (menufun(e) == USERFUN) def= 0;

    printf("\nHit a key as follows:\n\n");

    /* Show the choices. */
    for (e= env; e != nil; e= e->next) {
        switch (menufun(e)) {
            case DEFFUN:
                if (!def) break;
                /*FALL THROUGH*/
            case USERFUN:
                printf("  %c %s\n", e->arg[0], e->arg+2);
                break;
            case SELECT:
                printf("  %c Select %s kernel\n", e->arg[0], e->name);
                break;
            default:;
        }
    }

    /* Wait for a keypress. */
    do {
        c= getch();
        if (interrupt() || expired()) return;

        unschedule();

        for (e= env; e != nil; e= e->next) {
            switch (menufun(e)) {
                case DEFFUN:
                    if (!def) break;
                case USERFUN:
                case SELECT:
                    if (c == e->arg[0]) choice= e->value;
            }
        }
    } while (choice == nil);

    /* Execute the chosen function. */
    printf("%c\n", c);
    (void) tokenize(&cmds, choice);
}

void help(void)
/* Not everyone is a rocket scientist. */
{
    struct help {
        char    *thing;
        char    *help;
    } *pi;
    static struct help info[] = {
        { nil, "Names:" },
        { "rootdev", "Root device" },
        { "ramimagedev", "Device to use as RAM disk image" },
        { "ramsize", "RAM disk size (if no image device)" },
        { "bootdev", "Special name for the boot device" },
        { "fd0, d0p2, c0d0p1s0", "Devices (as in /dev)" },
        { "image", "Name of the boot image to use" },
        { "main", "Startup function" },
        { "bootdelay", "Delay in msec after loading image" },
        { nil, "Commands:" },
        { "name = [device] value", "Set environment variable" },
    }
}

```

```

        { "name() { ... }", "Define function" },
        { "name(key,text) { ... }",
          "A menu option like: minix(=,Start MINIX) {boot}" },
        { "name", "Call function" },
        { "boot [device]", "Boot Minix or another O.S." },
        { "ctty [line]", "Duplicate to serial line" },
        { "delay [msec]", "Delay (500 msec default)" },
        { "echo word ...", "Display the words" },
        { "ls [directory]", "List contents of directory" },
        { "menu", "Show menu and choose menu option" },
        { "save / set", "Save or show environment" },
        { "trap msec command", "Schedule command" },
        { "unset name ...", "Unset variable or set to default" },
        { "exit / off", "Exit the Monitor / Power off" },
    };

    for (pi= info; pi < arraylimit(info); pi++) {
        if (pi->thing != nil) printf(" %-24s- ", pi->thing);
        printf("%s\n", pi->help);
    }
}

void execute(void)
/* Get one command from the command chain and execute it. */
{
    token *second, *third, *fourth, *sep;
    char *name;
    int res;
    size_t n= 0;

    if (err) {
        /* An error occurred, stop interpreting. */
        while (cmds != nil) voidtoken();
        return;
    }

    if (expired()) { /* Timer expired? */
        parse_code(Thandler);
        unschedule();
    }

    /* There must be a separator lurking somewhere. */
    for (sep= cmds; sep != nil && sep->token[0] != ';'; sep= sep->next) n++;

    name= cmds->token;
    res= reserved(name);
    if ((second= cmds->next) != nil
        && (third= second->next) != nil)
        fourth= third->next;

    /* Null command? */
    if (n == 0) {
        voidtoken();
        return;
    } else
        /* name = [device] value? */
        if ((n == 3 || n == 4)
            && !sugar(name)
            && second->token[0] == '='
            && !sugar(third->token)
            && (n == 3 || (n == 4 && third->token[0] == 'd'
                          && !sugar(fourth->token)))
        ) {
            char *value= third->token;
            int flags= E_VAR;

            if (n == 4) { value= fourth->token; flags|= E_DEV; }

            if ((flags= b_setvar(flags, name, value)) != 0) {
                printf("%s is a %s\n", name,
                    flags & E_RESERVED ? "reserved word" :
                    "special function");
                err= 1;
            }
        }
}

```

```

        while (cmds != sep) voidtoken();
        return;
    } else
        /* name '(arg)' ... ? */
    if (n >= 3
        && !sugar(name)
        && second->token[0] == '('
    ) {
        token *fun;
        int c, flags, depth;
        char *body;
        size_t len;

        sep= fun= third;
        depth= 0;
        len= 1;
        while (sep != nil) {
            if ((c= sep->token[0]) == ';' && depth == 0) break;
            len+= strlen(sep->token) + 1;
            sep= sep->next;
            if (c == '{') depth++;
            if (c == '}' && --depth == 0) break;
        }

        body= malloc(len * sizeof(char));
        *body= 0;

        while (fun != sep) {
            strcat(body, fun->token);
            if (!sugar(fun->token)
                && !sugar(fun->next->token)
            ) strcat(body, " ");
            fun= fun->next;
        }
        second->token[strlen(second->token)-1]= 0;

        if (depth != 0) {
            printf("Missing '}'\n");
            err= 1;
        } else
            if ((flags= b_setenv(E_FUNCTION, name,
                                second->token+1, body)) != 0) {
                printf("%s is a %s\n", name,
                    flags & E_RESERVED ? "reserved word" :
                    "special variable");
                err= 1;
            }
        while (cmds != sep) voidtoken();
        free(body);
        return;
    } else
        /* Grouping? */
    if (name[0] == '{') {
        token **acmds= &cmds->next;
        char *t;
        int depth= 1;

        /* Find and remove matching '}' */
        depth= 1;
        while (*acmds != nil) {
            t= (*acmds)->token;
            if (t[0] == '{') depth++;
            if (t[0] == '}' && --depth == 0) { t[0]= ';'; break; }
            acmds= &(*acmds)->next;
        }
        voidtoken();
        return;
    } else
        /* Command coming up, check if ESC typed. */
    if (interrupt()) {
        return;
    } else
        /* unset name ..., echo word ...? */
    if (n >= 1 && (res == R_UNSET || res == R_ECHO)) {

```

```

char *arg= poptoken(), *p;

for (;;) {
    free(arg);
    if (cmds == sep) break;
    arg= poptoken();
    if (res == R_UNSET) { /* unset arg */
        b_unset(arg);
    } else { /* echo arg */
        p= arg;
        while (*p != 0) {
            if (*p != '\\') {
                putchar(*p);
            } else
            switch (*++p) {
                case 0:
                    if (cmds == sep) return;
                    continue;
                case 'n':
                    putchar('\\n');
                    break;
                case 'v':
                    printf(version);
                    break;
                case 'c':
                    clear_screen();
                    break;
                case 'w':
                    for (;;) {
                        if (interrupt())
                            return;
                        if (getch() == '\\n')
                            break;
                    }
                    break;
                default:
                    putchar(*p);
            }
            p++;
        }
        putchar(cmds != sep ? ' ' : '\\n');
    }
}
return;
} else
    /* boot -opts? */
if (n == 2 && res == R_BOOT && second->token[0] == '-') {
    static char optsvar[] = "bootopts";
    (void) b_setvar(E_VAR, optsvar, second->token);
    voidtoken();
    voidtoken();
    bootminix();
    b_unset(optsvar);
    return;
} else
    /* boot device, ls dir, delay msec? */
if (n == 2 && (res == R_BOOT || res == R_CTTY
              || res == R_DELAY || res == R_LS)
) {
    if (res == R_BOOT) boot_device(second->token);
    if (res == R_CTTY) cty(second->token);
    if (res == R_DELAY) delay(second->token);
    if (res == R_LS) ls(second->token);
    voidtoken();
    voidtoken();
    return;
} else
    /* trap msec command? */
if (n == 3 && res == R_TRAP && numeric(second->token)) {
    long msec= a2l(second->token);

    voidtoken();
    voidtoken();
    schedule(msec, poptoken());
}

```

```

        return;
    } else
        /* Simple command. */
    if (n == 1) {
        char *body;
        int ok= 0;

        name= poptoken();

        switch (res) {
            case R_BOOT:    bootminix();    ok= 1; break;
            case R_DELAY:   delay("500");   ok= 1; break;
            case R_LS:      ls(null);        ok= 1; break;
            case R_MENU:    menu();          ok= 1; break;
            case R_SAVE:    save_parameters(); ok= 1; break;
            case R_SET:     show_env();       ok= 1; break;
            case R_HELP:    help();          ok= 1; break;
            case R_EXIT:    exit(0);         ok= 1; break;
            case R_OFF:     off();           ok= 1; break;
        }

        /* Command to check bootparams: */
        if (strcmp(name, ":") == 0) ok= 1;

        /* User defined function. */
        if (!ok && (body= b_body(name)) != nil) {
            (void) tokenize(&cmds, body);
            ok= 1;
        }
        if (!ok) printf("%s: unknown function", name);
        free(name);
        if (ok) return;
    } else {
        /* Syntax error. */
        printf("Can't parse:");
        while (cmds != sep) {
            printf(" %s", cmds->token); voidtoken();
        }

        /* Getting here means that the command is not understood. */
        printf("\nTry 'help'\n");
        err= 1;
    }
}

int run_trailer(void)
/* Run the trailer function between loading Minix and handing control to it.
 * Return true iff there was no error.
 */
{
    token *save_cmds= cmds;

    cmds= nil;
    (void) tokenize(&cmds, "trailer");
    while (cmds != nil) execute();
    cmds= save_cmds;
    return !err;
}

void monitor(void)
/* Read a line and tokenize it. */
{
    char *line;

    unschedule();          /* Kill a trap. */
    err= 0;                /* Clear error state. */

    if (istty) printf("%s>", bootdev.name);
    line= readline();
    (void) tokenize(&cmds, line);
    free(line);
    (void) escape();        /* Forget if ESC typed. */
}

```



```

#if BIOS

unsigned char cdspec[25];
void bootcdinfo(u32_t, int *, int drive);

void boot(void)
/* Load Minix and start it, among other things. */
{

    /* Initialize tables. */
    initialize();

    /* Get environment variables from the parameter sector. */
    get_parameters();

    while (1) {
        /* While there are commands, execute them! */

        while (cmds != nil) execute();

        /* The "monitor" is just a "read one command" thing. */
        monitor();
    }
}
#endif /* BIOS */

#if UNIX

void main(int argc, char **argv)
/* Do not load or start anything, just edit parameters. */
{
    int i;
    char bootcode[SECTOR_SIZE];
    struct termios rawterm;

    istty= (argc <= 2 && tcgetattr(0, &termbuf) == 0);

    if (argc < 2) {
        fprintf(stderr, "Usage: edparams device [command ...]\n");
        exit(1);
    }

    /* Go over the arguments, changing control characters to spaces. */
    for (i= 2; i < argc; i++) {
        char *p;

        for (p= argv[i]; *p != 0; p++) {
            if ((unsigned) *p < ' ' && *p != '\n') *p= ' ';
        }
    }

    bootdev.name= argv[1];
    if (strncmp(bootdev.name, "/dev/", 5) == 0) bootdev.name+= 5;
    if ((bootdev.device= open(argv[1], O_RDWR, 0666)) < 0)
        fatal(bootdev.name);

    /* Check if it is a bootable Minix device. */
    if (readsectors(mon2abs(bootcode), lowsec, 1) != 0
        || memcmp(bootcode, boot_magic, sizeof(boot_magic)) != 0) {
        fprintf(stderr, "edparams: %s: not a bootable Minix device\n",
            bootdev.name);
        exit(1);
    }

    /* Print greeting message. */
    if (istty) printf("Boot parameters editor.\n");

    signal(SIGINT, trap);
    signal(SIGALRM, trap);

    if (istty) {
        rawterm= termbuf;
        rawterm.c_lflag&= ~(ICANON|ECHO|IEXTEN);
        rawterm.c_cc[VINTR]= ESC;
    }
}

```

```
        if (tcsetattr(0, TCSANOW, &rawterm) < 0) fatal("");
    }

    /* Get environment variables from the parameter sector. */
    get_parameters();

    i= 2;
    for (;;) {
        /* While there are commands, execute them! */
        while (cmds != nil || i < argc) {
            if (cmds == nil) {
                /* A command line command. */
                parse_code(argv[i++]);
            }
            execute();

            /* Bail out on errors if not interactive. */
            if (err && !istty) exit(1);
        }

        /* Commands on the command line? */
        if (argc > 2) break;

        /* The "monitor" is just a "read one command" thing. */
        monitor();
    }
    exit(0);
}
#endif /* UNIX */

/*
 * $PchId: boot.c,v 1.14 2002/02/27 19:46:14 philip Exp $
 */
```

```

/*      boot.h - Info between different parts of boot.  Author: Kees J. Bot
*/

#ifndef DEBUG
#define DEBUG 0
#endif

/* Constants describing the metal: */

#define SECTOR_SIZE      512
#define SECTOR_SHIFT     9
#define RATIO(b)         ((b) / SECTOR_SIZE)

#define PARAMSEC         1      /* Sector containing boot parameters. */

#define DSKBASE           0x1E   /* Floppy disk parameter vector. */
#define DSKPARSIZE       11     /* There are this many bytes of parameters. */

#define ESC               '\33'  /* Escape key. */

#define HEADERPOS        0x00600L /* Place for an array of struct exec's. */

#define FREEPOS          0x08000L /* Memory from FREEPOS to caddr is free to
    * play with.
    */

#if BIOS
#define MSEC_PER_TICK     55     /* Clock does 18.2 ticks per second. */
#define TICKS_PER_DAY    0x1800B0L /* After 24 hours it wraps. */
#endif

#if UNIX
#define MSEC_PER_TICK     1000   /* Clock does 18.2 ticks per second. */
#define TICKS_PER_DAY    86400L  /* Doesn't wrap, but that doesn't matter. */
#endif

#define BOOTPOS          0x07C00L /* Bootstraps are loaded here. */
#define SIGNATURE        0xAA55  /* Proper bootstraps have this signature. */
#define SIGNATOFF        510     /* Offset within bootblock. */

/* BIOS video modes. */
#define MONO_MODE        0x07    /* 80x25 monochrome. */
#define COLOR_MODE       0x03    /* 80x25 color. */

/* Variables shared with boothead.s: */
#ifndef EXTERN
#define EXTERN extern
#endif

typedef struct vector {          /* 8086 vector */
    ul6_t    offset;
    ul6_t    segment;
} vector;

EXTERN vector rem_part;          /* Boot partition table entry. */

EXTERN u32_t caddr, daddr;      /* Code and data address of the boot program. */
EXTERN u32_t runsize;           /* Size of this program. */

EXTERN ul6_t device;            /* Drive being booted from. */

typedef struct {                /* One chunk of free memory. */
    u32_t    base;              /* Start byte. */
    u32_t    size;              /* Number of bytes. */
} memory;

EXTERN memory mem[3];           /* List of available memory. */
EXTERN int mon_return;          /* Monitor stays in memory? */

typedef struct bios_env
{
    ul6_t ax;
    ul6_t bx;
    ul6_t cx;

```

```

    u16_t flags;
} bios_env_t;

#define FL_CARRY          0x0001  /* carry flag */

/* Functions defined by boothead.s: */

void exit(int code);
/* Exit the monitor. */
u32_t mon2abs(void *ptr);
/* Local monitor address to absolute address. */
u32_t vec2abs(vector *vec);
/* Vector to absolute address. */
void raw_copy(u32_t dstaddr, u32_t srcaddr, u32_t count);
/* Copy bytes from anywhere to anywhere. */
u16_t get_word(u32_t addr);
/* Get a word from anywhere. */
void put_word(u32_t addr, U16_t word);
/* Put a word anywhere. */
void relocate(void);
/* Switch to a copy of this program. */
int dev_open(void), dev_close(void);
/* Open device and determine params / close device. */
int dev_boundary(u32_t sector);
/* True if sector is on a track boundary. */
int readsectors(u32_t bufaddr, u32_t sector, U8_t count);
/* Read 1 or more sectors from "device". */
int writesectors(u32_t bufaddr, u32_t sector, U8_t count);
/* Write 1 or more sectors to "device". */
int getch(void);
/* Read a keypress. */
void scan_keyboard(void);
/* Read keypress directly from kb controller. */
void ungetch(int c);
/* Undo a keypress. */
int escape(void);
/* True if escape typed. */
void putch(int c);
/* Send a character to the screen. */
#if BIOS
void pause(void);
/* Wait for an interrupt. */
void serial_init(int line);
#endif
/* Enable copying console I/O to a serial line. */

void set_mode(unsigned mode);
void clear_screen(void);
/* Set video mode / clear the screen. */

u16_t get_bus(void);
/* System bus type, XT, AT, or MCA. */
u16_t get_video(void);
/* Display type, MDA to VGA. */
u32_t get_tick(void);
/* Current value of the clock tick counter. */

void bootstrap(int device, struct part_entry *entry);
/* Execute a bootstrap routine for a different O.S. */
void minix(u32_t koff, u32_t kcs, u32_t kds,
           char *bootparams, size_t paramsize, u32_t aout);
/* Start Minix. */
void int15(bios_env_t *);

/* Shared between boot.c and bootimage.c: */

/* Sticky attributes. */
#define E_SPECIAL          0x01  /* These are known to the program. */
#define E_DEV              0x02  /* The value is a device name. */
#define E_RESERVED         0x04  /* May not be set by user, e.g. 'boot' */
#define E_STICKY           0x07  /* Don't go once set. */

/* Volatile attributes. */
#define E_VAR              0x08  /* Variable */

```

```

#define E_FUNCTION      0x10      /* Function definition. */

/* Variables, functions, and commands. */
typedef struct environment {
    struct environment *next;
    char    flags;
    char    *name;          /* name = value */
    char    *arg;          /* name(arg) {value} */
    char    *value;
    char    *defval;        /* Safehouse for default values. */
} environment;

EXTERN environment *env;      /* Lists the environment. */

char *b_value(char *name);    /* Get/set the value of a variable. */
int b_setvar(int flags, char *name, char *value);

void parse_code(char *code);  /* Parse boot monitor commands. */

extern int fsok;             /* True if the boot device contains an FS. */
EXTERN u32_t lowsec;         /* Offset to the file system on the boot device. */

/* Called by boot.c: */

void bootminix(void);         /* Load and start a Minix image. */

/* Called by bootimage.c: */

void readerr(off_t sec, int err); /* Report a read error. */
char *ul2a(u32_t n, unsigned b), *ul2a10(u32_t n);
/* Transform u32_t to ASCII at base b or base 10. */
long a2l(char *a);
/* Cheap atol(). */
unsigned a2x(char *a);
/* ASCII to hex. */
dev_t name2dev(char *name);
/* Translate a device name to a device number. */
int numprefix(char *s, char **ps);
/* True for a numeric prefix. */
int numeric(char *s);
/* True for a numeric string. */
char *unix_err(int err);
/* Give a descriptive text for some UNIX errors. */
int run_trailer(void);
/* Run the trailer function. */

#if DOS
/* The monitor runs under MS-DOS. */
extern char PSP[256];      /* Program Segment Prefix. */
EXTERN char *vdisk;        /* Name of the virtual disk. */
EXTERN char *drun;         /* Initial command from DOS command line. */
#else
/* The monitor uses only the BIOS. */
#define DOS      0
#endif

void readblock(off_t, char *, int);
void delay(char *);

/*
 * $PchId: boot.h,v 1.12 2002/02/27 19:42:45 philip Exp $
 */

```

```

!      Boothead.s - BIOS support for boot.c                      Author: Kees J. Bot
!
!
! This file contains the startup and low level support for the secondary
! boot program. It contains functions for disk, tty and keyboard I/O,
! copying memory to arbitrary locations, etc.
!
! The primary bootstrap code supplies the following parameters in registers:
!     dl      = Boot-device.
!     es:si   = Partition table entry if hard disk.
!
.text

    o32      =      0x66 ! This assembler doesn't know 386 extensions
    BOOTOFF  =      0x7C00 ! 0x0000:BOOTOFF load a bootstrap here
    LOADSEG  =      0x1000 ! Where this code is loaded.
    BUFFER   =      0x0600 ! First free memory
    PENTRYSIZE =      16 ! Partition table entry size.
    a_flags  =        2 ! From a.out.h, struct exec
    a_text   =        8
    a_data   =       12
    a_bss    =       16
    a_total  =       24
    A_SEP    =      0x20 ! Separate I&D flag
    K_I386   =      0x0001 ! Call Minix in 386 mode
    K_RET    =      0x0020 ! Returns to the monitor on reboot
    K_INT86  =      0x0040 ! Requires generic INT support
    K_MEML   =      0x0080 ! Pass a list of free memory

    DS_SELECTOR =      3*8 ! Kernel data selector
    ES_SELECTOR =      4*8 ! Flat 4 Gb
    SS_SELECTOR =      5*8 ! Monitor stack
    CS_SELECTOR =      6*8 ! Kernel code
    MCS_SELECTOR=      7*8 ! Monitor code

    ESC      =      0x1B ! Escape character

! Imported variables and functions:
.extern _caddr, _daddr, _runsize, _edata, _end ! Runtime environment
.extern _device ! BIOS device number
.extern _rem_part ! To pass partition info
.extern _k_flags ! Special kernel flags
.extern _mem ! Free memory list

.text

! Set segment registers and stack pointer using the programs own header!
! The header is either 32 bytes (short form) or 48 bytes (long form). The
! bootblock will jump to address 0x10030 in both cases, calling one of the
! two jmpf instructions below.

    jmpf     boot, LOADSEG+3 ! Set cs right (skipping long a.out header)
    .space   11 ! jmpf + 11 = 16 bytes
    jmpf     boot, LOADSEG+2 ! Set cs right (skipping short a.out header)
boot:
    mov      ax, #LOADSEG
    mov      ds, ax ! ds = header

    movb     al, a_flags
    testb    al, #A_SEP ! Separate I&D?
    jnz      sepID
comID:
    xor      ax, ax
    xchg     ax, a_text ! No text
    add      a_data, ax ! Treat all text as data
sepID:
    mov      ax, a_total ! Total nontext memory usage
    and      ax, #0xFFFFE ! Round down to even
    mov      a_total, ax ! total - text = data + bss + heap + stack
    cli      ! Ignore interrupts while stack in limbo
    mov      sp, ax ! Set sp at the top of all that

    mov      ax, a_text ! Determine offset of ds above cs
    movb     cl, #4
    shr      ax, cl

```

```

    mov     cx, cs
    add     ax, cx
    mov     ds, ax          ! ds = cs + text / 16
    mov     ss, ax
    sti
    push    es              ! Stack ok now
    mov     es, ax          ! Save es, we need it for the partition table
    cld                    ! C compiler wants UP

! Clear bss
    xor     ax, ax          ! Zero
    mov     di, #_edata     ! Start of bss is at end of data
    mov     cx, #_end       ! End of bss (begin of heap)
    sub     cx, di          ! Number of bss bytes
    shr     cx, #1          ! Number of words
    rep
    stos
    ! Clear bss

! Copy primary boot parameters to variables. (Can do this now that bss is
! cleared and may be written into).
    xorb    dh, dh
    mov     _device, dx     ! Boot device (probably 0x00 or 0x80)
    mov     _rem_part+0, si ! Remote partition table offset
    pop     _rem_part+2     ! and segment (saved es)

! Remember the current video mode for restoration on exit.
    movb    ah, #0x0F       ! Get current video mode
    int     0x10
    andb    al, #0x7F       ! Mask off bit 7 (no blanking)
    movb    old_vid_mode, al
    movb    cur_vid_mode, al

! Give C code access to the code segment, data segment and the size of this
! process.
    xor     ax, ax
    mov     dx, cs
    call    seg2abs
    mov     _caddr+0, ax
    mov     _caddr+2, dx
    xor     ax, ax
    mov     dx, ds
    call    seg2abs
    mov     _daddr+0, ax
    mov     _daddr+2, dx
    push    ds
    mov     ax, #LOADSEG
    mov     ds, ax          ! Back to the header once more
    mov     ax, a_total+0
    mov     dx, a_total+2   ! dx:ax = data + bss + heap + stack
    add     ax, a_text+0
    adc     dx, a_text+2    ! dx:ax = text + data + bss + heap + stack
    pop     ds
    mov     _runsize+0, ax
    mov     _runsize+2, dx  ! 32 bit size of this process

! Determine available memory as a list of (base,size) pairs as follows:
! mem[0] = low memory, mem[1] = memory between 1M and 16M, mem[2] = memory
! above 16M. Last two coalesced into mem[1] if adjacent.
    mov     di, #_mem       ! di = memory list
    int     0x12            ! Returns low memory size (in K) in ax
    mul     c1024
    mov     4(di), ax       ! mem[0].size = low memory size in bytes
    mov     6(di), dx
    call    _getprocessor
    cmp     ax, #286        ! Only 286s and above have extended memory
    jb      no_ext
    cmp     ax, #486        ! Assume 486s were the first to have >64M
    jb      small_ext       ! (It helps to be paranoid when using the BIOS)

big_ext:
    mov     ax, #0xE801     ! Code for get memory size for >64M
    int     0x15            ! ax = mem at 1M per 1K, bx = mem at 16M per 64K
    jnc     got_ext

small_ext:
    movb    ah, #0x88      ! Code for get extended memory size

```

```

        clc                ! Carry will stay clear if call exists
        int     0x15        ! Returns size (in K) in ax for AT's
        jc      no_ext
        test    ax, ax      ! An AT with no extended memory?
        jz      no_ext
        xor     bx, bx      ! bx = mem above 16M per 64K = 0
got_ext:
        mov     cx, ax      ! cx = copy of ext mem at 1M
        mov     10(di), #0x0010 ! mem[1].base = 0x00100000 (1M)
        mul     c1024
        mov     12(di), ax  ! mem[1].size = "ext mem at 1M" * 1024
        mov     14(di), dx
        test    bx, bx
        jz      no_ext      ! No more ext mem above 16M?
        cmp     cx, #15*1024 ! Chunks adjacent? (precisely 15M at 1M?)
        je      adj_ext
        mov     18(di), #0x0100 ! mem[2].base = 0x01000000 (16M)
        mov     22(di), bx   ! mem[2].size = "ext mem at 16M" * 64K
        jmp     no_ext
adj_ext:
        add     14(di), bx    ! Add ext mem above 16M to mem below 16M
no_ext:

! Time to switch to a higher level language (not much higher)
        call    _boot

! void ..exit(int status)
!      Exit the monitor by rebooting the system.
#define _exit, __exit, ___exit      ! Make various compilers happy
_exit:
__exit:
___exit:
        mov     bx, sp
        cmp     2(bx), #0      ! Good exit status?
        jz      reboot
quit:   mov     ax, #any_key
        push    ax
        call    _printf
        xorb    ah, ah        ! Read character from keyboard
        int     0x16
reboot: call    dev_reset
        call    restore_video
        int     0x19          ! Reboot the system

.data
any_key:
        .ascii  "\nHit any key to reboot\n\0"

.text

! u32_t mon2abs(void *ptr)
!      Address in monitor data to absolute address.
#define _mon2abs
_mon2abs:
        mov     bx, sp
        mov     ax, 2(bx)      ! ptr
        mov     dx, ds         ! Monitor data segment
        jmp     seg2abs

! u32_t vec2abs(vector *vec)
!      8086 interrupt vector to absolute address.
#define _vec2abs
_vec2abs:
        mov     bx, sp
        mov     bx, 2(bx)
        mov     ax, (bx)
        mov     dx, 2(bx)      ! dx:ax vector
        ! jmp     seg2abs      ! Translate

seg2abs:                                ! Translate dx:ax to the 32 bit address dx-ax
        push    cx
        movb    ch, dh
        movb    cl, #4
        shl     dx, cl
        shrb    ch, cl          ! ch-dx = dx << 4

```



```

    add     ax, dx
    adcb    ch, #0          ! ch-ax = ch-dx + ax
    movb    dl, ch
    xorb    dh, dh          ! dx-ax = ch-ax
    pop     cx
    ret

abs2seg:                                ! Translate the 32 bit address dx-ax to dx:ax
    push    cx
    movb    ch, dl
    mov     dx, ax          ! ch-dx = dx-ax
    and     ax, #0x000F     ! Offset in ax
    movb    cl, #4
    shr     dx, cl
    shlb    ch, cl
    orb     dh, ch          ! dx = ch-dx >> 4
    pop     cx
    ret

! void raw_copy(u32_t dstaddr, u32_t srcaddr, u32_t count)
!     Copy count bytes from srcaddr to dstaddr. Don't do overlaps.
!     Also handles copying words to or from extended memory.
#define _raw_copy
_raw_copy:
    push    bp
    mov     bp, sp
    push    si
    push    di              ! Save C variable registers
copy:
    cmp     14(bp), #0
    jnz     bigcopy
    mov     cx, 12(bp)
    jcxz    copydone        ! Count is zero, end copy
    cmp     cx, #0xFFFF0
    jb      smallcopy
bigcopy:mov     cx, #0xFFFF0    ! Don't copy more than about 64K at once
smallcopy:
    push    cx              ! Save copying count
    mov     ax, 4(bp)
    mov     dx, 6(bp)
    cmp     dx, #0x0010     ! Copy to extended memory?
    jae     ext_copy
    cmp     10(bp), #0x0010 ! Copy from extended memory?
    jae     ext_copy
    call    abs2seg
    mov     di, ax
    mov     es, dx          ! es:di = dstaddr
    mov     ax, 8(bp)
    mov     dx, 10(bp)
    call    abs2seg
    mov     si, ax
    mov     ds, dx          ! ds:si = srcaddr
    shr     cx, #1          ! Words to move
    rep     movs             ! Do the word copy
    adc     cx, cx          ! One more byte?
    rep     movsb           ! Do the byte copy
    mov     ax, ss          ! Restore ds and es from the remaining ss
    mov     ds, ax
    mov     es, ax
    jmp     copyadjust

ext_copy:
    mov     x_dst_desc+2, ax
    movb    x_dst_desc+4, dl ! Set base of destination segment
    mov     ax, 8(bp)
    mov     dx, 10(bp)
    mov     x_src_desc+2, ax
    movb    x_src_desc+4, dl ! Set base of source segment
    mov     si, #x_gdt      ! es:si = global descriptor table
    shr     cx, #1          ! Words to move
    movb    ah, #0x87       ! Code for extended memory move
    int     0x15
copyadjust:

```

```

        pop        cx                ! Restore count
        add        4(bp), cx
        adc        6(bp), #0         ! srcaddr += copycount
        add        8(bp), cx
        adc        10(bp), #0        ! dstaddr += copycount
        sub        12(bp), cx
        sbb        14(bp), #0        ! count -= copycount
        jmp        copy              ! and repeat
copydone:
        pop        di
        pop        si                ! Restore C variable registers
        pop        bp
        ret

! u16_t get_word(u32_t addr);
! void put_word(u32_t addr, u16_t word);
!      Read or write a 16 bits word at an arbitrary location.
#define _get_word, _put_word
_get_word:
        mov        bx, sp
        call       gp_getaddr
        mov        ax, (bx)          ! Word to get from addr
        jmp        gp_ret
_put_word:
        mov        bx, sp
        push       6(bx)             ! Word to store at addr
        call       gp_getaddr
        pop        (bx)              ! Store the word
        jmp        gp_ret
gp_getaddr:
        mov        ax, 2(bx)
        mov        dx, 4(bx)
        call       abs2seg
        mov        bx, ax
        mov        ds, dx            ! ds:bx = addr
        ret
gp_ret:
        push       es
        pop        ds                ! Restore ds
        ret

! void relocate(void);
!      After the program has copied itself to a safer place, it needs to change
!      the segment registers. Caddr has already been set to the new location.
#define _relocate
_relocate:
        pop        bx                ! Return address
        mov        ax, _caddr+0
        mov        dx, _caddr+2
        call       abs2seg
        mov        cx, dx            ! cx = new code segment
        mov        ax, cs            ! Old code segment
        sub        ax, cx            ! ax = -(new - old) = -Moving offset
        mov        dx, ds
        sub        dx, ax
        mov        ds, dx            ! ds += (new - old)
        mov        es, dx
        mov        ss, dx
        xor        ax, ax
        call       seg2abs
        mov        _daddr+0, ax
        mov        _daddr+2, dx      ! New data address
        push       cx                ! New text segment
        push       bx                ! Return offset of this function
        retf                        ! Relocate

! void *brk(void *addr)
! void *sbrk(size_t incr)
!      Cannot fail implementations of brk(2) and sbrk(3), so we can use
!      malloc(3). They reboot on stack collision instead of returning -1.
.data
        .align 2
break: .data2 _end                ! A fake heap pointer
.text

```

```

.define _brk, __brk, _sbrk, __sbrk
_brk:
__brk:                                ! __brk is for the standard C compiler
    xor     ax, ax
    jmp     sbrk                      ! break= 0; return sbrk(addr);
_sbrk:
__sbrk:
sbrk:  mov     ax, break              ! ax= current break
    push    ax                      ! save it as future return value
    mov     bx, sp
    add     ax, 4(bx)                ! Stack is now: (retval, retaddr, incr, ...)
    mov     break, ax                ! ax= break + increment
    lea     dx, -1024(bx)            ! Set new break
    cmp     dx, ax                  ! sp minus a bit of breathing space
    jnb     heaperr                  ! Compare with the new break
    lea     dx, -4096(bx)            ! Suffocating noises
    cmp     dx, ax                  ! A warning when heap+stack goes < 4K
    jae     plenty                  ! No reason to complain
    mov     ax, #memwarn
    push    ax
    call    _printf                  ! Warn about memory running low
    pop     ax
    movb    memwarn, #0              ! No more warnings
plenty: pop     ax                  ! Return old break (0 for brk)
    ret
heaperr: mov    ax, #chmem
    push    ax
    mov     ax, #nomem
    push    ax
    call    _printf
    jmp     quit

.data
nomem:  .ascii  "\nOut of%s\0"
memwarn: .ascii  "\nLow on"
chmem:  .ascii  " memory, use chmem to increase the heap\n\0"
.text

! int dev_open(void);
!     Given the device "_device" figure out if it exists and what its number
!     of heads and sectors may be.  Return the BIOS error code on error,
!     otherwise 0.
.define _dev_open
_dev_open:
    call    dev_reset                ! Optionally reset the disks
    movb    dev_state, #0            ! State is "closed"
    push    es
    push    di                      ! Save registers used by BIOS calls
    movb    dl, _device              ! The default device
    cmpb    dl, #0x80                ! Floppy < 0x80, winchester >= 0x80
    jae     winchester
floppy:
finit0:  mov     di, #3                ! Three tries to init drive by reading sector 0
    xor     ax, ax
    mov     es, ax
    mov     bx, #BUFFER              ! es:bx = scratch buffer
    mov     ax, #0x0201              ! Read sector, #sectors = 1
    mov     cx, #0x0001              ! Track 0, first sector
    xorb    dh, dh                  ! Drive dl, head 0
    int     0x13
    jnc     finit0ok                 ! Sector 0 read ok?
    cmpb    ah, #0x80                ! Disk timed out? (Floppy drive empty)
    je      geoerr
    dec     di
    jz      geoerr
    xorb    ah, ah                  ! Reset drive
    int     0x13
    jc      geoerr
    jmp     finit0                  ! Retry once more, it may need to spin up
finit0ok:
flast:  mov     di, #seclist          ! List of per floppy type sectors/track
    movb    cl, (di)                ! Sectors per track to test
    cmpb    cl, #9                  ! No need to do the last 720K/360K test
    je      ftestok
    xor     ax, ax

```

```

    mov     es, ax
    mov     bx, #BUFFER      ! es:bx = scratch buffer
    mov     ax, #0x0201      ! Read sector, #sectors = 1
    xorb    ch, ch           ! Track 0, last sector
    xorb    dh, dh           ! Drive dl, head 0
    int     0x13
    jnc     ftestok          ! Sector cl read ok?
    xorb    ah, ah           ! Reset drive
    int     0x13
    jc      geoerr
    inc     di                ! Try next sec/track number
    jmp     flast
ftestok:
    movb    dh, #2           ! Floppies have two sides
    jmp     geoboth
winchester:
    movb    ah, #0x08        ! Code for drive parameters
    int     0x13             ! dl still contains drive
    jc      geoerr           ! No such drive?
    andb    cl, #0x3F        ! cl = max sector number (1-origin)
    incb    dh               ! dh = 1 + max head number (0-origin)
geoboth:
    movb    sectors, cl      ! Sectors per track
    movb    al, cl           ! al = sectors per track
    mulb    dh               ! ax = heads * sectors
    mov     secspcyl, ax     ! Sectors per cylinder = heads * sectors
    movb    dev_state, #1    ! Device state is "open"
    xor     ax, ax           ! Code for success
geodone:
    pop     di
    pop     es               ! Restore di and es registers
    ret
geoerr:
    movb    al, ah
    xorb    ah, ah           ! ax = BIOS error code
    jmp     geodone
.data
seclist:
    .data1  18, 15, 9       ! 1.44M, 1.2M, and 360K/720K floppy sec/track
.text
! int dev_close(void);
!     Close the current device.  Under the BIOS this does nothing much.
.define _dev_close
_dev_close:
    xor     ax, ax
    movb    dev_state, al    ! State is "closed"
    ret

! Reset the disks if needed.  Minix may have messed things up.
dev_reset:
    cmpb    dev_state, #0    ! Need reset if dev_state < 0
    jge     0f
    xorb    ah, ah           ! Reset (ah = 0)
    movb    dl, #0x80        ! All disks
    int     0x13
    movb    dev_state, #0    ! State is "closed"
0:
    ret

! int dev_boundary(u32_t sector);
!     True if a sector is on a boundary, i.e. sector % sectors == 0.
.define _dev_boundary
_dev_boundary:
    mov     bx, sp
    xor     dx, dx
    mov     ax, 4(bx)        ! divide high half of sector number
    div     sectors
    mov     ax, 2(bx)        ! divide low half of sector number
    div     sectors          ! dx = sector % sectors
    sub     dx, #1           ! CF = dx == 0
    sbb     ax, ax           ! ax = -CF
    neg     ax               ! ax = (sector % sectors) == 0
    ret

! int readsectors(u32_t bufaddr, u32_t sector, u8_t count)

```

```

! int writesectors(u32_t bufaddr, u32_t sector, u8_t count)
!     Read/write several sectors from/to disk or floppy. The buffer must
!     be between 64K boundaries! Count must fit in a byte. The external
!     variables _device, sectors and secspcyl describe the disk and its
!     geometry. Returns 0 for success, otherwise the BIOS error code.
!
.define _readsectors, _writesectors
_writesectors:
    push    bp
    mov     bp, sp
    movb    13(bp), #0x03    ! Code for a disk write
    jmp     rwsec
_readsectors:
    push    bp
    mov     bp, sp
    movb    13(bp), #0x02    ! Code for a disk read
rwsec:
    push    si
    push    di
    push    es
    cmpb    dev_state, #0    ! Device state?
    jg      0f               ! >0 if open
    call    _dev_open        ! Initialize
    test    ax, ax
    jnz     badopen
0:
    mov     ax, 4(bp)
    mov     dx, 6(bp)
    call    abs2seg
    mov     bx, ax
    mov     es, dx          ! es:bx = bufaddr
    mov     di, #3          ! Execute 3 resets on floppy error
    cmpb    _device, #0x80
    jb      nohd
nohd:
    mov     di, #1          ! But only 1 reset on hard disk error
    cmpb    12(bp), #0      ! count equals zero?
    jz      done
more:
    mov     ax, 8(bp)
    mov     dx, 10(bp)      ! dx:ax = abs sector. Divide it by sectors/cyl
    cmp     dx, #[1024*255*63-255]>>16 ! Near 8G limit?
    jae     bigdisk
    div     secspcyl        ! ax = cylinder, dx = sector within cylinder
    xchg    ax, dx          ! ax = sector within cylinder, dx = cylinder
    movb    ch, dl          ! ch = low 8 bits of cylinder
    divb    sectors        ! al = head, ah = sector (0-origin)
    xorb    dl, dl          ! About to shift bits 8-9 of cylinder into dl
    shr     dx, #1
    shr     dx, #1          ! dl[6..7] = high cylinder
    orb     dl, ah          ! dl[0..5] = sector (0-origin)
    movb    cl, dl          ! cl[0..5] = sector, cl[6..7] = high cyl
    incb    cl              ! cl[0..5] = sector (1-origin)
    movb    dh, al          ! dh = head
    movb    dl, _device     ! dl = device to use
    movb    al, sectors     ! Sectors per track - Sector number (0-origin)
    subb    al, ah          ! = Sectors left on this track
    cmpb    al, 12(bp)      ! Compare with # sectors to transfer
    jbe     doit
    movb    al, 12(bp)      ! 12(bp) < sectors left on this track
doit:
    movb    ah, 13(bp)      ! Code for disk read (0x02) or write (0x03)
    push    ax              ! Save al = sectors to read
    int     0x13            ! call the BIOS to do the transfer
    pop     cx              ! Restore al in cl
    jmp     rdeval
bigdisk:
    mov     si, #ext_rw     ! si = extended read/write parameter packet
    movb    cl, 12(bp)
    movb    2(si), cl        ! Fill in # blocks to transfer
    mov     4(si), bx        ! Buffer address = es:bx
    mov     6(si), es
    mov     8(si), ax        ! Starting block number = dx:ax
    mov     10(si), dx
    movb    dl, _device     ! dl = device to use
    mov     ax, #0x4000      ! This, or-ed with 0x02 or 0x03 becomes
    orb     ah, 13(bp)      ! extended read (0x4200) or write (0x4300)
    int     0x13
    ! jmp    rdeval

```

```

rdeval:
    jc      ioerr          ! I/O error
    movb    al, cl         ! Restore al = sectors read
    addb    bh, al         ! bx += 2 * al * 256 (add bytes transferred)
    addb    bh, al         ! es:bx = where next sector is located
    add     8(bp), ax      ! Update address by sectors transferred
    adc     10(bp), #0     ! Don't forget high word
    subb    12(bp), al     ! Decrement sector count by sectors transferred
    jnz     more          ! Not all sectors have been transferred
done:    xorb  ah, ah       ! No error here!
        jmp  finish
ioerr:   cmpb  ah, #0x80    ! Disk timed out? (Floppy drive empty)
        je   finish
        cmpb  ah, #0x03    ! Disk write protected?
        je   finish
        dec   di          ! Do we allow another reset?
        jl   finish       ! No, report the error
        xorb  ah, ah       ! Code for a reset (0)
        int   0x13
        jnc   more        ! Successful reset, try request again
finish:  movb  al, ah
        xorb  ah, ah       ! ax = error number
badopen: pop  es
        pop  di
        pop  si
        pop  bp
        ret
.data
    .align 4
! Extended read/write commands require a parameter packet.
ext_rw:
    .data1  0x10          ! Length of extended r/w packet
    .data1  0             ! Reserved
    .data2  0             ! Blocks to transfer (to be filled in)
    .data2  0             ! Buffer address offset (tbfi)
    .data2  0             ! Buffer address segment (tbfi)
    .data4  0             ! Starting block number low 32 bits (tbfi)
    .data4  0             ! Starting block number high 32 bits
.text
! int getch(void);
! Read a character from the keyboard, and check for an expired timer.
! A carriage return is changed into a linefeed for UNIX compatibility.
#define _getch
_getch:
    xor     ax, ax
    xchg    ax, unchar     ! Ungotten character?
    test    ax, ax
    jnz     gotch
getch:
    hlt     ! Play dead until interrupted (see pause())
    movb    ah, #0x01      ! Keyboard status
    int     0x16
    jz      0f             ! Nothing typed
    xorb    ah, ah         ! Read character from keyboard
    int     0x16
    jmp     press          ! Keypress
0:         mov     dx, line  ! Serial line?
    test    dx, dx
    jz      0f
    add     dx, #5         ! Line Status Register
    inb     dx
    testb   al, #0x01      ! Data Ready?
    jz      0f
    mov     dx, line
    !add    dx, 0          ! Receive Buffer Register
    inb     dx             ! Get character
    jmp     press
0:         call    _expired  ! Timer expired?
    test    ax, ax
    jz      getch
    mov     ax, #ESC       ! Return ESC
    ret
press:

```

```

        cmpb    al, #0x0D        ! Carriage return?
        jnz     nocr
nocr:    movb    al, #0x0A        ! Change to linefeed
        cmpb    al, #ESC        ! Escape typed?
        jne     noesc
        inc     escape          ! Set flag
noesc:   xorb    ah, ah          ! ax = al
gotch:   ret

! int ungetch(void);
!       Return a character to undo a getch().
#define _ungetch
_ungetch:
        mov     bx, sp
        mov     ax, 2(bx)
        mov     unchar, ax
        ret

! int escape(void);
!       True if ESC has been typed.
#define _escape
_escape:
        movb    ah, #0x01        ! Keyboard status
        int     0x16
        jz      escflg          ! Keypress?
        cmpb    al, #ESC        ! Escape typed?
        jne     escflg
        xorb    ah, ah          ! Discard the escape
        int     0x16
        inc     escape          ! Set flag
escflg:  xor     ax, ax
        xchg    ax, escape      ! Escape typed flag
        ret

! int putch(int c);
!       Write a character in teletype mode. The putk synonym is
!       for the kernel printf function that uses it.
!       Newlines are automatically preceded by a carriage return.
!
#define _putch, _putk
_putch:
_putk:   mov     bx, sp
        movb    al, 2(bx)        ! al = character to be printed
        testb   al, al          ! Kernel printf adds a null char to flush queue
        jz      nulch
        cmpb    al, #0x0A        ! al = newline?
        jnz     putc
        movb    al, #0x0D
        call    putc            ! putc('\r')
        movb    al, #0x0A        ! Restore the '\n' and print it
putc:    movb    ah, #0x0E        ! Print character in teletype mode
        mov     bx, #0x0001      ! Page 0, foreground color
        int     0x10
        mov     bx, line        ! Serial line?
        test    bx, bx
        jz      nulch
        push    ax              ! Save character to print
        call    _get_tick       ! Current clock tick counter
        mov     cx, ax
        add     cx, #2          ! Don't want to see it count twice
1:        lea     dx, 5(bx)      ! Line Status Register
        inb     dx
        testb   al, #0x20        ! Transmitter Holding Register Empty?
        jnz     0f
        call    _get_tick
        cmp     ax, cx          ! Clock ticked more than once?
        jne     1b
0:        pop     ax              ! Restore character to print
        mov     dx, bx          ! Transmit Holding Register
        outb    dx              ! Send character down the serial line
nulch:   ret

! void pause(void);
!       Wait for an interrupt using the HLT instruction. This either saves

```

```

! power, or tells an x86 emulator that nothing is happening right now.
.define _pause
_pause:
    hlt
    ret

! void set_mode(unsigned mode);
! void clear_screen(void);
! Set video mode / clear the screen.
.define _set_mode, _clear_screen
_set_mode:
    mov     bx, sp
    mov     ax, 2(bx)      ! Video mode
    cmp     ax, cur_vid_mode
    je      modeok        ! Mode already as requested?
    mov     cur_vid_mode, ax
_clear_screen:
    xor     ax, ax
    mov     es, ax        ! es = Vector segment
    mov     ax, cur_vid_mode
    movb    ch, ah        ! Copy of the special flags
    andb    ah, #0x0F     ! Test bits 8-11, clear special flags
    jnz     xvesa        ! VESA extended mode?
    int     0x10          ! Reset video (ah = 0)
    jmp     md_480
xvesa:     mov     bx, ax        ! bx = extended mode
    mov     ax, #0x4F02     ! Reset video
    int     0x10
md_480:    ! Basic video mode is set, now build on it
    testb   ch, #0x20      ! 480 scan lines requested?
    jz      md_14pt
    mov     dx, #0x3CC     ! Get CRTC port
    inb     dx
    movb    dl, #0xD4
    testb   al, #1        ! Mono or color?
    jnz     0f
    movb    dl, #0xB4
0:         mov     ax, #0x110C ! Vertical sync end (also unlocks CR0-7)
    call    out2
    mov     ax, #0x060B    ! Vertical total
    call    out2
    mov     ax, #0x073E    ! (Vertical) overflow
    call    out2
    mov     ax, #0x10EA    ! Vertical sync start
    call    out2
    mov     ax, #0x12DF    ! Vertical display end
    call    out2
    mov     ax, #0x15E7    ! Vertical blank start
    call    out2
    mov     ax, #0x1604    ! Vertical blank end
    call    out2
    push    dx
    movb    dl, #0xCC     ! Misc output register (read)
    inb     dx
    movb    dl, #0xC2     ! (write)
    andb    al, #0x0D     ! Preserve clock select bits and color bit
    orb     al, #0xE2     ! Set correct sync polarity
    outb    dx
    pop     dx            ! Index register still in dx
md_14pt:   testb   ch, #0x40 ! 9x14 point font requested?
    jz      md_8pt
    mov     ax, #0x1111    ! Load ROM 9 by 14 font
    xorb    bl, bl        ! Load block 0
    int     0x10
    testb   ch, #0x20      ! 480 scan lines?
    jz      md_8pt
    mov     ax, #0x12DB    ! VGA vertical display end
    call    out2
    movb    eseg, 0x0484, #33 ! Tell BIOS the last line number
md_8pt:    testb   ch, #0x80 ! 8x8 point font requested?
    jz      setcur
    mov     ax, #0x1112    ! Load ROM 8 by 8 font

```



```

    xorb    bl, bl        ! Load block 0
    int     0x10
    testb   ch, #0x20     ! 480 scan lines?
    jz      setcur
    mov     ax, #0x12DF    ! VGA vertical display end
    call    out2
eseg movb   0x0484, #59    ! Tell BIOS the last line number
setcur:
    xor     dx, dx        ! dl = column = 0, dh = row = 0
    xorb    bh, bh        ! Page 0
    movb    ah, #0x02     ! Set cursor position
    int     0x10
    push    ss
    pop     es            ! Restore es
modeok: ret

! Out to the usual [index, data] port pair that are common for VGA devices
! dx = port, ah = index, al = data.
out2:
    push    dx
    push    ax
    movb    al, ah
    outb    dx            ! Set index
    inc     dx
    pop     ax
    outb    dx            ! Send data
    pop     dx
    ret

restore_video:           ! To restore the video mode on exit
    mov     ax, old_vid_mode
    push    ax
    call    _set_mode
    pop     ax
    ret

! void serial_init(int line)
!     Initialize copying console I/O to a serial line.
#define _serial_init
_serial_init:
    mov     bx, sp
    mov     dx, 2(bx)     ! Line number
    push    ds
    xor     ax, ax
    mov     ds, ax        ! Vector and BIOS data segment
    mov     bx, dx        ! Line number
    shl     bx, #1        ! Word offset
    mov     bx, 0x0400(bx) ! I/O port for the given line
    pop     ds
    mov     line, bx      ! Remember I/O port
serial_init:
    mov     bx, line
    test    bx, bx        ! I/O port must be nonzero
    jz      0f
    mov     ax, #0x00E3    ! 9600 N-8-1
    int     0x14          ! Initialize serial line dx
    lea     dx, 4(bx)      ! Modem Control Register
    movb    al, #0x0B      ! DTR, RTS, OUT2
    outb    dx
0:      ret

! u32_t get_tick(void);
!     Return the current value of the clock tick counter. This counter
!     increments 18.2 times per second. Poll it to do delays. Does not
!     work on the original PC, but works on the PC/XT.
#define _get_tick
_get_tick:
    push    cx
    xorb    ah, ah        ! Code for get tick count
    int     0x1A
    mov     ax, dx
    mov     dx, cx        ! dx:ax = cx:dx = tick count
    pop     cx
    ret

```

```
! Functions used to obtain info about the hardware.  Boot uses this information
! itself, but will also pass them on to a pure 386 kernel, because one can't
! make BIOS calls from protected mode.  The video type could probably be
! determined by the kernel too by looking at the hardware, but there is a small
! chance on errors that the monitor allows you to correct by setting variables.
```

```
.define _get_bus          ! returns type of system bus
.define _get_video        ! returns type of display
```

```
! ul6_t get_bus(void)
!   Return type of system bus, in order: XT, AT, MCA.
```

```
_get_bus:
    call    _getprocessor
    xor     dx, dx                ! Assume XT
    cmp     ax, #286              ! An AT has at least a 286
    jb      got_bus
    inc     dx                    ! Assume AT
    movb    ah, #0xC0             ! Code for get configuration
    int     0x15
    jc      got_bus              ! Carry clear and ah = 00 if supported
    testb   ah, ah
    jne     got_bus
    eseg
    movb    al, 5(bx)             ! Load feature byte #1
    inc     dx                    ! Assume MCA
    testb   al, #0x02             ! Test bit 1 - "bus is Micro Channel"
    jnz     got_bus
    dec     dx                    ! Assume AT
    testb   al, #0x40             ! Test bit 6 - "2nd 8259 installed"
    jnz     got_bus
    dec     dx                    ! It is an XT
got_bus:
    push    ds
    pop     es                    ! Restore es
    mov     ax, dx                ! Return bus code
    mov     bus, ax               ! Keep bus code, A20 handler likes to know
    ret
```

```
! ul6_t get_video(void)
!   Return type of display, in order: MDA, CGA, mono EGA, color EGA,
!   mono VGA, color VGA.
```

```
_get_video:
    mov     ax, #0x1A00           ! Function 1A returns display code
    int     0x10                 ! al = 1A if supported
    cmpb    al, #0x1A
    jnz     no_dc                ! No display code function supported

    mov     ax, #2
    cmpb    bl, #5                ! Is it a monochrome EGA?
    jz      got_video
    inc     ax
    cmpb    bl, #4                ! Is it a color EGA?
    jz      got_video
    inc     ax
    cmpb    bl, #7                ! Is it a monochrome VGA?
    jz      got_video
    inc     ax
    cmpb    bl, #8                ! Is it a color VGA?
    jz      got_video

no_dc:
    movb    ah, #0x12             ! Get information about the EGA
    movb    bl, #0x10
    int     0x10
    cmpb    bl, #0x10             ! Did it come back as 0x10? (No EGA)
    jz      no_ega

    mov     ax, #2
    cmpb    bh, #1                ! Is it monochrome?
    jz      got_video
    inc     ax
    jmp     got_video
```

```

no_ega: int      0x11          ! Get bit pattern for equipment
        and      ax, #0x30     ! Isolate color/mono field
        sub      ax, #0x30
        jz       got_video     ! Is it an MDA?
        mov      ax, #1        ! No it's CGA

got_video:
        ret

! Functions to leave the boot monitor.
.define _bootstrap      ! Call another bootstrap
.define _minix          ! Call Minix

! void _bootstrap(int device, struct part_entry *entry)
!     Call another bootstrap routine to boot MS-DOS for instance. (No real
!     need for that anymore, now that you can format floppies under Minix).
!     The bootstrap must have been loaded at BOOTSEG from "device".
_bootstrap:
        call      restore_video
        mov      bx, sp
        movb     dl, 2(bx)      ! Device to boot from
        mov      si, 4(bx)      ! ds:si = partition table entry
        xor      ax, ax
        mov      es, ax        ! Vector segment
        mov      di, #BUFFER    ! es:di = buffer in low core
        mov      cx, #PENTRYSIZE ! cx = size of partition table entry
rep     movsb
        mov      si, #BUFFER    ! es:si = partition table entry
        mov      ds, ax        ! Some bootstraps need zero segment registers
        cli
        mov      ss, ax
        mov      sp, #BOOTOFF   ! This should do it
        sti
        jmpf     BOOTOFF, 0      ! Back to where the BIOS loads the boot code

! void minix(u32_t koff, u32_t kcs, u32_t kds,
!           char *bootparams, size_t paramsize, u32_t aout);
!     Call Minix.
_minix:
        push     bp
        mov      bp, sp        ! Pointer to arguments

        mov      dx, #0x03F2    ! Floppy motor drive control bits
        movb     al, #0x0C      ! Bits 4-7 for floppy 0-3 are off
        outb     dx
        push     ds
        xor      ax, ax        ! Vector & BIOS data segments
        mov      ds, ax
        andb     0x043F, #0xF0  ! Clear diskette motor status bits of BIOS
        pop      ds
        cli                    ! No more interruptions

        test     _k_flags, #K_I386 ! Switch to 386 mode?
        jnz      minix386

! Call Minix in real mode.
minix86:
        test     _k_flags, #K_MEML ! New memory arrangements?
        jz       0f
        push     22(bp)          ! Address of a.out headers
        push     20(bp)

0:
        push     18(bp)          ! # bytes of boot parameters
        push     16(bp)          ! Address of boot parameters

        test     _k_flags, #K_RET ! Can the kernel return?
        jz       noret86
        xor      dx, dx          ! If little ext mem then monitor not preserved
        xor      ax, ax
        cmp      _mon_return, ax ! Minix can return to the monitor?
        jz       0f
        mov      dx, cs          ! Monitor far return address
        mov      ax, #ret86

```

```

0:      push    dx                ! Push monitor far return address or zero
      push    ax
noret86:

      mov     ax, 8(bp)
      mov     dx, 10(bp)
      call    abs2seg
      push    dx                ! Kernel code segment
      push    4(bp)             ! Kernel code offset
      mov     ax, 12(bp)
      mov     dx, 14(bp)
      call    abs2seg
      mov     ds, dx            ! Kernel data segment
      mov     es, dx            ! Set es to kernel data too
      retf                     ! Make a far call to the kernel

! Call Minix in 386 mode.
minix386:
cseg   mov     cs_real-2, cs     ! Patch CS and DS into the instructions that
cseg   mov     ds_real-2, ds     ! reload them when switching back to real mode
      .data1   0x0F,0x20,0xC0    ! mov     eax, cr0
      orb     al, #0x01          ! Set PE (protection enable) bit
      .data1   032
      mov     msw, ax           ! Save as protected mode machine status word

      mov     dx, ds            ! Monitor ds
      mov     ax, #p_gdt         ! dx:ax = Global descriptor table
      call    seg2abs
      mov     p_gdt_desc+2, ax
      movb    p_gdt_desc+4, dl ! Set base of global descriptor table

      mov     ax, 12(bp)
      mov     dx, 14(bp)        ! Kernel ds (absolute address)
      mov     p_ds_desc+2, ax
      movb    p_ds_desc+4, dl ! Set base of kernel data segment

      mov     dx, ss            ! Monitor ss
      xor     ax, ax            ! dx:ax = Monitor stack segment
      call    seg2abs           ! Minix starts with the stack of the monitor
      mov     p_ss_desc+2, ax
      movb    p_ss_desc+4, dl

      mov     ax, 8(bp)
      mov     dx, 10(bp)        ! Kernel cs (absolute address)
      mov     p_cs_desc+2, ax
      movb    p_cs_desc+4, dl

      mov     dx, cs            ! Monitor cs
      xor     ax, ax            ! dx:ax = Monitor code segment
      call    seg2abs
      mov     p_mcs_desc+2, ax
      movb    p_mcs_desc+4, dl

      push    #MCS_SELECTOR
      test    _k_flags, #K_INT86 ! Generic INT86 support?
      jz      0f
      push    #int86             ! Far address to INT86 support
      jmp     1f
0:      push    #bios13           ! Far address to BIOS int 13 support
1:

      test    _k_flags, #K_MEML ! New memory arrangements?
      jz      0f
      .data1   032
0:      push    20(bp)            ! Address of a.out headers

      push    #0
      push    18(bp)            ! 32 bit size of parameters on stack
      push    #0
      push    16(bp)            ! 32 bit address of parameters (ss relative)

      test    _k_flags, #K_RET ! Can the kernel return?
      jz      noret386
      push    #MCS_SELECTOR
      push    #ret386           ! Monitor far return address

```

noret386:

```

    push    #0
    push    #CS_SELECTOR
    push    6(bp)
    push    4(bp)                ! 32 bit far address to kernel entry point

    call    real2prot            ! Switch to protected mode
    mov     ax, #DS_SELECTOR ! Kernel data
    mov     ds, ax
    mov     ax, #ES_SELECTOR ! Flat 4 Gb
    mov     es, ax
    .data1  o32                ! Make a far call to the kernel
    retf

```

! Minix-86 returns here on a halt or reboot.

ret86:

```

    mov     _reboot_code+0, ax
    mov     _reboot_code+2, dx    ! Return value (obsolete method)
    jmp     return

```

! Minix-386 returns here on a halt or reboot.

ret386:

```

    .data1  o32
    mov     _reboot_code, ax      ! Return value (obsolete method)
    call    prot2real            ! Switch to real mode

```

**return:**

```

    mov     sp, bp                ! Pop parameters
    sti                     ! Can take interrupts again

```

```

    call    _get_video            ! MDA, CGA, EGA, ...
    movb    dh, #24              ! dh = row 24
    cmp     ax, #2               ! At least EGA?
    jb      is25                 ! Otherwise 25 rows
    push    ds
    xor     ax, ax                ! Vector & BIOS data segments
    mov     ds, ax
    movb    dh, 0x0484           ! Number of rows on display minus one
    pop     ds

```

is25:

```

    xorb    dl, dl                ! dl = column 0
    xorb    bh, bh                ! Page 0
    movb    ah, #0x02            ! Set cursor position
    int     0x10

```

```

    movb    dev_state, #-1       ! Minix may have upset the disks, must reset.
    call    serial_init          ! Likewise with our serial console

```

```

    call    _getprocessor
    cmp     ax, #286
    jb      noclock
    xorb    al, al

```

tryclk:

```

    decb    al
    jz      noclock
    movb    ah, #0x02            ! Get real-time clock time (from CMOS clock)
    int     0x1A
    jc      tryclk                ! Carry set, not running or being updated
    movb    al, ch                ! ch = hour in BCD
    call    bcd                  ! al = (al >> 4) * 10 + (al & 0x0F)
    mulb    c60                  ! 60 minutes in an hour
    mov     bx, ax                ! bx = hour * 60
    movb    al, cl                ! cl = minutes in BCD
    call    bcd
    add     bx, ax                ! bx = hour * 60 + minutes
    movb    al, dh                ! dh = seconds in BCD
    call    bcd
    xchg    ax, bx                ! ax = hour * 60 + minutes, bx = seconds
    mul     c60                  ! dx-ax = (hour * 60 + minutes) * 60
    add     bx, ax
    adc     dx, #0                ! dx-bx = seconds since midnight
    mov     ax, dx
    mul     c19663
    xchg    ax, bx

```

```

    mul    c19663
    add    dx, bx          ! dx-ax = dx-bx * (0x1800B0 / (2*2*2*2*5))
    mov    cx, ax          ! (0x1800B0 = ticks per day of BIOS clock)
    mov    ax, dx
    xor    dx, dx
    div    c1080
    xchg   ax, cx
    div    c1080          ! cx-ax = dx-ax / (24*60*60 / (2*2*2*2*5))
    mov    dx, ax          ! cx-dx = ticks since midnight
    movb   ah, #0x01      ! Set system time
    int    0x1A
noclock:
    pop    bp
    ret                      ! Return to monitor as if nothing much happened

! Transform BCD number in al to a regular value in ax.
bcd:    movb   ah, al
        shrb   ah, #4
        andb   al, #0x0F
        .data1 0xD5,10 ! aad          ! ax = (al >> 4) * 10 + (al & 0x0F)
        ret                      ! (BUG: assembler messes up aad & aam!)

! Support function for Minix-386 to make a BIOS int 13 call (disk I/O).
bios13:
    mov     bp, sp
    call    prot2real
    sti                      ! Enable interrupts

    mov     ax, 8(bp)        ! Load parameters
    mov     bx, 10(bp)
    mov     cx, 12(bp)
    mov     dx, 14(bp)
    mov     es, 16(bp)
    int     0x13             ! Make the BIOS call
    mov     8(bp), ax        ! Save results
    mov     10(bp), bx
    mov     12(bp), cx
    mov     14(bp), dx
    mov     16(bp), es

    cli                      ! Disable interrupts
    call    real2prot
    mov     ax, #DS_SELECTOR ! Kernel data
    mov     ds, ax
    .data1  032
    retf                     ! Return to the kernel

! Support function for Minix-386 to make an 8086 interrupt call.
int86:
    mov     bp, sp
    call    prot2real

    .data1  032
    xor     ax, ax
    mov     es, ax          ! Vector & BIOS data segments
    .data1  032
    eseg mov 0x046C, ax      ! Clear BIOS clock tick counter

    sti                      ! Enable interrupts

    movb    al, #0xCD        ! INT instruction
    movb    ah, 8(bp)        ! Interrupt number?
    testb   ah, ah
    jnz     0f               ! Nonzero if INT, otherwise far call
    push    cs
    push    #intret+2        ! Far return address
    .data1  032
    push    12(bp)           ! Far driver address
    mov     ax, #0x90CB      ! RETF; NOP

0:
    cseg    cmp     ax, intret ! Needs to be changed?
    je     0f               ! If not then avoid a huge I-cache stall
    cseg    mov     intret, ax ! Patch 'INT n' or 'RETF; NOP' into code

```

```

0:      jmp      .+2                ! Clear instruction queue

      mov      ds, 16(bp)          ! Load parameters
      mov      es, 18(bp)
      .data1   o32
      mov      ax, 20(bp)
      .data1   o32
      mov      bx, 24(bp)
      .data1   o32
      mov      cx, 28(bp)
      .data1   o32
      mov      dx, 32(bp)
      .data1   o32
      mov      si, 36(bp)
      .data1   o32
      mov      di, 40(bp)
      .data1   o32
      mov      bp, 44(bp)

intret: int      0xFF              ! Do the interrupt or far call

      .data1   o32                ! Save results
      push     bp
      .data1   o32
      pushf
      mov      bp, sp
      .data1   o32
      pop      8+8(bp)             ! eflags
      mov      8+16(bp), ds
      mov      8+18(bp), es
      .data1   o32
      mov      8+20(bp), ax
      .data1   o32
      mov      8+24(bp), bx
      .data1   o32
      mov      8+28(bp), cx
      .data1   o32
      mov      8+32(bp), dx
      .data1   o32
      mov      8+36(bp), si
      .data1   o32
      mov      8+40(bp), di
      .data1   o32
      pop      8+44(bp)            ! ebp

      cli                          ! Disable interrupts

      xor      ax, ax
      mov      ds, ax              ! Vector & BIOS data segments
      .data1   o32
      mov      cx, 0x046C          ! Collect lost clock ticks in ecx

      mov      ax, ss
      mov      ds, ax              ! Restore monitor ds
      call     real2prot
      mov      ax, #DS_SELECTOR    ! Kernel data
      mov      ds, ax
      .data1   o32
      retf                          ! Return to the kernel

! Switch from real to protected mode.
real2prot:
      movb     ah, #0x02           ! Code for A20 enable
      call     gate_A20

      lgdt     p_gdt_desc          ! Global descriptor table
      .data1   o32
      mov      ax, pdir            ! Load page directory base register
      .data1   0x0F,0x22,0xD8      ! mov cr3, eax
      .data1   0x0F,0x20,0xC0      ! mov eax, cr0
      .data1   o32
      xchg     ax, msr              ! Exchange real mode msr for protected mode msr
      .data1   0x0F,0x22,0xC0      ! mov cr0, eax
      jmpf     cs_prot, MCS_SELECTOR ! Set code segment selector

```

```

cs_prot:
    mov     ax, #SS_SELECTOR ! Set data selectors
    mov     ds, ax
    mov     es, ax
    mov     ss, ax
    ret

! Switch from protected to real mode.
prot2real:
    lidt     p_idt_desc      ! Real mode interrupt vectors
    .data1   0x0F,0x20,0xD8 ! mov     eax, cr3
    .data1   0x32
    mov     pdbr, ax        ! Save page directory base register
    .data1   0x0F,0x20,0xC0 ! mov     eax, cr0
    .data1   0x32
    xchg     ax, msw         ! Exchange protected mode msb for real mode msb
    .data1   0x0F,0x22,0xC0 ! mov     cr0, eax
    jmpf     cs_real, 0xDEAD ! Reload cs register

cs_real:
    mov     ax, #0xBEEF

ds_real:
    mov     ds, ax          ! Reload data segment registers
    mov     es, ax
    mov     ss, ax

    xorb     ah, ah         ! Code for A20 disable
    ! jmp     gate_A20

! Enable (ah = 0x02) or disable (ah = 0x00) the A20 address line.
gate_A20:
    cmp     bus, #2         ! PS/2 bus?
    je      gate_PS_A20
    call    kb_wait
    movb     al, #0xD1      ! Tell keyboard that a command is coming
    outb     0x64
    call    kb_wait
    movb     al, #0xDD      ! 0xDD = A20 disable code if ah = 0x00
    orb      al, ah         ! 0xDF = A20 enable code if ah = 0x02
    outb     0x60
    call    kb_wait
    movb     al, #0xFF      ! Pulse output port
    outb     0x64
    call    kb_wait        ! Wait for the A20 line to settle down
    ret

kb_wait:
    inb      0x64
    testb    al, #0x02      ! Keyboard input buffer full?
    jnz     kb_wait        ! If so, wait
    ret

gate_PS_A20:
    ! The PS/2 can twiddle A20 using port A
    inb      0x92          ! Read port A
    andb     al, #0xFD
    orb      al, ah        ! Set A20 bit to the required state
    outb     0x92          ! Write port A
    jmp      .+2           ! Small delay
A20ok:
    inb      0x92          ! Check port A
    andb     al, #0x02
    cmpb     al, ah        ! A20 line settled down to the new state?
    jne     A20ok         ! If not then wait
    ret

! void int15(bios_env_t *ep)
! Do an "INT 15" call, primarily for APM (Power Management).
#define _int15
_int15:
    push     si            ! Save callee-save register si
    mov     si, sp
    mov     si, 4(si)      ! ep
    mov     ax, (si)       ! ep→ax
    mov     bx, 2(si)      ! ep→bx
    mov     cx, 4(si)      ! ep→cx
    int     0x15           ! INT 0x15 BIOS call
    pushf                    ! Save flags

```



```

    mov     (si), ax      ! ep→ax
    mov     2(si), bx     ! ep→bx
    mov     4(si), cx     ! ep→cx
    pop     6(si)         ! ep→flags
    pop     si            ! Restore
    ret

! void scan_keyboard(void)
!   Read keyboard character. Needs to be done in case one is waiting.
.define _scan_keyboard
_scan_keyboard:
    inb     0x60
    inb     0x61
    movb    ah, al
    orb     al, #0x80
    outb    0x61
    movb    al, ah
    outb    0x61
    ret

.data
    .ascii  "(null)\0"      ! Just in case someone follows a null pointer
    .align  2
c60:      .data2  60         ! Constants for MUL and DIV
c1024:    .data2  1024
c1080:    .data2  1080
c19663:   .data2  19663

! Global descriptor tables.
    UNSET   = 0             ! Must be computed

! For "Extended Memory Block Move".
x_gdt:
x_null_desc:
    ! Null descriptor
    .data2  0x0000, 0x0000
    .data1  0x00, 0x00, 0x00, 0x00
x_gdt_desc:
    ! Descriptor for this descriptor table
    .data2  6*8-1, UNSET
    .data1  UNSET, 0x00, 0x00, 0x00
x_src_desc:
    ! Source segment descriptor
    .data2  0xFFFF, UNSET
    .data1  UNSET, 0x92, 0x00, 0x00
x_dst_desc:
    ! Destination segment descriptor
    .data2  0xFFFF, UNSET
    .data1  UNSET, 0x92, 0x00, 0x00
x_bios_desc:
    ! BIOS segment descriptor (scratch for int 0x15)
    .data2  UNSET, UNSET
    .data1  UNSET, UNSET, UNSET, UNSET
x_ss_desc:
    ! BIOS stack segment descriptor (scratch for int 0x15)
    .data2  UNSET, UNSET
    .data1  UNSET, UNSET, UNSET, UNSET

! Protected mode descriptor table.
p_gdt:
p_null_desc:
    ! Null descriptor
    .data2  0x0000, 0x0000
    .data1  0x00, 0x00, 0x00, 0x00
p_gdt_desc:
    ! Descriptor for this descriptor table
    .data2  8*8-1, UNSET
    .data1  UNSET, 0x00, 0x00, 0x00
p_idt_desc:
    ! Real mode interrupt descriptor table descriptor
    .data2  0x03FF, 0x0000
    .data1  0x00, 0x00, 0x00, 0x00
p_ds_desc:
    ! Kernel data segment descriptor (4 Gb flat)

```

```
.data2 0xFFFF, UNSET
.data1 UNSET, 0x92, 0xCF, 0x00
p_es_desc:
! Physical memory descriptor (4 Gb flat)
.data2 0xFFFF, 0x0000
.data1 0x00, 0x92, 0xCF, 0x00
p_ss_desc:
! Monitor data segment descriptor (64 kb flat)
.data2 0xFFFF, UNSET
.data1 UNSET, 0x92, 0x00, 0x00
p_cs_desc:
! Kernel code segment descriptor (4 Gb flat)
.data2 0xFFFF, UNSET
.data1 UNSET, 0x9A, 0xCF, 0x00
p_mcs_desc:
! Monitor code segment descriptor (64 kb flat)
.data2 0xFFFF, UNSET
.data1 UNSET, 0x9A, 0x00, 0x00

.bss
.comm old_vid_mode, 2 ! Video mode at startup
.comm cur_vid_mode, 2 ! Current video mode
.comm dev_state, 2 ! Device state: reset (-1), closed (0), open (1)
.comm sectors, 2 ! # sectors of current device
.comm secspcyl, 2 ! (Sectors * heads) of current device
.comm msw, 4 ! Saved machine status word (cr0)
.comm pdbr, 4 ! Saved page directory base register (cr3)
.comm escape, 2 ! Escape typed?
.comm bus, 2 ! Saved return value of _get_bus
.comm unchar, 2 ! Char returned by ungetch(c)
.comm line, 2 ! Serial line I/O port to copy console I/O to.
```

```

/*      bootimage.c - Load an image and start it.      Author: Kees J. Bot
*                                                    19 Jan 1992
*/
#define BIOS 1 /* Can only be used under the BIOS. */
#define nil 0
#define _POSIX_SOURCE 1
#define _MINIX 1
#include <stddef.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <string.h>
#include <errno.h>
#include <a.out.h>
#include <minix/config.h>
#include <minix/const.h>
#include <minix/type.h>
#include <minix/syslib.h>
#include <kernel/const.h>
#include <kernel/type.h>
#include <ibm/partition.h>
#include "rawfs.h"
#include "image.h"
#include "boot.h"

static int block_size = 0;

#define click_shift      clck_shft      /* 7 char clash with click_size. */

/* Some kernels have extra features: */
#define K_I386 0x0001 /* Make the 386 transition before you call me. */
#define K_CLAIM 0x0002 /* I will acquire my own bss pages, thank you. */
#define K_CHMEM 0x0004 /* This kernel listens to chmem for its stack size. */
#define K_HIGH 0x0008 /* Load mm, fs, etc. in extended memory. */
#define K_HDR 0x0010 /* No need to patch sizes, kernel uses the headers. */
#define K_RET 0x0020 /* Returns to the monitor on reboot. */
#define K_INT86 0x0040 /* Requires generic INT support. */
#define K_MEML 0x0080 /* Pass a list of free memory. */
#define K_BRET 0x0100 /* New monitor code on shutdown in boot parameters. */
#define K_ALL 0x01FF /* All feature bits this monitor supports. */

/* Data about the different processes. */

#define PROCESS_MAX 16 /* Must match the space in kernel/mpx.x */
#define KERNEL 0 /* The first process is the kernel. */
#define FS 2 /* The third must be fs. */

struct process { /* Per-process memory addresses. */
    u32_t entry; /* Entry point. */
    u32_t cs; /* Code segment. */
    u32_t ds; /* Data segment. */
    u32_t data; /* To access the data segment. */
    u32_t end; /* End of this process, size = (end - cs). */
} process[PROCESS_MAX];
int n_procs; /* Number of processes. */

/* Magic numbers in process' data space. */
#define MAGIC_OFF 0 /* Offset of magic # in data seg. */
#define CLICK_OFF 2 /* Offset in kernel text to click_shift. */
#define FLAGS_OFF 4 /* Offset in kernel text to flags. */
#define KERNEL_D_MAGIC 0x526F /* Kernel magic number. */

/* Offsets of sizes to be patched into kernel and fs. */
#define P_SIZ_OFF 0 /* Process' sizes into kernel data. */
#define P_INIT_OFF 4 /* Init cs & sizes into fs data. */

#define between(a, c, z) ((unsigned) ((c) - (a)) <= ((z) - (a)))

void pretty_image(char *image)
/* Pretty print the name of the image to load. Translate '/' and '_' to

```

```

* space, first letter goes uppercase. An 'r' before a digit prints as
* 'revision'. E.g. 'minix/1.6.16r10' -> 'Minix 1.6.16 revision 10'.
* The idea is that the part before the 'r' is the official Minix release
* and after the 'r' you can put version numbers for your own changes.
*/
{
    int up= 0, c;

    while ((c= *image++) != 0) {
        if (c == '/' || c == '_') c= ' ';

        if (c == 'r' && between('0', *image, '9')) {
            printf(" revision ");
            continue;
        }
        if (!up && between('a', c, 'z')) c= c - 'a' + 'A';

        if (between('A', c, 'Z')) up= 1;

        putchar(c);
    }
}

void raw_clear(u32_t addr, u32_t count)
/* Clear "count" bytes at absolute address "addr". */
{
    static char zeros[128];
    u32_t dst;
    u32_t zct;

    zct= sizeof(zeros);
    if (zct > count) zct= count;
    raw_copy(addr, mon2abs(&zeros), zct);
    count-= zct;

    while (count > 0) {
        dst= addr + zct;
        if (zct > count) zct= count;
        raw_copy(dst, addr, zct);
        count-= zct;
        zct*= 2;
    }
}

/* Align a to a multiple of n (a power of 2): */
#define align(a, n) (((u32_t)(a) + ((u32_t)(n) - 1)) & ~((u32_t)(n) - 1))
unsigned click_shift;
unsigned click_size; /* click_size = Smallest kernel memory object. */
unsigned k_flags; /* Not all kernels are created equal. */
u32_t reboot_code; /* Obsolete reboot code return pointer. */

int params2params(char *params, size_t psize)
/* Repackage the environment settings for the kernel. */
{
    size_t i, n;
    environment *e;
    char *name, *value;
    dev_t dev;

    i= 0;
    for (e= env; e != nil; e= e->next) {
        name= e->name;
        value= e->value;

        if (!(e->flags & E_VAR)) continue;

        if (e->flags & E_DEV) {
            if ((dev= name2dev(value)) == -1) return 0;
            value= ul2a10((u16_t) dev);
        }

        n= i + strlen(name) + 1 + strlen(value) + 1;
        if (n < psize) {
            strcpy(params + i, name);

```

```

        strcat(params + i, "=");
        strcat(params + i, value);
    }
    i = n;
}

if (!(k_flags & K_MEML)) {
    /* Require old memory size variables. */

    value = ul2a10((mem[0].base + mem[0].size) / 1024);
    n = i + 7 + 1 + strlen(value) + 1;
    if (n < psize) {
        strcpy(params + i, "memsize=");
        strcat(params + i, value);
    }
    i = n;
    value = ul2a10(mem[1].size / 1024);
    n = i + 7 + 1 + strlen(value) + 1;
    if (n < psize) {
        strcpy(params + i, "emssize=");
        strcat(params + i, value);
    }
    i = n;
}

if (i >= psize) {
    printf("Too many boot parameters\n");
    return 0;
}
params[i] = 0; /* End marked with empty string. */
return 1;
}

void patch_sizes(void)
/* Patch sizes of each process into kernel data space, kernel ds into kernel
 * text space, and sizes of init into data space of fs. All the patched
 * numbers are based on the kernel click size, not hardware segments.
 */
{
    ul6_t text_size, data_size;
    int i;
    struct process *procp, *initp;
    u32_t doff;

    if (k_flags & K_HDR) return; /* Uses the headers. */

    /* Patch text and data sizes of the processes into kernel data space.
     */
    doff = process[KERNEL].data + P_SIZ_OFF;

    for (i = 0; i < n_procs; i++) {
        procp = &process[i];
        text_size = (procp->ds - procp->cs) >> click_shift;
        data_size = (procp->end - procp->ds) >> click_shift;

        /* Two words per process, the text and data size: */
        put_word(doff, text_size); doff += 2;
        put_word(doff, data_size); doff += 2;

        initp = procp; /* The last process must be init. */
    }

    if (k_flags & (K_HIGH|K_MEML)) return; /* Doesn't need FS patching. */

    /* Patch cs and sizes of init into fs data. */
    put_word(process[FS].data + P_INIT_OFF+0, initp->cs >> click_shift);
    put_word(process[FS].data + P_INIT_OFF+2, text_size);
    put_word(process[FS].data + P_INIT_OFF+4, data_size);
}

int selected(char *name)
/* True iff name has no label or the proper label. */
{
    char *colon, *label;

```

```

    int cmp;

    if ((colon= strchr(name, ':')) == nil) return 1;
    if ((label= b_value("label")) == nil) return 1;

    *colon= 0;
    cmp= strcmp(label, name);
    *colon= ':';
    return cmp == 0;
}

u32_t proc_size(struct image_header *hdr)
/* Return the size of a process in sectors as found in an image. */
{
    u32_t len= hdr->process.a_text;

    if (hdr->process.a_flags & A_PAL) len+= hdr->process.a_hdrlen;
    if (hdr->process.a_flags & A_SEP) len= align(len, SECTOR_SIZE);
    len= align(len + hdr->process.a_data, SECTOR_SIZE);

    return len >> SECTOR_SHIFT;
}

off_t image_off, image_size;
u32_t (*vir2sec)(u32_t vsec); /* Where is a sector on disk? */

u32_t file_vir2sec(u32_t vsec)
/* Translate a virtual sector number to an absolute disk sector. */
{
    off_t blk;

    if(!block_size) { errno = 0; return -1; }

    if ((blk= r_vir2abs(vsec / RATIO(block_size))) == -1) {
        errno= EIO;
        return -1;
    }
    return blk == 0 ? 0 : lowsec + blk * RATIO(block_size) + vsec % RATIO(block_size)
;
}

u32_t flat_vir2sec(u32_t vsec)
/* Simply add an absolute sector offset to vsec. */
{
    return lowsec + image_off + vsec;
}

char *get_sector(u32_t vsec)
/* Read a sector "vsec" from the image into memory and return its address.
 * Return nil on error. (This routine tries to read an entire track, so
 * the next request is usually satisfied from the track buffer.)
 */
{
    u32_t sec;
    int r;
#define SECBUFS 16
    static char buf[SECBUFS * SECTOR_SIZE];
    static size_t count; /* Number of sectors in the buffer. */
    static u32_t bufsec; /* First Sector now in the buffer. */

    if (vsec == 0) count= 0; /* First sector; initialize. */

    if ((sec= (*vir2sec)(vsec)) == -1) return nil;

    if (sec == 0) {
        /* A hole. */
        count= 0;
        memset(buf, 0, SECTOR_SIZE);
        return buf;
    }

    /* Can we return a sector from the buffer? */
    if ((sec - bufsec) < count) {
        return buf + ((size_t) (sec - bufsec) << SECTOR_SHIFT);
    }

```

```

    }

    /* Not in the buffer. */
    count= 0;
    bufsec= sec;

    /* Read a whole track if possible. */
    while (++count < SECBUFS && !dev_boundary(bufsec + count)) {
        vsec++;
        if ((sec= (*vir2sec)(vsec)) == -1) break;

        /* Consecutive? */
        if (sec != bufsec + count) break;
    }

    /* Actually read the sectors. */
    if ((r= readsectors(mon2abs(buf), bufsec, count)) != 0) {
        readerr(bufsec, r);
        count= 0;
        errno= 0;
        return nil;
    }
    return buf;
}

int get_clickshift(u32_t ksec, struct image_header *hdr)
/* Get the click shift and special flags from kernel text. */
{
    char *textp;

    if ((textp= get_sector(ksec)) == nil) return 0;

    if (hdr->process.a_flags & A_PAL) textp+= hdr->process.a_hdrlen;
    click_shift= * (u16_t *) (textp + CLICK_OFF);
    k_flags= * (u16_t *) (textp + FLAGS_OFF);

    if ((k_flags & ~K_ALL) != 0) {
        printf("%s requires features this monitor doesn't offer\n",
            hdr->name);
        return 0;
    }

    if (click_shift < HCLICK_SHIFT || click_shift > 16) {
        printf("%s click size is bad\n", hdr->name);
        errno= 0;
        return 0;
    }

    click_size= 1 << click_shift;

    return 1;
}

int get_segment(u32_t *vsec, long *size, u32_t *addr, u32_t limit)
/* Read *size bytes starting at virtual sector *vsec to memory at *addr. */
{
    char *buf;
    size_t cnt, n;

    cnt= 0;
    while (*size > 0) {
        if (cnt == 0) {
            if ((buf= get_sector((*vsec)++)) == nil) return 0;
            cnt= SECTOR_SIZE;
        }
        if (*addr + click_size > limit) { errno= ENOMEM; return 0; }
        n= click_size;
        if (n > cnt) n= cnt;
        raw_copy(addr, mon2abs(buf), n);
        *addr+= n;
        *size-= n;
        buf+= n;
        cnt-= n;
    }
}

```

```

    /* Zero extend to a click. */
    n= align(*addr, click_size) - *addr;
    raw_clear(*addr, n);
    *addr+= n;
    *size-= n;
    return 1;
}

void exec_image(char *image)
/* Get a Minix image into core, patch it up and execute. */
{
    char *delayvalue;
    int i;
    struct image_header hdr;
    char *buf;
    u32_t vsec, addr, limit, aout, n;
    struct process *procp; /* Process under construction. */
    long a_text, a_data, a_bss, a_stack;
    int banner= 0;
    long processor= a2l(b_value("processor"));
    u16_t mode;
    char *console;
    char params[SECTOR_SIZE];
    extern char *sbrk(int);

    /* The stack is pretty deep here, so check if heap and stack collide. */
    (void) sbrk(0);

    printf("\nLoading ");
    pretty_image(image);
    printf("\n\n");

    vsec= 0; /* Load this sector from image next. */
    addr= mem[0].base; /* Into this memory block. */
    limit= mem[0].base + mem[0].size;
    if (limit > caddr) limit= caddr;

    /* Allocate and clear the area where the headers will be placed. */
    aout = (limit -= PROCESS_MAX * A_MINHDR);

    /* Clear the area where the headers will be placed. */
    raw_clear(aout, PROCESS_MAX * A_MINHDR);

    /* Read the many different processes: */
    for (i= 0; vsec < image_size; i++) {
        if (i == PROCESS_MAX) {
            printf("There are more then %d programs in %s\n",
                PROCESS_MAX, image);
            errno= 0;
            return;
        }
        procp= &process[i];

        /* Read header. */
        for (;;) {
            if ((buf= get_sector(vsec++)) == nil) return;

            memcpy(&hdr, buf, sizeof(hdr));

            if (BADMAG(hdr.process)) { errno= ENOEXEC; return; }

            /* Check the optional label on the process. */
            if (selected(hdr.name)) break;

            /* Bad label, skip this process. */
            vsec+= proc_size(&hdr);
        }

        /* Sanity check: an 8086 can't run a 386 kernel. */
        if (hdr.process.a_cpu == A_I80386 && processor < 386) {
            printf("You can't run a 386 kernel on this 80%ld\n",
                processor);
            errno= 0;
        }
    }
}

```



```

        return;
    }

    /* Get the click shift from the kernel text segment. */
    if (i == KERNEL) {
        if (!get_clickshift(vsec, &hdr)) return;
        addr= align(addr, click_size);
    }

    /* Save a copy of the header for the kernel, with a_syms
     * misused as the address where the process is loaded at.
     */
    hdr.process.a_syms= addr;
    raw_copy(aout + i * A_MINHDR, mon2abs(&hdr.process), A_MINHDR);

    if (!banner) {
        printf("  cs  ds  text  data  bss");
        if (k_flags & K_CHMEM) printf("  stack");
        putchar('\n');
        banner= 1;
    }

    /* Segment sizes. */
    a_text= hdr.process.a_text;
    a_data= hdr.process.a_data;
    a_bss= hdr.process.a_bss;
    if (k_flags & K_CHMEM) {
        a_stack= hdr.process.a_total - a_data - a_bss;
        if (!(hdr.process.a_flags & A_SEP)) a_stack-= a_text;
    } else {
        a_stack= 0;
    }

    /* Collect info about the process to be. */
    procp->cs= addr;

    /* Process may be page aligned so that the text segment contains
     * the header, or have an unmapped zero page against vaxisms.
     */
    procp->entry= hdr.process.a_entry;
    if (hdr.process.a_flags & A_PAL) a_text+= hdr.process.a_hdrlen;
    if (hdr.process.a_flags & A_UZP) procp->cs-= click_size;

    /* Separate I&D: two segments. Common I&D: only one. */
    if (hdr.process.a_flags & A_SEP) {
        /* Read the text segment. */
        if (!get_segment(&vsec, &a_text, &addr, limit)) return;

        /* The data segment follows. */
        procp->ds= addr;
        if (hdr.process.a_flags & A_UZP) procp->ds-= click_size;
        procp->data= addr;
    } else {
        /* Add text to data to form one segment. */
        procp->data= addr + a_text;
        procp->ds= procp->cs;
        a_data+= a_text;
    }

    /* Read the data segment. */
    if (!get_segment(&vsec, &a_data, &addr, limit)) return;

    /* Make space for bss and stack unless... */
    if (i != KERNEL && (k_flags & K_CLAIM)) a_bss= a_stack= 0;

    printf("%07lx %07lx %8ld %8ld %8ld",
        procp->cs, procp->ds,
        hdr.process.a_text, hdr.process.a_data,
        hdr.process.a_bss
    );
    if (k_flags & K_CHMEM) printf(" %8ld", a_stack);

    printf(" %s\n", hdr.name);

```

```

    /* Note that a_data may be negative now, but we can look at it
     * as -a_data bss bytes.
     */

    /* Compute the number of bss clicks left. */
    a_bss+= a_data;
    n= align(a_bss, click_size);
    a_bss-= n;

    /* Zero out bss. */
    if (addr + n > limit) { errno= ENOMEM; return; }
    raw_clear(addr, n);
    addr+= n;

    /* And the number of stack clicks. */
    a_stack+= a_bss;
    n= align(a_stack, click_size);
    a_stack-= n;

    /* Add space for the stack. */
    addr+= n;

    /* Process endpoint. */
    procp->end= addr;

    if (i == 0 && (k_flags & K_HIGH)) {
        /* Load the rest in extended memory. */
        addr= mem[1].base;
        limit= mem[1].base + mem[1].size;
    }
}

if ((n_procs= i) == 0) {
    printf("There are no programs in %s\n", image);
    errno= 0;
    return;
}

/* Check the kernel magic number. */
if (get_word(process[KERNEL].data + MAGIC_OFF) != KERNEL_D_MAGIC) {
    printf("Kernel magic number is incorrect\n");
    errno= 0;
    return;
}

/* Patch sizes, etc. into kernel data. */
patch_sizes();

#ifdef !DOS
    if (!(k_flags & K_MEML)) {
        /* Copy the a.out headers to the old place. */
        raw_copy(HEADERPOS, aout, PROCESS_MAX * A_MINHDR);
    }
#endif

/* Do delay if wanted. */
if((delayvalue = b_value("bootdelay")) != nil & > 0) {
    delay(delayvalue);
}

/* Run the trailer function just before starting Minix. */
if (!run_trailer()) { errno= 0; return; }

/* Translate the boot parameters to what Minix likes best. */
if (!params2params(params, sizeof(params))) { errno= 0; return; }

/* Set the video to the required mode. */
if ((console= b_value("console")) == nil || (mode= a2x(console)) == 0) {
    mode= strcmp(b_value("chrome"), "color") == 0 ? COLOR_MODE :
                                                    MONO_MODE;
}
set_mode(mode);

/* Close the disk. */

```

```

(void) dev_close();

/* Minix. */
minix(process[KERNEL].entry, process[KERNEL].cs,
        process[KERNEL].ds, params, sizeof(params), aout);

if (!(k_flags & K_BRET)) {
    extern u32_t reboot_code;
    raw_copy(mon2abs(params), reboot_code, sizeof(params));
}
parse_code(params);

/* Return from Minix. Things may have changed, so assume nothing. */
fsok = -1;
errno = 0;

/* Read leftover character, if any. */
scan_keyboard();
}

ino_t latest_version(char *version, struct stat *stp)
/* Recursively read the current directory, selecting the newest image on
 * the way up. (One can't use r_stat while reading a directory.)
 */
{
    char name[NAME_MAX + 1];
    ino_t ino, newest;
    time_t mtime;

    if ((ino = r_readdir(name)) == 0) { stp->st_mtime = 0; return 0; }

    newest = latest_version(version, stp);
    mtime = stp->st_mtime;
    r_stat(ino, stp);

    if (S_ISREG(stp->st_mode) && stp->st_mtime > mtime) {
        newest = ino;
        strcpy(version, name);
    } else {
        stp->st_mtime = mtime;
    }
    return newest;
}

char *select_image(char *image)
/* Look image up on the filesystem, if it is a file then we're done, but
 * if its a directory then we want the newest file in that directory. If
 * it doesn't exist at all, then see if it is 'number:number' and get the
 * image from that absolute offset off the disk.
 */
{
    ino_t image_ino;
    struct stat st;

    image = strcpy(malloc((strlen(image) + 1 + NAME_MAX + 1)
                          * sizeof(char)), image);

    fsok = r_super(&block_size) != 0;
    if (!fsok || (image_ino = r_lookup(ROOT_INO, image)) == 0) {
        char *size;

        if (numprefix(image, &size) && *size++ == ':'
            && numeric(size)) {
            vir2sec = flat_vir2sec;
            image_off = a2l(image);
            image_size = a2l(size);
            strcpy(image, "Minix");
            return image;
        }
        if (!fsok)
            printf("No image selected\n");
        else
            printf("Can't load %s: %s\n", image, unix_err(errno));
        goto bail_out;
    }
}

```

```

    }

    r_stat(image_ino, &st);
    if (!S_ISREG(st.st_mode)) {
        char *version= image + strlen(image);
        char dots[NAME_MAX + 1];

        if (!S_ISDIR(st.st_mode)) {
            printf("%s:%s\n", image, unix_err(ENOTDIR));
            goto bail_out;
        }
        (void) r_readdir(dots);
        (void) r_readdir(dots); /* "." & ".." */
        *version++= '/';
        *version= 0;
        if ((image_ino= latest_version(version, &st)) == 0) {
            printf("There are no images in %s\n", image);
            goto bail_out;
        }
        r_stat(image_ino, &st);
    }
    vir2sec= file_vir2sec;
    image_size= (st.st_size + SECTOR_SIZE - 1) >> SECTOR_SHIFT;
    return image;
bail_out:
    free(image);
    return nil;
}

void bootminix(void)
/* Load Minix and run it. (Given the size of this program it is surprising
 * that it ever gets to that.)
 */
{
    char *image;

    if ((image= select_image(b_value("image")) == nil) return;

    exec_image(image);

    switch (errno) {
    case ENOEXEC:
        printf("%s contains a bad program header\n", image);
        break;
    case ENOMEM:
        printf("Not enough memory to load %s\n", image);
        break;
    case EIO:
        printf("Unsuspected EOF on %s\n", image);
    case 0:
        /* No error or error already reported. */
    }
    free(image);
}

/*
 * $PchId: bootimage.c,v 1.10 2002/02/27 19:39:09 philip Exp $
 */

```

```

!      Doshead.s - DOS & BIOS support for boot.c      Author: Kees J. Bot
!
!
! This file contains the startup and low level support for the secondary
! boot program. It contains functions for disk, tty and keyboard I/O,
! copying memory to arbitrary locations, etc.
!
! This runs under MS-DOS as a .COM file. A .COM file is what Minix calls
! a common I&D executable, except that the first 256 bytes contains DOS
! thingies.
!
.sect .text; .sect .rom; .sect .data; .sect .bss

      K_I386      =      0x0001 ! Call Minix in 386 mode
      STACK       =      16384 ! Number of bytes for the stack

      DS_SELECTOR =        3*8 ! Kernel data selector
      ES_SELECTOR =        4*8 ! Flat 4 Gb
      SS_SELECTOR =        5*8 ! Monitor stack
      CS_SELECTOR =        6*8 ! Kernel code
      MCS_SELECTOR=        7*8 ! Monitor code

      ESC         =        0x1B ! Escape character

! Imported variables and functions:
.extern _caddr, _daddr, _runsize, _edata, _end ! Runtime environment
.extern _k_flags ! Special kernel flags
.extern _mem ! Free memory list
.extern _vdisk ! Name of the virtual disk

.sect .text

.usel6 ! Tell 386 assembler we're in 16-bit mode

.define _PSP
_PSP:
    .space 256 ! Program Segment Prefix

dosboot:
    cld ! C compiler wants UP
    xor ax, ax ! Zero
    mov di, _edata ! Start of bss is at end of data
    mov cx, _end ! End of bss (begin of heap)
    sub cx, di ! Number of bss bytes
    shr cx, 1 ! Number of words
    rep stos ! Clear bss
    cmp sp, _end+STACK
    jb 0f
    mov sp, _end+STACK ! "chmem" to 16 kb
0:

! Are we called with the /U option?
    movb cl, (_PSP+0x80) ! Argument byte count
    xorb ch, ch
    mov bx, _PSP+0x81 ! Argument string
0:
    jcxz notuflag
    cmpb (bx), 0x20 ! Whitespace?
    ja 1f
    inc bx
    dec cx
    jmp 0b
1:
    cmp cx, 2 ! '/U' is two bytes
    jne notuflag
    cmpb (bx), 0x2F ! '/'?
    jne notuflag
    movb al, 1(bx)
    andb al, ~0x20 ! Ignore case
    cmpb al, 0x55 ! 'U'?
    jne notuflag
    jmp keepumb ! Go grab an UMB
notuflag:

! Remember the current video mode for restoration on exit.
    movb ah, 0x0F ! Get current video mode

```

```

    int     0x10
    andb    al, 0x7F      ! Mask off bit 7 (no blanking)
    movb    (old_vid_mode), al
    movb    (cur_vid_mode), al

! We require at least MS-DOS 3.0.
    mov     ax, 0x3000    ! Get DOS version
    int     0x21
    cmpb    al, 3        ! DOS 3.0+ ?
    jae     dosok
    push    tellbaddos
    call    _printf
    jmp     quit
.sect      .rom
tellbaddos: .ascii  "MS-DOS 3.0 or better required\n\0"
.sect      .text
dosok:

! Find out how much "low" memory there is available, where it starts and
! where it ends.
    mov     di, _mem      ! di = memory list
    mov     ax, _PSP+0x80 ! From PSP:80 to next PSP is ours
    mov     dx, ds
    call    seg2abs
    mov     (di), ax
    mov     2(di), dx     ! mem[0].base = ds * 16 + 0x80
    xor     ax, ax
    mov     dx, (_PSP+2)  ! First in-use segment far above us
    call    seg2abs
    sub     ax, (di)
    sbb     dx, 2(di)     ! Minus base gives size
    mov     4(di), ax
    mov     6(di), dx     ! mem[1].size = free low memory size

! Give C code access to the code segment, data segment and the size of this
! process.
    xor     ax, ax
    mov     dx, cs
    call    seg2abs
    mov     (_caddr+0), ax
    mov     (_caddr+2), dx
    xor     ax, ax
    mov     dx, ds
    call    seg2abs
    mov     (_daddr+0), ax
    mov     (_daddr+2), dx
    mov     ax, sp
    mov     dx, ss       ! End of stack = end of program
    call    seg2abs
    sub     ax, (_caddr+0)
    sbb     dx, (_caddr+2) ! Minus start of our code
    mov     (_runsize+0), ax
    mov     (_runsize+2), dx ! Is our size

! Patch the regular _getprocessor library routine to jump to 'getprocessor',
! that checks if we happen to be in a V8086 straightjacket by returning '86'.
cseg  movb    (_getprocessor+0), 0xE9
      mov     ax, getprocessor
      sub     ax, _getprocessor+3
cseg  mov     (_getprocessor+1), ax

! Grab the largest chunk of extended memory available.
    call    _getprocessor
    cmp     ax, 286      ! Only 286s and above have extended memory
    jb      no_ext
    mov     ax, 0x4300   ! XMS driver check
    int     0x2F
    cmpb    al, 0x80     ! XMS driver exists?
    je      xmsthere
get_ext: ! No driver, so can use all ext memory directly
    call    _getprocessor
    cmp     ax, 486      ! Assume 486s were the first to have >64M
    jb      small_ext    ! (It helps to be paranoid when using the BIOS)
big_ext:

```

```

    mov     ax, 0xE801      ! Code for get memory size for >64M
    int     0x15           ! ax = mem at 1M per 1K, bx = mem at 16M per 64K
    jnc     got_ext

small_ext:
    movb    ah, 0x88       ! Code for get extended memory size
    clc                     ! Carry will stay clear if call exists
    int     0x15           ! Returns size (in K) in ax for AT's
    jc      no_ext
    test    ax, ax         ! An AT with no extended memory?
    jz      no_ext
    xor     bx, bx         ! bx = mem above 16M per 64K = 0

got_ext:
    mov     cx, ax         ! cx = copy of ext mem at 1M
    mov     10(di), 0x0010 ! mem[1].base = 0x00100000 (1M)
    mul     (c1024)
    mov     12(di), ax     ! mem[1].size = "ext mem at 1M" * 1024
    mov     14(di), dx
    test    bx, bx
    jz      no_ext         ! No more ext mem above 16M?
    cmp     cx, 15*1024    ! Chunks adjacent? (precisely 15M at 1M?)
    je      adj_ext
    mov     18(di), 0x0100 ! mem[2].base = 0x01000000 (16M)
    mov     22(di), bx     ! mem[2].size = "ext mem at 16M" * 64K
    jmp     no_ext

adj_ext:
    add     14(di), bx     ! Add ext mem above 16M to mem below 16M

no_ext:
    jmp     gotxms

xmsthere:
    mov     ax, 0x4310      ! Get XMS driver address
    int     0x2F
    mov     (xms_driver+0), bx
    mov     (xms_driver+2), es
    push    ds
    pop     es
    movb    ah, 0x08       ! Query free extended memory
    xorb    bl, bl
    callf   (xms_driver)
    testb   bl, bl
    jnz     xmserr
    push    ax             ! ax = size of largest block in kb
    mul     (c1024)
    mov     12(di), ax
    mov     14(di), dx     ! mem[1].size = ax * 1024
    pop     dx             ! dx = size of largest block in kb
    movb    ah, 0x09       ! Allocate XMS block of size dx
    callf   (xms_driver)
    test    ax, ax
    jz      xmserr
    mov     (xms_handle), dx ! Save handle
    movb    ah, 0x0C       ! Lock XMS block (handle in dx)
    callf   (xms_driver)
    test    ax, ax
    jz      xmserr
    mov     8(di), bx
    mov     10(di), dx     ! mem[1].base = Address of locked block

gotxms:
    ! If we're running in a DOS box then they're might be an Upper Memory Block
    ! we can use. Every little bit helps when in real mode.
    mov     ax, 20(di)
    or      ax, 22(di)     ! Can we use mem[2]?
    jnz     gotumb
    mov     dx, 0xFFFF
    call    getumb
    test    cx, cx         ! Get UMB, dx = segment, cx = length
    jz      gotumb        ! Did we get a block?
    xor     ax, ax         ! dx:ax = memory block
    call    seg2abs
    mov     16(di), ax
    mov     18(di), dx     ! mem[2].base = memory block base
    mov     dx, cx
    xor     ax, ax         ! dx:ax = length of memory block

```

```

        call    seg2abs
        mov     20(di), ax
        mov     22(di), dx                ! mem[2].size = memory block length
gotumb:

! Set up an INT 24 "critical error" handler that returns "fail". This way
! Minix won't suffer from "(A)bort, (R)etry, (I)nfluence with a large hammer?".
        mov     (0x007C), 0x03B0        ! movb al, 0x03 (fail code)
        movb    (0x007E), 0xCF          ! iret
        movb    ah, 0x25                ! Set interrupt vector
        mov     dx, 0x007C              ! ds:dx = ds:0x007C = interrupt handler
        int     0x21

! Time to switch to a higher level language (not much higher)
        call    _boot

! void ..exit(int status)
! Exit the monitor by returning to DOS.
#define _exit, __exit, ___exit        ! Make various compilers happy
_exit:
__exit:
___exit:
        mov     dx, (xms_handle)
        cmp     dx, -1                  ! Is there an ext mem block in use?
        je      nohandle
        movb    ah, 0x0D                ! Unlock extended memory block
        callf   (xms_driver)
        mov     dx, (xms_handle)
        movb    ah, 0x0A                ! Free extended memory block
        callf   (xms_driver)
nohandle:
        call    restore_video
        pop     ax
        pop     ax                      ! Return code in al
        movb    ah, 0x4C                ! Terminate with return code
        int     0x21

quit:                                     ! exit(1)
        movb    al, 1
        push    ax
        call    _exit

xmserr:
        xorb    bh, bh
        push    bx
        push    tellxmserr
        call    _printf
        jmp     quit
.sect .rom
tellxmserr: .ascii "Extended memory problem, error 0x%02x\n\0"
.sect .text

! int getprocessor(void)
! Prefix for the regular _getprocessor call that first checks if we're
! running in a virtual 8086 box.
getprocessor:
        push    sp                      ! Is pushed sp equal to sp?
        pop     ax
        cmp     ax, sp
        jne     gettrueproc            ! If not then it's a plain 8086 or 80186
        .data1  0x0F, 0x01, 0xE0       ! Use old 286 SMSW instruction to get the MSW
        testb   al, 0x01               ! Protected mode enabled?
        jz      gettrueproc            ! If not then a 286 or better in real mode
        mov     ax, 86                 ! Forget fancy tricks, say it's an 8086
        ret
gettrueproc:                            ! Get the true processor type
        push    bp                      ! _getprocessor prologue that is patched over.
        mov     bp, sp
        jmp     _getprocessor+3

! Try to get an Upper Memory Block under MS-DOS 5+. Try to get one up to size
! dx, return segment of UMB found in dx and size in paragraphs in cx.
getumb:
        xor     cx, cx                  ! Initially nothing found

```



```

    mov     ax, 0x3000        ! Get DOS version
    int     0x21
    cmpb    al, 5             ! MS-DOS 5.0 or better?
    jnb     retumb
    mov     ax, 0x544D        ! Get UMB kept by BOOT /U
    int     0x15              ! Returns dx = segment, cx = size
    jc      0f
    cmp     ax, 0x4D54        ! Carry clear and ax byte swapped?
    je      retumb
0:    mov     ax, 0x5802      ! Get UMB link state
    int     0x21
    xorb    ah, ah
    push    ax                ! Save UMB link state
    mov     ax, 0x5803        ! Set UMB link state
    mov     bx, 0x0001        ! Add UMBs to DOS memory chain
    int     0x21
    mov     ax, 0x5800        ! Get memory allocation strategy
    int     0x21
    push    ax                ! Save allocation strategy
    mov     ax, 0x5801        ! Set memory allocation strategy
    mov     bx, 0x0080        ! First fit, try high then low memory
    int     0x21
    movb    ah, 0x48          ! Allocate memory
    mov     bx, dx            ! Number of paragraphs wanted
    int     0x21              ! Fails with bx = size of largest
    jnc     0f                ! Succeeds with ax = allocated block
    test    bx, bx            ! Is there any?
    jz      no_umb
    movb    ah, 0x48          ! Allocate memory
    int     0x21
    jc      no_umb            ! Did we get some?
0:    mov     dx, ax           ! dx = segment
    mov     cx, bx           ! cx = size
no_umb: mov     ax, 0x5801    ! Set memory allocation strategy
    pop     bx               ! bx = saved former strategy
    int     0x21
    mov     ax, 0x5803        ! Set UMB link state
    pop     bx               ! bx = saved former link state
    int     0x21
retumb: ret

! 'BOOT /U' instructs this program to grab the biggest available UMB and to
! sit on it until the next invocation of BOOT wants it back. These shenanigans
! are necessary because Windows 95 keeps all UMBs to itself unless you get hold
! of them first.
    umb = 0x80                ! UMB base and size
    old15 = 0x84              ! Old 15 interrupt vector
    new15 = 0x88              ! New 15 interrupt handler
keepumb:
    mov     ax, 0x544D        ! "Keep UMB" handler already present?
    int     0x15
    jc      0f
    cmp     ax, 0x4D54
    je      exitumb           ! Already present, so quit
0:
    mov     si, new15start
    mov     di, new15
    mov     cx, new15end
    sub     cx, si
    rep     movsb              ! Copy handler into place
    add     di, 15
    movb    cl, 4
    shr     di, cl             ! di = first segment above handler
    mov     cx, cs
    cmp     cx, 0xA000         ! Are we loaded high perchance?
    jnb     nothigh
werhigh:
    add     cx, di
    mov     (umb+0), cx        ! Use my own memory as the UMB to keep
    mov     ax, (_PSP+2)      ! Up to the next in-use segment
    sub     ax, dx             ! ax = size of my free memory
    cmp     ax, 0x1000         ! At least 64K?
    jnb     exitumb           ! Don't bother if less
    mov     (umb+2), 0x1000    ! Size of UMB

```

```

        add     di, 0x1000          ! Keep my code plus 64K when TSR
        jmp     hook15

nothigh:
        mov     dx, 0x1000
        call    getumb             ! Grab an UMB of at most 64K
        cmp     cx, 0x1000         ! Did we get 64K?
        jbe     exitumb           ! Otherwise don't bother
        mov     (umb+0), dx
        mov     (umb+2), cx

hook15:
        mov     ax, 0x3515         ! Get interrupt vector
        int     0x21
        mov     (old15+0), bx
        mov     (old15+2), es      ! Old 15 interrupt
        mov     ax, 0x2515         ! Set interrupt vector
        mov     dx, new15         ! ds:dx = new 15 handler
        int     0x21
        mov     ax, 0x3100         ! Terminate and stay resident
        mov     dx, di            ! dx = di = paragraphs we keep
        int     0x21

exitumb:
        mov     ax, 0x4C00         ! exit(0)
        int     0x21

new15start:
                                ! New interrupt 15 handler
        pushf
        cmp     ax, 0x544D         ! Is it my call?
        je      my15
        popf
        cseg    jmpf     (old15)    ! No, continue with old 15
my15:
        popf
        push    bp
        mov     bp, sp
        andb    6(bp), ~0x01       ! clear carry, call will succeed
        xchgb   al, ah            ! ax = 4D54, also means call works
        cseg    mov     dx, (umb+0) ! dx = base of UMB
        cseg    mov     cx, (umb+2) ! cx = size of UMB
        pop     bp
        iret                     ! return to caller

new15end:

! u32_t mon2abs(void *ptr)
!   Address in monitor data to absolute address.
.define _mon2abs
_mon2abs:
        mov     bx, sp
        mov     ax, 2(bx)         ! ptr
        mov     dx, ds            ! Monitor data segment
        jmp     seg2abs

seg2abs:
                                ! Translate dx:ax to the 32 bit address dx-ax
        push    cx
        movb    ch, dh
        movb    cl, 4
        shl     dx, cl
        shr     ch, cl            ! ch-dx = dx << 4
        add     ax, dx
        adcb    ch, 0            ! ch-ax = ch-dx + ax
        movb    dl, ch
        xorb    dh, dh          ! dx-ax = ch-ax
        pop     cx
        ret

abs2seg:
                                ! Translate the 32 bit address dx-ax to dx:ax
        push    cx
        movb    ch, dl
        mov     dx, ax
        and     ax, 0x000F        ! Offset in ax
        movb    cl, 4
        shr     dx, cl
        shlb    ch, cl
        orb     dh, ch          ! dx = ch-dx >> 4
        pop     cx
        ret

```

```

! void raw_copy(u32_t dstaddr, u32_t srcaddr, u32_t count)
!     Copy count bytes from srcaddr to dstaddr. Don't do overlaps.
!     Also handles copying words to or from extended memory.
.define _raw_copy
_raw_copy:
    push    bp
    mov     bp, sp
    push    si
    push    di                ! Save C variable registers
copy:
    cmp     14(bp), 0
    jnz     bigcopy
    mov     cx, 12(bp)
    jcxz    copydone          ! Count is zero, end copy
    cmp     cx, 0xFFFF0
    jb      smallcopy
bigcopy: mov     cx, 0xFFFF0    ! Don't copy more than about 64K at once
smallcopy:
    push    cx                ! Save copying count
    mov     ax, 4(bp)
    mov     dx, 6(bp)
    cmp     dx, 0x0010        ! Copy to extended memory?
    jae     ext_copy
    cmp     10(bp), 0x0010    ! Copy from extended memory?
    jae     ext_copy
    call    abs2seg
    mov     di, ax
    mov     es, dx            ! es:di = dstaddr
    mov     ax, 8(bp)
    mov     dx, 10(bp)
    call    abs2seg
    mov     si, ax
    mov     ds, dx            ! ds:si = srcaddr
    shr     cx, 1             ! Words to move
rep     movs    cx, cx        ! Do the word copy
    adc     cx, cx            ! One more byte?
rep     movsb    cx, cx        ! Do the byte copy
    mov     ax, ss            ! Restore ds and es from the remaining ss
    mov     ds, ax
    mov     es, ax
    jmp     copyadjust
ext_copy:
    mov     (x_dst_desc+2), ax
    movb    (x_dst_desc+4), dl ! Set base of destination segment
    mov     ax, 8(bp)
    mov     dx, 10(bp)
    mov     (x_src_desc+2), ax
    movb    (x_src_desc+4), dl ! Set base of source segment
    mov     si, x_gdt         ! es:si = global descriptor table
    shr     cx, 1             ! Words to move
    movb    ah, 0x87          ! Code for extended memory move
    int     0x15
copyadjust:
    pop     cx                ! Restore count
    add     4(bp), cx
    adc     6(bp), 0          ! srcaddr += copycount
    add     8(bp), cx
    adc     10(bp), 0         ! dstaddr += copycount
    sub     12(bp), cx
    sbb     14(bp), 0         ! count -= copycount
    jmp     copy              ! and repeat
copydone:
    pop     di
    pop     si                ! Restore C variable registers
    pop     bp
    ret

! u16_t get_word(u32_t addr);
! void put_word(u32_t addr, u16_t word);
!     Read or write a 16 bits word at an arbitrary location.
.define _get_word, _put_word
_get_word:
    mov     bx, sp

```

```

        call    gp_getaddr
        mov     ax, (bx)          ! Word to get from addr
        jmp     gp_ret
_put_word:
        mov     bx, sp
        push    6(bx)            ! Word to store at addr
        call    gp_getaddr
        pop     (bx)             ! Store the word
        jmp     gp_ret
gp_getaddr:
        mov     ax, 2(bx)
        mov     dx, 4(bx)
        call    abs2seg
        mov     bx, ax
        mov     ds, dx           ! ds:bx = addr
        ret
gp_ret:
        push    es
        pop     ds               ! Restore ds
        ret

! void relocate(void);
!     After the program has copied itself to a safer place, it needs to change
!     the segment registers.  Caddr has already been set to the new location.
#define _relocate
_relocate:
        pop     bx               ! Return address
        mov     ax, (_caddr+0)
        mov     dx, (_caddr+2)
        call    abs2seg
        mov     cx, dx           ! cx = new code segment
        mov     ax, cs           ! Old code segment
        sub     ax, cx           ! ax = -(new - old) = -Moving offset
        mov     dx, ds
        sub     dx, ax
        mov     ds, dx           ! ds += (new - old)
        mov     es, dx
        mov     ss, dx
        xor     ax, ax
        call    seg2abs
        mov     (_daddr+0), ax
        mov     (_daddr+2), dx   ! New data address
        push    cx               ! New text segment
        push    bx               ! Return offset of this function
        retf                    ! Relocate

! void *brk(void *addr)
! void *sbrk(size_t incr)
!     Cannot fail implementations of brk(2) and sbrk(3), so we can use
!     malloc(3).  They reboot on stack collision instead of returning -1.
.sect .data
        .align 2
break:  .data2 _end             ! A fake heap pointer
.sect .text
#define __brk, __brk, __sbrk, __sbrk
__brk:
__brk:
        xor     ax, ax           ! __brk is for the standard C compiler
        jmp     sbrk            ! break= 0; return sbrk(addr);
__sbrk:
__sbrk:
        mov     ax, (break)     ! ax= current break
sbrk:   push    ax               ! save it as future return value
        mov     bx, sp
        add     ax, 4(bx)        ! Stack is now: (retval, retaddr, incr, ...)
        mov     (break), ax     ! Set new break
        lea     dx, -1024(bx)    ! sp minus a bit of breathing space
        cmp     dx, ax           ! Compare with the new break
        jb      heaperr         ! Suffocating noises
        pop     ax               ! Return old break (0 for brk)
        ret
heaperr:push    nomem
        call    _printf
        call    quit

```

```

.sect .rom
nomem: .ascii  "\nOut of memory\n\0"
.sect .text

! int dev_open(void);
!     Open file 'vdisk' to use as the Minix virtual disk.  Store handle in
!     vfd.  Returns 0 for success, otherwise the DOS error code.
.define _dev_open
_dev_open:
    call    _dev_close      ! If already open then first close
    mov     dx, (_vdisk)    ! ds:dx = Address of file name
    mov     ax, 0x3D22      ! Open file read-write & deny write
    int     0x21
    jnc     opok            ! Open succeeded?
    cmp     ax, 5           ! Open failed, "access denied"?
    jne     opbad
    mov     ax, 0x3D40      ! Open file read-only
    int     0x21
    jc      opbad
opok:     mov     (vfd), ax  ! File handle to open file
    xor     ax, ax         ! Zero for success
opbad:    ret

! int dev_close(void);
!     Close the dos virtual disk.
.define _dev_close
_dev_close:
    mov     bx, -1
    cmp     (vfd), bx      ! Already closed?
    je      1f
    movb    ah, 0x3E       ! Close file
    xchg    bx, (vfd)      ! bx = vfd; vfd = -1;
    int     0x21
    jc      0f
1:        xor     ax, ax
0:        ret

! int dev_boundary(u32_t sector);
!     Returns false; files have no visible boundaries.
.define _dev_boundary
_dev_boundary:
    xor     ax, ax
    ret

! int readsectors(u32_t bufaddr, u32_t sector, u8_t count)
! int writesectors(u32_t bufaddr, u32_t sector, u8_t count)
!     Read/write several sectors from/to the Minix virtual disk.  Count
!     must fit in a byte.  The external variable vfd is the file handle.
!     Returns 0 for success, otherwise the DOS error code.
!
.define _readsectors, _writesectors
_writesectors:
    push    bp
    mov     bp, sp
    movb    13(bp), 0x40    ! Code for a file write
    jmp     rwsec
_readsectors:
    push    bp
    mov     bp, sp
    movb    13(bp), 0x3F    ! Code for a file read
rwsec:
    cmp     (vfd), -1      ! Currently closed?
    jne     0f
    call    _dev_open      ! Open file if needed
    test    ax, ax
    jnz     rwerr
0:        mov     dx, 8(bp)
    mov     bx, 10(bp)     ! bx-dx = Sector number
    mov     cx, 9
mul512:   shl     dx, 1
    rcl     bx, 1          ! bx-dx *= 512
    loop    mul512
    mov     cx, bx         ! cx-dx = Byte position in file
    mov     bx, (vfd)      ! bx = File handle

```

```

    mov     ax, 0x4200      ! Lseek absolute
    int     0x21
    jb      rwerr
    mov     bx, (vfd)       ! bx = File handle
    mov     ax, 4(bp)
    mov     dx, 6(bp)       ! dx-ax = Address to transfer data to/from
    call    abs2seg
    mov     ds, dx
    mov     dx, ax          ! ds:dx = Address to transfer data to/from
    xorb    cl, cl
    movb    ch, 12(bp)      ! ch = Number of sectors to transfer
    shl     cx, 1           ! cx = Number of bytes to transfer
    push    cx              ! Save count
    movb    ah, 13(bp)      ! Read or write
    int     0x21
    pop     cx              ! Restore count
    push    es
    pop     ds              ! Restore ds
    jb      rwerr
    cmp     ax, cx          ! All bytes transferred?
    je      rwall
    mov     ax, 0x05        ! The DOS code for "I/O error", but different
    jmp     rwerr
rwall:    call    wheel      ! Display tricks
    xor     ax, ax
rwerr:    pop     bp
    ret

! int getch(void);
!     Read a character from the keyboard, and check for an expired timer.
!     A carriage return is changed into a linefeed for UNIX compatibility.
.define _getch
_getch:
    xor     ax, ax
    xchg    ax, (unchar)    ! Ungotten character?
    test    ax, ax
    jnz     gotch
getch:    hlt              ! Play dead until interrupted (see pause())
    movb    ah, 0x01        ! Keyboard status
    int     0x16
    jnz     press          ! Keypress?
    call    _expired        ! Timer expired?
    test    ax, ax
    jz      getch
    mov     ax, ESC        ! Return ESC
    ret
press:
    xorb    ah, ah          ! Read character from keyboard
    int     0x16
    cmpb    al, 0x0D        ! Carriage return?
    jnz     nocr
    movb    al, 0x0A        ! Change to linefeed
nocr:     cmpb    al, ESC    ! Escape typed?
    jne     noesc
    inc     (escape)        ! Set flag
noesc:    xorb    ah, ah    ! ax = al
gotch:    ret

! int ungetch(void);
!     Return a character to undo a getch().
.define _ungetch
_ungetch:
    mov     bx, sp
    mov     ax, 2(bx)
    mov     (unchar), ax
    ret

! int escape(void);
!     True if ESC has been typed.
.define _escape
_escape:
    movb    ah, 0x01        ! Keyboard status
    int     0x16
    jz      escflg          ! Keypress?

```

```

        cmpb    al, ESC          ! Escape typed?
        jne     escflg
        xorb    ah, ah           ! Discard the escape
        int     0x16
        inc     (escape)        ! Set flag
escflg: xor     ax, ax
        xchg    ax, (escape)    ! Escape typed flag
        ret

! int putch(int c);
!   Write a character in teletype mode.  The putk synonym is
!   for the kernel printf function that uses it.
!   Newlines are automatically preceded by a carriage return.
!
.define _putch, _putk
_putch:
_putk: mov     bx, sp
        movb    al, 2(bx)       ! al = character to be printed
        testb   al, al          ! Kernel printf adds a null char to flush queue
        jz      nulch
        cmpb    al, 0x0A        ! al = newline?
        jnz     putc
        movb    al, 0x20        ! Erase wheel and do a carriage return
        call    plotc           ! plotc(' ');
nodirt: movb    al, 0x0D
        call    putc           ! putc('\r')
        movb    al, 0x0A        ! Restore the '\n' and print it
putc:   movb    ah, 0x0E        ! Print character in teletype mode
        mov     bx, 0x0001      ! Page 0, foreground color
        int     0x10           ! Call BIOS VIDEO_IO
nulch:  ret

! |/-\|/-\|/-\|/-\|/-\ (playtime)
wheel:  mov     bx, (gp)
        movb    al, (bx)
        inc     bx              ! al = *gp++;
        cmp     bx, glyphs+4
        jne     0f
        mov     bx, glyphs
0:       mov     (gp), bx       ! gp= gp == glyphs + 4 ? glyphs : gp;
        ! jmp    plotc
plotc:  movb    ah, 0x0A        ! 0x0A = write character at cursor
        mov     bx, 0x0001      ! Page 0, foreground color
        mov     cx, 0x0001      ! Just one character
        int     0x10
        ret

.sect .data
        .align  2
gp:      .data2  glyphs
glyphs:  .ascii  "|/-\\"
.sect .text

! void pause(void);
!   Wait for an interrupt using the HLT instruction.  This either saves
!   power, or tells an x86 emulator that nothing is happening right now.
.define _pause
_pause:  hlt
        ret

! void set_mode(unsigned mode);
! void clear_screen(void);
!   Set video mode / clear the screen.
.define _set_mode, _clear_screen
_set_mode:
        mov     bx, sp
        mov     ax, 2(bx)       ! Video mode
        cmp     ax, (cur_vid_mode)
        je      modeok          ! Mode already as requested?
        mov     (cur_vid_mode), ax
_clear_screen:
        mov     ax, (cur_vid_mode)
        andb    ah, 0x7F        ! Test bits 8-14, clear bit 15 (8x8 flag)
        jnz     xvesa           ! VESA extended mode?

```

```

        int     0x10          ! Reset video (ah = 0)
        jmp     mdset
xvesa:  mov     bx, ax          ! bx = extended mode
        mov     ax, 0x4F02     ! Reset video
        int     0x10
mdset:  testb   (cur_vid_mode+1), 0x80
        jz      setcur        ! 8x8 font requested?
        mov     ax, 0x1112     ! Load ROM 8 by 8 double-dot patterns
        xorb    bl, bl        ! Load block 0
        int     0x10
setcur: xor     dx, dx          ! dl = column = 0, dh = row = 0
        xorb    bh, bh        ! Page 0
        movb    ah, 0x02      ! Set cursor position
        int     0x10
modeok: ret

restore_video:                ! To restore the video mode on exit
        movb    al, 0x20
        call    plotc         ! Erase wheel
        push    (old_vid_mode)
        call    _set_mode
        pop     ax
        ret

! u32_t get_tick(void);
! Return the current value of the clock tick counter. This counter
! increments 18.2 times per second. Poll it to do delays. Does not
! work on the original PC, but works on the PC/XT.
#define _get_tick
_get_tick:
        xorb    ah, ah        ! Code for get tick count
        int     0x1A
        mov     ax, dx
        mov     dx, cx        ! dx:ax = cx:dx = tick count
        ret

! Functions used to obtain info about the hardware. Boot uses this information
! itself, but will also pass them on to a pure 386 kernel, because one can't
! make BIOS calls from protected mode. The video type could probably be
! determined by the kernel too by looking at the hardware, but there is a small
! chance on errors that the monitor allows you to correct by setting variables.

#define _get_bus              ! returns type of system bus
#define _get_video            ! returns type of display

! u16_t get_bus(void)
! Return type of system bus, in order: XT, AT, MCA.
_get_bus:
        call    gettrueproc
        xor     dx, dx        ! Assume XT
        cmp     ax, 286        ! An AT has at least a 286
        jb      got_bus
        inc     dx            ! Assume AT
        movb    ah, 0xC0      ! Code for get configuration
        int     0x15
        jc      got_bus        ! Carry clear and ah = 00 if supported
        testb   ah, ah
        jne     got_bus
eseg    movb    al, 5(bx)      ! Load feature byte #1
        inc     dx            ! Assume MCA
        testb   al, 0x02      ! Test bit 1 - "bus is Micro Channel"
        jnz     got_bus
        dec     dx            ! Assume AT
        testb   al, 0x40      ! Test bit 6 - "2nd 8259 installed"
        jnz     got_bus
        dec     dx            ! It is an XT
got_bus:
        push    ds
        pop     es            ! Restore es
        mov     ax, dx        ! Return bus code
        mov     (bus), ax     ! Keep bus code, A20 handler likes to know
        ret

```



```

! u16_t get_video(void)
!     Return type of display, in order: MDA, CGA, mono EGA, color EGA,
!     mono VGA, color VGA.
_get_video:
    mov     ax, 0x1A00      ! Function 1A returns display code
    int     0x10           ! al = 1A if supported
    cmpb    al, 0x1A
    jnz     no_dc          ! No display code function supported

    mov     ax, 2
    cmpb    bl, 5          ! Is it a monochrome EGA?
    jz      got_video
    inc     ax
    cmpb    bl, 4          ! Is it a color EGA?
    jz      got_video
    inc     ax
    cmpb    bl, 7          ! Is it a monochrome VGA?
    jz      got_video
    inc     ax
    cmpb    bl, 8          ! Is it a color VGA?
    jz      got_video

no_dc:     movb    ah, 0x12      ! Get information about the EGA
    movb    bl, 0x10
    int     0x10
    cmpb    bl, 0x10        ! Did it come back as 0x10? (No EGA)
    jz      no_ega

    mov     ax, 2
    cmpb    bh, 1          ! Is it monochrome?
    jz      got_video
    inc     ax
    jmp     got_video

no_ega:    int     0x11        ! Get bit pattern for equipment
    and     ax, 0x30        ! Isolate color/mono field
    sub     ax, 0x30
    jz      got_video      ! Is it an MDA?
    mov     ax, 1          ! No it's CGA

got_video:
    ret

! Function to leave the boot monitor and run Minix.
.define _minix

! void minix(u32_t koff, u32_t kcs, u32_t kds,
!           char *bootparams, size_t paramsize, u32_t aout);
_minix:
    push    bp
    mov     bp, sp        ! Pointer to arguments

    mov     dx, 0x03F2     ! Floppy motor drive control bits
    movb    al, 0x0C      ! Bits 4-7 for floppy 0-3 are off
    outb    dx            ! Kill the motors
    push    ds
    xor     ax, ax        ! Vector & BIOS data segments
    mov     ds, ax
    andb    (0x043F), 0xF0 ! Clear diskette motor status bits of BIOS
    pop     ds
    cli                     ! No more interruptions

    test    (_k_flags), K_I386 ! Minix-386?
    jnz     minix386

! Call Minix in real mode.
minix86:
    push    22(bp)        ! Address of a.out headers
    push    20(bp)

    push    18(bp)        ! # bytes of boot parameters
    push    16(bp)        ! Address of boot parameters

```

```

0:    mov     dx, cs             ! Monitor far return address
    mov     ax, ret86
    cmp     (_mem+14), 0       ! Any extended memory? (mem[1].size > 0 ?)
    jnz     0f
    xor     dx, dx             ! If no ext mem then monitor not preserved
    xor     ax, ax
    push    dx                 ! Push monitor far return address or zero
    push    ax

    mov     ax, 8(bp)
    mov     dx, 10(bp)
    call    abs2seg
    push    dx                 ! Kernel code segment
    push    4(bp)              ! Kernel code offset
    mov     ax, 12(bp)
    mov     dx, 14(bp)
    call    abs2seg
    mov     ds, dx             ! Kernel data segment
    mov     es, dx             ! Set es to kernel data too
    retf                       ! Make a far call to the kernel

! Call 386 Minix in 386 mode.
minix386:
cseg  mov     (cs_real-2), cs ! Patch CS and DS into the instructions that
cseg  mov     (ds_real-2), ds ! reload them when switching back to real mode
    mov     eax, cr0
    orb     al, 0x01          ! Set PE (protection enable) bit
o32   mov     (msw), eax       ! Save as protected mode machine status word

    mov     dx, ds             ! Monitor ds
    mov     ax, p_gdt          ! dx:ax = Global descriptor table
    call    seg2abs
    mov     (p_gdt_desc+2), ax
    movb    (p_gdt_desc+4), dl ! Set base of global descriptor table

    mov     ax, 12(bp)
    mov     dx, 14(bp)         ! Kernel ds (absolute address)
    mov     (p_ds_desc+2), ax
    movb    (p_ds_desc+4), dl ! Set base of kernel data segment

    mov     dx, ss             ! Monitor ss
    xor     ax, ax             ! dx:ax = Monitor stack segment
    call    seg2abs            ! Minix starts with the stack of the monitor
    mov     (p_ss_desc+2), ax
    movb    (p_ss_desc+4), dl

    mov     ax, 8(bp)
    mov     dx, 10(bp)         ! Kernel cs (absolute address)
    mov     (p_cs_desc+2), ax
    movb    (p_cs_desc+4), dl

    mov     dx, cs             ! Monitor cs
    xor     ax, ax             ! dx:ax = Monitor code segment
    call    seg2abs
    mov     (p_mcs_desc+2), ax
    movb    (p_mcs_desc+4), dl

    push    MCS_SELECTOR
    push    int86              ! Far address to INT86 support
o32   push    20(bp)           ! Address of a.out headers

    push    0
    push    18(bp)             ! 32 bit size of parameters on stack
    push    0
    push    16(bp)             ! 32 bit address of parameters (ss relative)

    push    MCS_SELECTOR
    push    ret386             ! Monitor far return address

    push    0
    push    CS_SELECTOR
    push    6(bp)
    push    4(bp)              ! 32 bit far address to kernel entry point

```

```

        call    real2prot      ! Switch to protected mode
        mov     ax, DS_SELECTOR
        mov     ds, ax        ! Kernel data
        mov     ax, ES_SELECTOR
        mov     es, ax        ! Flat 4 Gb
o32     retf                  ! Make a far call to the kernel

! Minix-86 returns here on a halt or reboot.
ret86:
        mov     8(bp), ax
        mov     10(bp), dx    ! Return value
        jmp     return

! Minix-386 returns here on a halt or reboot.
ret386:
o32     mov     8(bp), eax     ! Return value
        call    prot2real     ! Switch to real mode

return:
        mov     sp, bp        ! Pop parameters
        sti                     ! Can take interrupts again

        call    _get_video    ! MDA, CGA, EGA, ...
        movb    dh, 24        ! dh = row 24
        cmp     ax, 2         ! At least EGA?
        jb      is25          ! Otherwise 25 rows
        push    ds
        xor     ax, ax        ! Vector & BIOS data segments
        mov     ds, ax
        movb    dh, (0x0484)  ! Number of rows on display minus one
is25:
        pop     ds

        xorb     dl, dl        ! dl = column 0
        xorb     bh, bh        ! Page 0
        movb     ah, 0x02      ! Set cursor position
        int      0x10

        xorb     ah, ah        ! Whack the disk system, Minix may have messed
        movb     dl, 0x80      ! it up
        int      0x13

        call     gettrueproc
        cmp     ax, 286
        jb      noclock
        xorb     al, al
tryclk: decb     al
        jz      noclock
        movb     ah, 0x02      ! Get real-time clock time (from CMOS clock)
        int      0x1A
        jc      tryclk        ! Carry set, not running or being updated
        movb     al, ch        ! ch = hour in BCD
        call     bcd           ! al = (al >> 4) * 10 + (al & 0x0F)
        mulb     (c60)         ! 60 minutes in an hour
        mov      bx, ax        ! bx = hour * 60
        movb     al, cl        ! cl = minutes in BCD
        call     bcd
        add      bx, ax        ! bx = hour * 60 + minutes
        movb     al, dh        ! dh = seconds in BCD
        call     bcd
        xchg     ax, bx        ! ax = hour * 60 + minutes, bx = seconds
        mul      (c60)         ! dx-ax = (hour * 60 + minutes) * 60
        add      bx, ax
        adc     dx, 0          ! dx-bx = seconds since midnight
        mov      ax, dx
        mul      (c19663)
        xchg     ax, bx
        mul      (c19663)
        add      dx, bx        ! dx-ax = dx-bx * (0x1800B0 / (2*2*2*2*5))
        mov      cx, ax        ! (0x1800B0 = ticks per day of BIOS clock)
        mov      ax, dx
        xor      dx, dx
        div      (c1080)
        xchg     ax, cx

```

```

        div     (c1080)          ! cx-ax = dx-ax / (24*60*60 / (2*2*2*2*5))
        mov     dx, ax           ! cx-dx = ticks since midnight
        movb    ah, 0x01        ! Set system time
        int     0x1A
noclock:

        mov     ax, 8(bp)
        mov     dx, 10(bp)      ! dx-ax = return value from the kernel
        pop     bp
        ret                     ! Return to monitor as if nothing much happened

! Transform BCD number in al to a regular value in ax.
bcd:     movb    ah, al
        shr     ah, 4
        andb    al, 0x0F
        aad                     ! ax = (al >> 4) * 10 + (al & 0x0F)
        ret

! Support function for Minix-386 to make an 8086 interrupt call.
int86:
        mov     bp, sp
        call    prot2real

o32 xor     eax, eax
        mov     es, ax          ! Vector & BIOS data segments
o32 eseg mov    (0x046C), eax    ! Clear BIOS clock tick counter

        sti                     ! Enable interrupts

        movb    al, 0xCD        ! INT instruction
        movb    ah, 8(bp)       ! Interrupt number?
        testb   ah, ah
        jnz     0f              ! Nonzero if INT, otherwise far call
        push    cs
        push    intret+2        ! Far return address
o32 push    12(bp)              ! Far driver address
        mov     ax, 0x90CB      ! RETF; NOP
0: cseg mov     (intret), ax     ! Patch 'INT n' or 'RETF; NOP' into code

        mov     ds, 16(bp)      ! Load parameters
        mov     es, 18(bp)
o32 mov     eax, 20(bp)
o32 mov     ebx, 24(bp)
o32 mov     ecx, 28(bp)
o32 mov     edx, 32(bp)
o32 mov     esi, 36(bp)
o32 mov     edi, 40(bp)
o32 mov     ebp, 44(bp)

intret: int     0xFF            ! Do the interrupt or far call

o32 push    ebp                ! Save results
o32 pushf
        mov     bp, sp
o32 pop     8+8(bp)            ! eflags
        mov     8+16(bp), ds
        mov     8+18(bp), es
o32 mov     8+20(bp), eax
o32 mov     8+24(bp), ebx
o32 mov     8+28(bp), ecx
o32 mov     8+32(bp), edx
o32 mov     8+36(bp), esi
o32 mov     8+40(bp), edi
o32 pop     8+44(bp)          ! ebp

        cli                     ! Disable interrupts

        xor     ax, ax
        mov     ds, ax          ! Vector & BIOS data segments
o32 mov     cx, (0x046C)        ! Collect lost clock ticks in ecx

        mov     ax, ss
        mov     ds, ax          ! Restore monitor ds
        call    real2prot

```

```

        mov     ax, DS_SELECTOR ! Kernel data
        mov     ds, ax
o32     retf           ! Return to the kernel

! Switch from real to protected mode.
real2prot:
        movb    ah, 0x02        ! Code for A20 enable
        call    gate_A20

        lgdt     (p_gdt_desc)    ! Global descriptor table
o32     mov     eax, (pdir)       ! Load page directory base register
        mov     cr3, eax
        mov     eax, cr0
o32     xchg    eax, (msw)        ! Exchange real mode msw for protected mode msw
        mov     cr0, eax
        jmpf    MCS_SELECTOR:cs_prot ! Set code segment selector
cs_prot:
        mov     ax, SS_SELECTOR ! Set data selectors
        mov     ds, ax
        mov     es, ax
        mov     ss, ax
        ret

! Switch from protected to real mode.
prot2real:
        lidt     (p_idt_desc)    ! Real mode interrupt vectors
        mov     eax, cr3
o32     mov     (pdir), eax       ! Save page directory base register
        mov     eax, cr0
o32     xchg    eax, (msw)        ! Exchange protected mode msw for real mode msw
        mov     cr0, eax
        jmpf    0xDEAD:cs_real    ! Reload cs register
cs_real:
        mov     ax, 0xBEEF
ds_real:
        mov     ds, ax           ! Reload data segment registers
        mov     es, ax
        mov     ss, ax

        xorb    ah, ah          ! Code for A20 disable
        ! jmp    gate_A20

! Enable (ah = 0x02) or disable (ah = 0x00) the A20 address line.
gate_A20:
        cmp     (bus), 2        ! PS/2 bus?
        je      gate_PS_A20
        call    kb_wait
        movb    al, 0xD1        ! Tell keyboard that a command is coming
        outb    0x64
        call    kb_wait
        movb    al, 0xDD        ! 0xDD = A20 disable code if ah = 0x00
        orb     al, ah          ! 0xDF = A20 enable code if ah = 0x02
        outb    0x60
        call    kb_wait
        movb    al, 0xFF        ! Pulse output port
        outb    0x64
        call    kb_wait        ! Wait for the A20 line to settle down
        ret
kb_wait:
        inb     0x64
        testb   al, 0x02        ! Keyboard input buffer full?
        jnz     kb_wait        ! If so, wait
        ret

gate_PS_A20:                ! The PS/2 can twiddle A20 using port A
        inb     0x92            ! Read port A
        andb    al, 0xFD
        orb     al, ah          ! Set A20 bit to the required state
        outb    0x92            ! Write port A
        jmp     .+2             ! Small delay
A20ok:  inb     0x92            ! Check port A
        andb    al, 0x02
        cmpb    al, ah          ! A20 line settled down to the new state?
        jne     A20ok          ! If not then wait

```

```

        ret

! void int15(bios_env_t *ep)
!   Do an "INT 15" call, primarily for APM (Power Management).
.define _int15
_int15:
    push    si                ! Save callee-save register si
    mov     si, sp
    mov     si, 4(si)         ! ep
    mov     ax, (si)          ! ep→ax
    mov     bx, 2(si)         ! ep→bx
    mov     cx, 4(si)         ! ep→cx
    int     0x15              ! INT 0x15 BIOS call
    pushf                    ! Save flags
    mov     (si), ax          ! ep→ax
    mov     2(si), bx         ! ep→bx
    mov     4(si), cx         ! ep→cx
    pop     6(si)             ! ep→flags
    pop     si                ! Restore
    ret

.sect    .rom
        .align    4
c60:     .data2    60          ! Constants for MUL and DIV
c1024:   .data2    1024
c1080:   .data2    1080
c19663:  .data2    19663

.sect    .data
        .align    4

! Global descriptor tables.
        UNSET     = 0          ! Must be computed

! For "Extended Memory Block Move".
x_gdt:
x_null_desc:
    ! Null descriptor
    .data2    0x0000, 0x0000
    .data1    0x00, 0x00, 0x00, 0x00
x_gdt_desc:
    ! Descriptor for this descriptor table
    .data2    6*8-1, UNSET
    .data1    UNSET, 0x00, 0x00, 0x00
x_src_desc:
    ! Source segment descriptor
    .data2    0xFFFF, UNSET
    .data1    UNSET, 0x92, 0x00, 0x00
x_dst_desc:
    ! Destination segment descriptor
    .data2    0xFFFF, UNSET
    .data1    UNSET, 0x92, 0x00, 0x00
x_bios_desc:
    ! BIOS segment descriptor (scratch for int 0x15)
    .data2    UNSET, UNSET
    .data1    UNSET, UNSET, UNSET, UNSET
x_ss_desc:
    ! BIOS stack segment descriptor (scratch for int 0x15)
    .data2    UNSET, UNSET
    .data1    UNSET, UNSET, UNSET, UNSET

! Protected mode descriptor table.
p_gdt:
p_null_desc:
    ! Null descriptor
    .data2    0x0000, 0x0000
    .data1    0x00, 0x00, 0x00, 0x00
p_gdt_desc:
    ! Descriptor for this descriptor table
    .data2    8*8-1, UNSET
    .data1    UNSET, 0x00, 0x00, 0x00
p_idt_desc:
    ! Real mode interrupt descriptor table descriptor
    .data2    0x03FF, 0x0000

```

```
.data1 0x00, 0x00, 0x00, 0x00
p_ds_desc:
! Kernel data segment descriptor (4Gb flat)
.data2 0xFFFF, UNSET
.data1 UNSET, 0x92, 0xCF, 0x00
p_es_desc:
! Physical memory descriptor (4Gb flat)
.data2 0xFFFF, 0x0000
.data1 0x00, 0x92, 0xCF, 0x00
p_ss_desc:
! Monitor data segment descriptor (64Kb flat)
.data2 0xFFFF, UNSET
.data1 UNSET, 0x92, 0x00, 0x00
p_cs_desc:
! Kernel code segment descriptor (4Gb flat)
.data2 0xFFFF, UNSET
.data1 UNSET, 0x9A, 0xCF, 0x00
p_mcs_desc:
! Monitor code segment descriptor (64 kb flat) (unused)
.data2 0xFFFF, UNSET
.data1 UNSET, 0x9A, 0x00, 0x00

xms_handle: .data2 -1 ! Handle of allocated XMS block
vfd: .data2 -1 ! Virtual disk file handle

.sect .bss
.comm xms_driver, 4 ! Vector to XMS driver
.comm old_vid_mode, 2 ! Video mode at startup
.comm cur_vid_mode, 2 ! Current video mode
.comm msw, 4 ! Saved machine status word (cr0)
.comm pdbr, 4 ! Saved page directory base register (cr3)
.comm escape, 2 ! Escape typed?
.comm bus, 2 ! Saved return value of _get_bus
.comm unchar, 2 ! Char returned by ungetch(c)

!
! $PchId: doshead.ack.s,v 1.7 2002/02/27 19:37:52 philip Exp $
```

```
/*      image.h - Info between installboot and boot.      Author: Kees J. Bot
*/

#define IM_NAME_MAX      63

struct image_header {
    char      name[IM_NAME_MAX + 1];  /* Null terminated. */
    struct exec      process;
};

/*
 * $PchId: image.h,v 1.4 1995/11/27 22:23:12 philip Exp $
 */
```



```

/*      installboot 3.0 - Make a device bootable      Author: Kees J. Bot
*                                                    21 Dec 1991
*
* Either make a device bootable or make an image from kernel, mm, fs, etc.
*/
#define nil 0
#define _POSIX_SOURCE 1
#define _MINIX 1
#include <stdio.h>
#include <stddef.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <dirent.h>
#include <a.out.h>
#include <minix/config.h>
#include <minix/const.h>
#include <minix/partition.h>
#include <minix/u64.h>
#include "rawfs.h"
#include "image.h"

#define BOOTBLOCK 0      /* Of course */
#define SECTOR_SIZE 512  /* Disk sector size. */
#define RATIO(b) ((b)/SECTOR_SIZE)
#define SIGNATURE 0xAA55 /* Boot block signature. */
#define BOOT_MAX 64      /* Absolute maximum size of secondary boot */
#define SIGPOS 510        /* Where to put signature word. */
#define PARTPOS 446       /* Offset to the partition table in a master
                          * boot block.
                          */

#define between(a, c, z) ((unsigned) ((c) - (a)) <= ((z) - (a)))
#define control(c) between('\0', (c), '\37')

#define BOOT_BLOCK_SIZE 1024

void report(char *label)
/* installboot: label: No such file or directory */
{
    fprintf(stderr, "installboot: %s: %s\n", label, strerror(errno));
}

void fatal(char *label)
{
    report(label);
    exit(1);
}

char *basename(char *name)
/* Return the last component of name, stripping trailing slashes from name.
 * Precondition: name != "/". If name is prefixed by a label, then the
 * label is copied to the basename too.
 */
{
    static char base[IM_NAME_MAX];
    char *p, *bp = base;

    if ((p = strchr(name, '.')) != nil) {
        while (name <= p && bp < base + IM_NAME_MAX - 1)
            *bp++ = *name++;
    }
    for (;;) {
        if ((p = strchr(name, '/')) == nil) { p = name; break; }
        if (*++p != 0) break;
        *--p = 0;
    }
    while (*p != 0 && bp < base + IM_NAME_MAX - 1) *bp++ = *p++;
    *bp = 0;
}

```

```
    return base;
}

void bread(FILE *f, char *name, void *buf, size_t len)
/* Read len bytes. Don't dare return without them. */
{
    if (len > 0 && fread(buf, len, 1, f) != 1) {
        if (ferror(f)) fatal(name);
        fprintf(stderr, "installboot: Unexpected EOF on %s\n", name);
        exit(1);
    }
}

void bwrite(FILE *f, char *name, void *buf, size_t len)
{
    if (len > 0 && fwrite(buf, len, 1, f) != 1) fatal(name);
}

long total_text= 0, total_data= 0, total_bss= 0;
int making_image= 0;

void read_header(int talk, char *proc, FILE *procf, struct image_header *ihdr)
/* Read the a.out header of a program and check it. If procf happens to be
 * nil then the header is already in *image_hdr and need only be checked.
 */
{
    int n, big= 0;
    static int banner= 0;
    struct exec *phdr= &ihdr->process;

    if (procf == nil) {
        /* Header already present. */
        n= phdr->a_hdrlen;
    } else {
        memset(ihdr, 0, sizeof(*ihdr));

        /* Put the basename of proc in the header. */
        strncpy(ihdr->name, basename(proc), IM_NAME_MAX);

        /* Read the header. */
        n= fread(phdr, sizeof(char), A_MINHDR, procf);
        if (ferror(procf)) fatal(proc);
    }

    if (n < A_MINHDR || BADMAG(*phdr)) {
        fprintf(stderr, "installboot: %s is not an executable\n", proc);
        exit(1);
    }

    /* Get the rest of the exec header. */
    if (procf != nil) {
        bread(procf, proc, ((char *) phdr) + A_MINHDR,
              phdr->a_hdrlen - A_MINHDR);
    }

    if (talk && !banner) {
        printf("  text  data  bss  size\n");
        banner= 1;
    }

    if (talk) {
        printf(" %8ld %8ld %8ld %9ld %s\n",
              phdr->a_text, phdr->a_data, phdr->a_bss,
              phdr->a_text + phdr->a_data + phdr->a_bss, proc);
    }
    total_text+= phdr->a_text;
    total_data+= phdr->a_data;
    total_bss+= phdr->a_bss;

    if (phdr->a_cpu == A_I8086) {
        long data= phdr->a_data + phdr->a_bss;

        if (!(phdr->a_flags & A_SEP)) data+= phdr->a_text;
    }
}
```

```

        if (phdr->a_text >= 65536) big|= 1;
        if (data >= 65536) big|= 2;
    }
    if (big) {
        fprintf(stderr,
            "%s will crash, %s%s%s segment%s larger then 64K\n",
            proc,
            big & 1 ? "text" : "",
            big == 3 ? " and " : "",
            big & 2 ? "data" : "",
            big == 3 ? "s are" : " is");
    }
}

void padimage(char *image, FILE *imagef, int n)
/* Add n zeros to image to pad it to a sector boundary. */
{
    while (n > 0) {
        if (putc(0, imagef) == EOF) fatal(image);
        n--;
    }
}

#define align(n)          (((n) + ((SECTOR_SIZE) - 1)) & ~((SECTOR_SIZE) - 1))

void copyexec(char *proc, FILE *procf, char *image, FILE *imagef, long n)
/* Copy n bytes from proc to image padded to fill a sector. */
{
    int pad, c;

    /* Compute number of padding bytes. */
    pad= align(n) - n;

    while (n > 0) {
        if ((c= getc(procf)) == EOF) {
            if (ferror(procf)) fatal(proc);
            fprintf(stderr, "installboot: premature EOF on %s\n", proc);
            exit(1);
        }
        if (putc(c, imagef) == EOF) fatal(image);
        n--;
    }
    padimage(image, imagef, pad);
}

void make_image(char *image, char **procv)
/* Collect a set of files in an image, each "segment" is nicely padded out
 * to SECTOR_SIZE, so it may be read from disk into memory without trickery.
 */
{
    FILE *imagef, *procf;
    char *proc, *file;
    int procn;
    struct image_header ihdr;
    struct exec phdr;
    struct stat st;

    making_image= 1;

    if ((imagef= fopen(image, "w")) == nil) fatal(image);

    for (procn= 0; (proc= *procv++) != nil; procn++) {
        /* Remove the label from the file name. */
        if ((file= strchr(proc, ':')) != nil) file++; else file= proc;

        /* Real files please, may need to seek. */
        if (stat(file, &st) < 0
            || (errno= EISDIR, !S_ISREG(st.st_mode))
            || (procf= fopen(file, "r")) == nil)
            fatal(proc);

        /* Read a.out header. */
        read_header(1, proc, procf, &ihdr);
    }
}

```

```

    /* Scratch. */
    phdr= ihdr.process;

    /* The symbol table is always stripped off. */
    ihdr.process.a_syms= 0;
    ihdr.process.a_flags &= ~A_NSYM;

    /* Write header padded to fill a sector */
    bwrite(imagef, image, &ihdr, sizeof(ihdr));

    padimage(image, imagef, SECTOR_SIZE - sizeof(ihdr));

    /* A page aligned executable needs the header in text. */
    if (phdr.a_flags & A_PAL) {
        rewind(procfd);
        phdr.a_text+= phdr.a_hdrlen;
    }

    /* Copy text and data of proc to image. */
    if (phdr.a_flags & A_SEP) {
        /* Separate I&D: pad text & data separately. */

        copyexec(proc, procfd, image, imagef, phdr.a_text);
        copyexec(proc, procfd, image, imagef, phdr.a_data);
    } else {
        /* Common I&D: keep text and data together. */

        copyexec(proc, procfd, image, imagef,
                  phdr.a_text + phdr.a_data);
    }

    /* Done with proc. */
    (void) fclose(procfd);
}
/* Done with image. */

if (fclose(imagef) == EOF) fatal(image);

printf(" ----- \n");
printf(" %8ld %8ld %8ld %9ld total\n",
        total_text, total_data, total_bss,
        total_text + total_data + total_bss);
}

void extractexec(FILE *imagef, char *image, FILE *procfd, char *proc,
                 long count, off_t *alen)
/* Copy a segment of an executable. It is padded to a sector in image. */
{
    char buf[SECTOR_SIZE];

    while (count > 0) {
        bread(imagef, image, buf, sizeof(buf));
        *alen+= sizeof(buf);

        bwrite(procfd, proc, buf,
               count < sizeof(buf) ? (size_t) count : sizeof(buf));
        count-= sizeof(buf);
    }
}

void extract_image(char *image)
/* Extract the executables from an image. */
{
    FILE *imagef, *procfd;
    off_t len;
    struct stat st;
    struct image_header ihdr;
    struct exec phdr;
    char buf[SECTOR_SIZE];

    if (stat(image, &st) < 0) fatal(image);

    /* Size of the image. */

```

```

len= S_ISREG(st.st_mode) ? st.st_size : -1;

if ((imagef= fopen(image, "r")) == nil) fatal(image);

while (len != 0) {
    /* Extract a program, first sector is an extended header. */
    bread(imagef, image, buf, sizeof(buf));
    len-= sizeof(buf);

    memcpy(&ihdr, buf, sizeof(ihdr));
    phdr= ihdr.process;

    /* Check header. */
    read_header(1, ihdr.name, nil, &ihdr);

    if ((procf= fopen(ihdr.name, "w")) == nil) fatal(ihdr.name);

    if (phdr.a_flags & A_PAL) {
        /* A page aligned process contains a header in text. */
        phdr.a_text+= phdr.a_hdrlen;
    } else {
        bwrite(procf, ihdr.name, &ihdr.process, phdr.a_hdrlen);
    }

    /* Extract text and data segments. */
    if (phdr.a_flags & A_SEP) {
        extractexec(imagef, image, procf, ihdr.name,
                    phdr.a_text, &len);
        extractexec(imagef, image, procf, ihdr.name,
                    phdr.a_data, &len);
    } else {
        extractexec(imagef, image, procf, ihdr.name,
                    phdr.a_text + phdr.a_data, &len);
    }

    if (fclose(procf) == EOF) fatal(ihdr.name);
}

int rawfd;      /* File descriptor to open device. */
char *rawdev;   /* Name of device. */

void readblock(off_t blk, char *buf, int block_size)
/* For rawfs, so that it can read blocks. */
{
    int n;

    if (lseek(rawfd, blk * block_size, SEEK_SET) < 0
        || (n= read(rawfd, buf, block_size)) < 0
    ) fatal(rawdev);

    if (n < block_size) {
        fprintf(stderr, "installboot: Unexpected EOF on %s\n", rawdev);
        exit(1);
    }
}

void writeblock(off_t blk, char *buf, int block_size)
/* Add a function to write blocks for local use. */
{
    if (lseek(rawfd, blk * block_size, SEEK_SET) < 0
        || write(rawfd, buf, block_size) < 0
    ) fatal(rawdev);
}

int raw_install(char *file, off_t *start, off_t *len, int block_size)
/* Copy bootcode or an image to the boot device at the given absolute disk
 * block number. This "raw" installation is used to place bootcode and
 * image on a disk without a filesystem to make a simple boot disk. Useful
 * in automated scripts for J. Random User.
 * Note: *len == 0 when an image is read. It is set right afterwards.
 */
{
    static char buf[_MAX_BLOCK_SIZE]; /* Nonvolatile block buffer. */

```

```

FILE *f;
off_t sec;
unsigned long devsize;
static int banner= 0;
struct partition entry;

/* See if the device has a maximum size. */
devsize= -1;
if (ioctl(rawfd, DIOCGETP, &entry) == 0) devsize= cv64ul(entry.size);

if ((f= fopen(file, "r")) == nil) fatal(file);

/* Copy sectors from file onto the boot device. */
sec= *start;
do {
    int off= sec % RATIO(BOOT_BLOCK_SIZE);

    if (fread(buf + off * SECTOR_SIZE, 1, SECTOR_SIZE, f) == 0)
        break;

    if (sec >= devsize) {
        fprintf(stderr,
            "installboot: %s can't be attached to %s\n",
                file, rawdev);
        return 0;
    }

    if (off == RATIO(BOOT_BLOCK_SIZE) - 1) writeblock(sec / RATIO(BOOT_BLOCK_
SIZE), buf, BOOT_BLOCK_SIZE);
    } while (++sec != *start + *len);

    if (ferror(f)) fatal(file);
    (void) fclose(f);

    /* Write a partial block, this may be the last image. */
    if (sec % RATIO(BOOT_BLOCK_SIZE) != 0) writeblock(sec / RATIO(BOOT_BLOCK_SIZE), b
uf, BOOT_BLOCK_SIZE);

    if (!banner) {
        printf(" sector length\n");
        banner= 1;
    }
    *len= sec - *start;
    printf("%08ld%08ld %s\n", *start, *len, file);
    *start= sec;
    return 1;
}

enum howto { FS, BOOT };

void make_bootable(enum howto how, char *device, char *bootblock,
                    char *bootcode, char **imagev)
/* Install bootblock on the bootsector of device with the disk addresses to
 * bootcode patched into the data segment of bootblock. "How" tells if there
 * should or shoudn't be a file system on the disk. The images in the imagev
 * vector are added to the end of the device.
 */
{
    char buf[_MAX_BLOCK_SIZE + 256], *adrp, *parmp;
    struct fileaddr {
        off_t    address;
        int      count;
    } bootaddr[BOOT_MAX + 1], *bap= bootaddr;
    struct exec boothdr;
    struct image_header dummy;
    struct stat st;
    ino_t ino;
    off_t sector, max_sector;
    FILE *bootf;
    off_t addr, fssize, pos, len;
    char *labels, *label, *image;
    int nolabel;
    int block_size = 0;

```

```
/* Open device and set variables for readblock. */
if ((rawfd= open(rawdev= device, O_RDWR)) < 0) fatal(device);

/* Read and check the superblock. */
fssize= r_super(&block_size);

switch (how) {
case FS:
    if (fssize == 0) {
        fprintf(stderr,
            "installboot: %s is not a Minix file system\n",
            device);
        exit(1);
    }
    break;
case BOOT:
    if (fssize != 0) {
        int s;
        printf("%s contains a file system!\n", device);
        printf("Scribbling in 10 seconds");
        for (s= 0; s < 10; s++) {
            fputc('.', stdout);
            fflush(stdout);
            sleep(1);
        }
        fputc('\n', stdout);
    }
    fssize= 1;      /* Just a boot block. */
}

if (how == FS) {
    /* See if the boot code can be found on the file system. */
    if ((ino= r_lookup(ROOT_INO, bootcode)) == 0) {
        if (errno != ENOENT) fatal(bootcode);
    }
} else {
    /* Boot code must be attached at the end. */
    ino= 0;
}

if (ino == 0) {
    /* For a raw installation, we need to copy the boot code onto
     * the device, so we need to look at the file to be copied.
     */
    if (stat(bootcode, &st) < 0) fatal(bootcode);

    if ((bootf= fopen(bootcode, "r")) == nil) fatal(bootcode);
} else {
    /* Boot code is present in the file system. */
    r_stat(ino, &st);

    /* Get the header from the first block. */
    if ((addr= r_vir2abs((off_t) 0)) == 0) {
        boothdr.a_magic[0]= !A_MAGIC0;
    } else {
        readblock(addr, buf, block_size);
        memcpy(&boothdr, buf, sizeof(struct exec));
    }
    bootf= nil;
    dummy.process= boothdr;
}

/* See if it is an executable (read_header does the check). */
read_header(0, bootcode, bootf, &dummy);
boothdr= dummy.process;

if (bootf != nil) fclose(bootf);

/* Get all the sector addresses of the secondary boot code. */
max_sector= (boothdr.a_hdrlen + boothdr.a_text
    + boothdr.a_data + SECTOR_SIZE - 1) / SECTOR_SIZE;

if (max_sector > BOOT_MAX * RATIO(block_size)) {
    fprintf(stderr, "installboot: %s is way too big\n", bootcode);
    exit(0);
}
```

```

}

/* Determine the addresses to the boot code to be patched into the
 * boot block.
 */
bap->count= 0; /* Trick to get the address recording going. */

for (sector= 0; sector < max_sector; sector++) {
    if (ino == 0) {
        addr= fssize + (sector / RATIO(block_size));
    } else
    if ((addr= r_vir2abs(sector / RATIO(block_size))) == 0) {
        fprintf(stderr, "installboot: %s has holes!\n",
                                bootcode);
        exit(1);
    }
    addr= (addr * RATIO(block_size)) + (sector % RATIO(block_size));

    /* First address of the addresses array? */
    if (bap->count == 0) bap->address= addr;

    /* Paste sectors together in a multisector read. */
    if (bap->address + bap->count == addr)
        bap->count++;
    else {
        /* New address. */
        bap++;
        bap->address= addr;
        bap->count= 1;
    }
}
(++bap)->count= 0; /* No more. */

/* Get the boot block and patch the pieces in. */
readblock(BOOTBLOCK, buf, BOOT_BLOCK_SIZE);

if ((bootf= fopen(bootblock, "r")) == nil) fatal(bootblock);

read_header(0, bootblock, bootf, &dummy);
boothdr= dummy.process;

if (boothdr.a_text + boothdr.a_data +
    4 * (bap - bootaddr) + 1 > PARTPOS) {
    fprintf(stderr,
        "installboot: %s + addresses to %s don't fit in the boot sector\n",
        bootblock, bootcode);
    fprintf(stderr,
        "You can try copying/reinstalling %s to defragment it\n",
        bootcode);
    exit(1);
}

/* All checks out right. Read bootblock into the boot block! */
bread(bootf, bootblock, buf, boothdr.a_text + boothdr.a_data);
(void) fclose(bootf);

/* Patch the addresses in. */
adrp= buf + (int) (boothdr.a_text + boothdr.a_data);
for (bap= bootaddr; bap->count != 0; bap++) {
    *adrp++= bap->count;
    *adrp++= (bap->address >> 0) & 0xFF;
    *adrp++= (bap->address >> 8) & 0xFF;
    *adrp++= (bap->address >> 16) & 0xFF;
}
/* Zero count stops bootblock's reading loop. */
*adrp++= 0;

if (bap > bootaddr+1) {
    printf("%s and %d addresses to %s patched into %s\n",
        bootblock, (int)(bap - bootaddr), bootcode, device);
}

/* Boot block signature. */
buf[SIGPOS+0]= (SIGNATURE >> 0) & 0xFF;

```



```

buf[SIGPOS+1]= (SIGNATURE >> 8) & 0xFF;

/* Sector 2 of the boot block is used for boot parameters, initially
 * filled with null commands (newlines). Initialize it only if
 * necessary.
 */
for (parmp= buf + SECTOR_SIZE; parmp < buf + 2*SECTOR_SIZE; parmp++) {
    if (*imagev != nil || (control(*parmp) && *parmp != '\n')) {
        /* Param sector must be initialized. */
        memset(buf + SECTOR_SIZE, '\n', SECTOR_SIZE);
        break;
    }
}

/* Offset to the end of the file system to add boot code and images. */
pos= fssize * RATIO(block_size);

if (ino == 0) {
    /* Place the boot code onto the boot device. */
    len= max_sector;
    if (!raw_install(bootcode, &pos, &len, block_size)) {
        if (how == FS) {
            fprintf(stderr,
                "\t(Isn't there a copy of %s on %s that can be used?)\n",
                    bootcode, device);
        }
        exit(1);
    }
}

parmp= buf + SECTOR_SIZE;
nolabel= 0;

if (how == BOOT) {
    /* A boot only disk needs to have floppies swapped. */
    strcpy(parmp,
        "trailer)echo \\nInsert the root diskette then hit RETURN\\n\\w\\c\\n" );
    parmp+= strlen(parmp);
}

while ((labels= *imagev++) != nil) {
    /* Place each kernel image on the boot device. */

    if ((image= strchr(labels, ':')) != nil)
        *image++= 0;
    else {
        if (nolabel) {
            fprintf(stderr,
                "installboot: Only one image can be the default\n");
            exit(1);
        }
        nolabel= 1;
        image= labels;
        labels= nil;
    }
    len= 0;
    if (!raw_install(image, &pos, &len, block_size)) exit(1);

    if (labels == nil) {
        /* Let this image be the default. */
        sprintf(parmp, "image=%ld:%ld\n", pos-len, len);
        parmp+= strlen(parmp);
    }

    while (labels != nil) {
        /* Image is prefixed by a comma separated list of
         * labels. Define functions to select label and image.
         */
        label= labels;
        if ((labels= strchr(labels, ',')) != nil) *labels++ = 0;

        sprintf(parmp,
            "%s(%c){label=%s,image=%ld:%ld;echo %s kernel selected;menu}\\n",
                label,

```

```

        between('A', label[0], 'Z')
        ? label[0] - 'A' + 'a' : label[0],
        label, pos - len, len, label);
    parmp += strlen(parmp);
}

if (parmp > buf + block_size) {
    fprintf(stderr,
"installboot: Out of parameter space, too many images\n");
    exit(1);
}
}
/* Install boot block. */
writeblock((off_t) BOOTBLOCK, buf, 1024);

if (pos > fssize * RATIO(block_size)) {
    /* Tell the total size of the data on the device. */
    printf("%16ld (%ld kb) total\n", pos,
        (pos + RATIO(block_size) - 1) / RATIO(blo
ck_size));
}
}

void install_master(char *device, char *masterboot, char **guide)
/* Booting a hard disk is a two stage process: The master bootstrap in sector
 * 0 loads the bootstrap from sector 0 of the active partition which in turn
 * starts the operating system. This code installs such a master bootstrap
 * on a hard disk. If guide[0] is non-null then the master bootstrap is
 * guided into booting a certain device.
 */
{
    FILE *masf;
    unsigned long size;
    struct stat st;
    static char buf[_MAX_BLOCK_SIZE];

    /* Open device. */
    if ((rawfd = open(rawdev = device, O_RDWR)) < 0) fatal(device);

    /* Open the master boot code. */
    if ((masf = fopen(masterboot, "r")) == nil) fatal(masterboot);

    /* See if the user is cloning a device. */
    if (fstat(fileno(masf), &st) >= 0 && S_ISBLK(st.st_mode))
        size = PARTPOS;
    else {
        /* Read and check header otherwise. */
        struct image_header ihdr;

        read_header(1, masterboot, masf, &ihdr);
        size = ihdr.process.a_text + ihdr.process.a_data;
    }
    if (size > PARTPOS) {
        fprintf(stderr, "installboot: %s is too big\n", masterboot);
        exit(1);
    }

    /* Read the master boot block, patch it, write. */
    readblock(BOOTBLOCK, buf, BOOT_BLOCK_SIZE);

    memset(buf, 0, PARTPOS);
    (void) bread(masf, masterboot, buf, size);

    if (guide[0] != nil) {
        /* Fixate partition to boot. */
        char *keys = guide[0];
        char *logical = guide[1];
        size_t i;
        int logfd;
        u32_t offset;
        struct partition geometry;

        /* A string of digits to be seen as keystrokes. */
        i = 0;

```

```

        do {
            if (!between('0', keys[i], '9')) {
                fprintf(stderr,
                    "installboot: bad guide keys '%s'\n",
                    keys);
                exit(1);
            }
        } while (keys[++i] != 0);

        if (size + i + 1 > PARTPOS) {
            fprintf(stderr,
                "installboot: not enough space after '%s' for '%s'\n",
                masterboot, keys);
            exit(1);
        }
        memcpy(buf + size, keys, i);
        size += i;
        buf[size] = '\r';

        if (logical != nil) {
            if ((logfd = open(logical, O_RDONLY)) < 0
                || ioctl(logfd, DIOCGETP, &geometry) < 0
            ) {
                fatal(logical);
            }
            offset = div64u(geometry.base, SECTOR_SIZE);
            if (size + 5 > PARTPOS) {
                fprintf(stderr,
                    "installboot: not enough space "
                    "after '%s' for '%s' and an offset "
                    "to '%s'\n",
                    masterboot, keys, logical);
                exit(1);
            }
            buf[size] = '#';
            memcpy(buf+size+1, &offset, 4);
        }
    }

    /* Install signature. */
    buf[SIGPOS+0] = (SIGNATURE >> 0) & 0xFF;
    buf[SIGPOS+1] = (SIGNATURE >> 8) & 0xFF;

    writeblock(BOOTBLOCK, buf, BOOT_BLOCK_SIZE);
}

void usage(void)
{
    fprintf(stderr,
        "Usage: installboot -i(image) image kernel mm fs ... init\n"
        "       installboot -e(xtract) image\n"
        "       installboot -d(evice) device bootblock boot [image ...]\n"
        "       installboot -b(oot) device bootblock boot image ...\n"
        "       installboot -m(aster) device masterboot [keys [logical]]\n" );
    exit(1);
}

int isoption(char *option, char *test)
/* Check if the option argument is equals "test". Also accept -i as short
 * for -image, and the special case -x for -extract.
 */
{
    if (strcmp(option, test) == 0) return 1;
    if (option[0] != '-' && strlen(option) != 2) return 0;
    if (option[1] == test[1]) return 1;
    if (option[1] == 'x' && test[1] == 'e') return 1;
    return 0;
}

int main(int argc, char **argv)
{
    if (argc < 2) usage();

    if (argc >= 4 && isoption(argv[1], "-image")) {

```

```
        make_image(argv[2], argv + 3);
    } else
    if (argc == 3 && isoption(argv[1], "-extract")) {
        extract_image(argv[2]);
    } else
    if (argc >= 5 && isoption(argv[1], "-device")) {
        make_bootable(FS, argv[2], argv[3], argv[4], argv + 5);
    } else
    if (argc >= 6 && isoption(argv[1], "-boot")) {
        make_bootable(BOOT, argv[2], argv[3], argv[4], argv + 5);
    } else
    if ((4 <= argc && argc <= 6) && isoption(argv[1], "-master")) {
        install_master(argv[2], argv[3], argv + 4);
    } else {
        usage();
    }
    exit(0);
}

/*
 * $PchId: installboot.c,v 1.10 2000/08/13 22:07:50 philip Exp $
 */
```

```

!      jumpboot 1.0 - Jump to another bootstrap      Author: Kees J. Bot
!                                                    14 Apr 1999
!
! This code may be placed into any free boot sector, like the first sector
! of an extended partition, a file system partition other than the root,
! or even the master bootstrap. It will load and run another bootstrap whose
! disk, partition, and slice number (not necessarily all three) are patched
! into this code by installboot. If the ALT key is held down when this code
! is booted then you can type the disk, partition, and slice numbers manually.
! The manual interface is default if no numbers are patched in by installboot.
!

      o32          =      0x66      ! This assembler doesn't know 386 extensions
      LOADOFF      =      0x7C00    ! 0x0000:LOADOFF is where this code is loaded
      BUFFER       =      0x0600    ! First free memory
      PART_TABLE   =      446       ! Location of partition table within master
      PENTRYSIZE   =      16        ! Size of one partition table entry
      MAGIC        =      510       ! Location of the AA55 magic number

      ! <ibm/partition.h>:
      MINIX_PART   =      0x81
      sysind       =      4
      lowsec       =      8

.text

! Find and load another bootstrap and jump to it.
jumpboot:
      xor          ax, ax
      mov          ds, ax
      mov          es, ax
      cli
      mov          ss, ax                ! ds = es = ss = Vector segment
      mov          sp, #LOADOFF
      sti

! Move this code to safety, then jump to it.
      mov          si, sp                ! si = start of this code
      mov          di, #BUFFER          ! di = Buffer area
      mov          cx, #512/2          ! One sector
      cld
      rep          movs
      jmpf         BUFFER+migrate, 0     ! To safety
migrate:

      mov          bp, #BUFFER+guide    ! Patched guiding characters
altkey:
      movb         ah, #0x02            ! Keyboard shift status
      int          0x16
      testb        al, #0x08            ! Bit 3 = ALT key
      jz           noalt                ! ALT key pressed?
again:
      mov          bp, #zero            ! Ignore patched stuff
noalt:

! Follow guide characters to find the boot partition.
      call         print
      .ascii       "d?\b\0"            ! Initial greeting

! Disk number?
disk:
      movb         dl, #0x80 - 0x30     ! Prepare to add an ASCII digit
      call         getch                ! Get number to tell which disk
      addb         dl, al                ! dl = 0x80 + (al - '0')
      jns          n0nboot              ! Result should be ≥ 0x80
      mov          si, #BUFFER+zero-lowsec ! si = where lowsec(si) is zero
      cmpb         (bp), #0x23          ! Next guide character is '#'?
      jne          notlogical
      lea          si, 1-lowsec(bp)     ! Logical sector offset follows '#'
notlogical:
      call         load                  ! Load chosen sector of chosen disk
      cmpb         (bp), #0x23
      je           boot                 ! Run bootstrap if a logical is chosen

```

```

        call    print                ! Intro to partition number
        .ascii  "p?\b\0"

part:
        call    getch                ! Get character to tell partition
        call    gettable             ! Get partition table
        call    sort                 ! Sort partition table
        call    choose_load          ! Compute chosen entry and load

        cmpb    sysind(si), #MINIX_PART ! Minix subpartition table possible?
        jne     waitboot

        call    print                ! Intro to slice number
        .ascii  "s?\b\0"

slice:
        call    getch                ! Get character to tell slice
        call    gettable             ! Get partition table
        call    choose_load          ! Compute chosen entry and load

waitboot:
        call    print                ! Intro to nothing
        .ascii  "?\b\0"
        call    getch                ! Supposed to type RETURN now
n0nboot: jmp     nonboot              ! Sorry, can't go further

! Get a character, either the patched-in, or one from the keyboard.
getch:
        movb    al, (bp)              ! Get patched-in character
        testb   al, al
        jz      getkey
        inc     bp
        jmp     gotkey
getkey: xorb    ah, ah                ! Wait for keypress
        int     0x16
gotkey: testb   dl, dl                ! Ignore CR if disk number not yet set
        jns     putch
        cmpb    al, #0x0D             ! Carriage return?
        je      boot
        ! jmp    putch

! Print a character
putch:  movb    ah, #0x0E              ! Print character in teletype mode
        mov     bx, #0x0001           ! Page 0, foreground color
        int     0x10
        ret

! Print a message.
print:  mov     cx, si                 ! Save si
        pop     si                    ! si = String following 'call print'
prnext: lodsb                          ! al = *si++ is char to be printed
        testb   al, al                ! Null marks end
        jz      prdone
        call    putch
        jmp     prnext
prdone: xchg    si, cx                 ! Restore si
        jmp     (cx)                  ! Continue after the string

! Return typed (or in patched data) means to run the bootstrap now in core!
boot:
        call    print                ! Make line on screen look proper
        .ascii  "\b \r\n\0"
        jmp     LOADOFF-BUFFER       ! Jump to LOADOFF

! Compute address of chosen partition entry from choice al into si, then
! continue to load the boot sector of that partition.
choose_load:
        subb    al, #0x30              ! al -= '0'
        cmpb    al, #4                 ! Only four partitions
        ja      n0nboot
        movb    ah, #PENTRYSIZE
        mulb    ah                     ! al *= PENTRYSIZE
        add     ax, #BUFFER+PART_TABLE

```

```

mov     si, ax                ! si = &part_table[al - '0']
movb    al, sysind(si)        ! System indicator
testb   al, al                ! Unused partition?
jz      n0nboot
! jmp    load                  ! Continue to load boot sector

```

! Load boot sector of the current partition.

load:

```

push    dx                    ! Save drive code
push    es                    ! Next call sets es
movb    ah, #0x08              ! Code for drive parameters
int     0x13
pop     es
andb    cl, #0x3F              ! cl = max sector number (1-origin)
incb    dh                    ! dh = 1 + max head number (0-origin)
movb    al, cl                 ! al = cl = sectors per track
mulb    dh                    ! dh = heads, ax = heads * sectors
mov     bx, ax                 ! bx = sectors per cylinder = heads * sectors
mov     ax, lowsec+0(si)
mov     dx, lowsec+2(si)       ! dx:ax = sector within drive
cmp     dx, #[1024*255*63-255]>>16 ! Near 8G limit?
jae     bigdisk
div     bx                    ! ax = cylinder, dx = sector within cylinder
xchg    ax, dx                ! ax = sector within cylinder, dx = cylinder
movb    ch, dl                ! ch = low 8 bits of cylinder
divb    cl                    ! al = head, ah = sector (0-origin)
xorb    dl, dl                ! About to shift bits 8-9 of cylinder into dl
shr     dx, #1
shr     dx, #1                ! dl[6..7] = high cylinder
orb     dl, ah                ! dl[0..5] = sector (0-origin)
movb    cl, dl                ! cl[0..5] = sector, cl[6..7] = high cyl
incb    cl                    ! cl[0..5] = sector (1-origin)
pop     dx                    ! Restore drive code in dl
movb    dh, al                ! dh = al = head
mov     bx, #LOADOFF          ! es:bx = where sector is loaded
mov     ax, #0x0201           ! ah = Code for read / al = one sector
int     0x13
jmp     rdeval                ! Evaluate read result

```

bigdisk:

```

mov     bx, dx                ! bx:ax = dx:ax = sector to read
pop     dx                    ! Restore drive code in dl
push    si                    ! Save si
mov     si, #BUFFER+ext_rw    ! si = extended read/write parameter packet
mov     8(si), ax              ! Starting block number = bx:ax
mov     10(si), bx
movb    ah, #0x42              ! Extended read
int     0x13
pop     si                    ! Restore si to point to partition entry
! jmp    rdeval

```

rdeval:

```
jnc     rdok
```

rderr:

```

call    print
.ascii  "\r\nRead error\r\n\0"
jmp     again

```

rdok:

```

cmp     LOADOFF+MAGIC, #0xAA55
je      sigok                 ! Signature ok?

```

nonboot:

```

call    print
.ascii  "\r\nNot bootable\r\n\0"
jmp     again

```

sigok:

```
ret
```

! Get the partition table into my space.

gettable:

```

mov     si, #LOADOFF+PART_TABLE
mov     di, #BUFFER+PART_TABLE
mov     cx, #4*PENTRYSIZE/2
rep     movs
ret

```

! Sort the partition table.

```

sort:
    mov     cx, #4                ! Four times is enough to sort
bubble: mov     si, #BUFFER+PART_TABLE ! First table entry
bubble1: lea     di, PENTRYSIZE(si)    ! Next entry
        cmpb    sysind(si), ch        ! Partition type, nonzero when in use
        jz      exchg                ! Unused entries sort to the end
inuse:  mov     bx, lowsec+0(di)
        sub     bx, lowsec+0(si)      ! Compute di→lowsec - si→lowsec
        mov     bx, lowsec+2(di)
        sbb     bx, lowsec+2(si)
        jae     order                ! In order if si→lowsec ≤ di→lowsec
exchg:  movb     bl, (si)
        xchgb    bl, PENTRYSIZE(si)   ! Exchange entries byte by byte
        movb     (si), bl
        inc      si
        cmp      si, di
        jb      exchg
order:  mov     si, di
        cmp      si, #BUFFER+PART_TABLE+3*PENTRYSIZE
        jb      bubble1
        loop     bubble
        ret

```

.data

! Extended read/write commands require a parameter packet.

```

ext_rw:
    .data1    0x10                ! Length of extended r/w packet
    .data1     0                  ! Reserved
    .data2     1                  ! Blocks to transfer (just one)
    .data2    LOADOFF             ! Buffer address offset
    .data2     0                  ! Buffer address segment
    .data4     0                  ! Starting block number low 32 bits (tbfi)
zero:  .data4     0                  ! Starting block number high 32 bits

```

.align 2

guide:

! Guide characters and possibly a logical partition number patched here by  
! installboot, up to 6 bytes maximum.



```
# Makefile for the boot monitor package.
```

```
SYS      = ..
```

```
CC       = exec cc
```

```
CC86     = exec cc -mi86 -Was-ncc
```

```
CFLAGS   = -I$(SYS)
```

```
LIBS     = -lsys
```

```
LD       = $(CC) -s -.o
```

```
LD86     = $(CC86) -.o
```

```
BIN      = /usr/bin
```

```
MDEC     = /usr/mdec
```

```
all:      bootblock boot edparams masterboot jumpboot installboot addaout
```

```
dos:      boot.com mkfile.com
```

```
bootblock:      bootblock.s  
                $(LD86) -com -o $@ bootblock.s
```

```
masterboot:     masterboot.s  
                $(LD86) -com -o $@ masterboot.s
```

```
jumpboot:       jumpboot.s  
                $(LD86) -com -o $@ jumpboot.s
```

```
boot.o:         boot.c  
                $(CC86) $(CFLAGS) -c boot.c
```

```
bootimage.o:    bootimage.c  
                $(CC86) $(CFLAGS) -c bootimage.c
```

```
rawfs86.o:      rawfs.c rawfs.o  
                ln -f rawfs.c rawfs86.c  
                $(CC86) $(CFLAGS) -c rawfs86.c  
                rm rawfs86.c  
                -cmp -s rawfs.o rawfs86.o && ln -f rawfs.o rawfs86.o
```

```
boot:           boothead.s boot.o bootimage.o rawfs86.o  
                $(LD86) -o $@ \  
                boothead.s boot.o bootimage.o rawfs86.o $(LIBS)  
                install -S 8kb boot
```

```
edparams.o:     boot.c  
                ln -f boot.c edparams.c  
                $(CC) $(CFLAGS) -DUNIX -c edparams.c  
                rm edparams.c
```

```
edparams:       edparams.o rawfs.o  
                $(CC) $(CFLAGS) $(STRIP) -o $@ edparams.o rawfs.o  
                install -S 16kw edparams
```

```
dosboot.o:      boot.c  
                $(CC86) $(CFLAGS) -DDOS -o $@ -c boot.c
```

```
doshead.o:      doshead.s  
                $(CC) -mi386 -o $@ -c doshead.s
```

```
dosboot:         doshead.o dosboot.o bootimage.o rawfs86.o  
                $(LD86) -com -o $@ \  
                doshead.o dosboot.o bootimage.o rawfs86.o $(LIBS)
```

```
boot.com:        dosboot  
                ./a.out2com dosboot boot.com
```

```
mkfile:          mkfhead.s mkfile.c  
                $(LD) -.o -mi86 -com -o $@ mkfhead.s mkfile.c $(LIBS)
```

```
mkfile.com:      mkfile  
                ./a.out2com mkfile mkfile.com
```

```
installboot:     installboot.o rawfs.o  
                $(CC) $(STRIP) -o installboot installboot.o rawfs.o  
                install -S 6kw installboot
```

```
addaout:      addaout.o
              $(CC) -o addaout addaout.o

installboot.o bootimage.o: image.h
boot.o bootimage.o dosboot.o edparams.o: boot.h
rawfs.o rawfs86.o installboot.o boot.o bootimage.o: rawfs.h

install:      $(MDEC)/bootblock $(MDEC)/boot $(MDEC)/masterboot \
              $(MDEC)/jumpboot $(BIN)/installboot $(BIN)/edparams
dosinstall:   $(MDEC)/boot.com $(MDEC)/mkfile.com

$(MDEC)/bootblock:      bootblock
                       install -cs -o bin -m 644 $? @$

$(MDEC)/boot:           boot
                       install -cs -o bin -m 644 $? @$

$(MDEC)/boot.com:       boot.com
                       install -c -m 644 $? @$

$(MDEC)/mkfile.com:     mkfile.com
                       install -c -m 644 $? @$

$(MDEC)/masterboot:     masterboot
                       install -cs -o bin -m 644 $? @$

$(MDEC)/jumpboot:       jumpboot
                       install -cs -o bin -m 644 $? @$

$(BIN)/installboot:     installboot
                       install -cs -o bin $? @$

$(BIN)/addaout: addaout
                       install -cs -o bin $? @$

$(BIN)/edparams:        edparams
                       install -cs -o bin $? @$

clean:
rm -f *.bak *.o
rm -f bootblock addaout installboot boot masterboot jumpboot edparams
rm -f dosboot boot.com mkfile mkfile.com
```

```

!      masterboot 2.0 - Master boot block code      Author: Kees J. Bot
!
! This code may be placed in the first sector (the boot sector) of a floppy,
! hard disk or hard disk primary partition. There it will perform the
! following actions at boot time:
!
! - If the booted device is a hard disk and one of the partitions is active
!   then the active partition is booted.
!
! - Otherwise the next floppy or hard disk device is booted, trying them one
!   by one.
!
! To make things a little clearer, the boot path might be:
!   /dev/fd0      - Floppy disk containing data, tries fd1 then d0
!   [/dev/fd1]    - Drive empty
!   /dev/c0d0     - Master boot block, selects active partition 2
!   /dev/c0d0p2   - Submaster, selects active subpartition 0
!   /dev/c0d0p2s0 - Minix bootblock, reads Boot Monitor /boot
!   Minix        - Started by /boot from a kernel image in /minix
!
! LOADOFF = 0x7C00 ! 0x0000:LOADOFF is where this code is loaded
! BUFFER  = 0x0600 ! First free memory
! PART_TABLE = 446 ! Location of partition table within this code
! PENTRYSIZE = 16  ! Size of one partition table entry
! MAGIC    = 510  ! Location of the AA55 magic number
!
! <ibm/partition>.h:
! bootind  = 0
! sysind   = 4
! lowsec   = 8
!
.text
! Find active (sub)partition, load its first sector, run it.
master:
    xor     ax, ax
    mov     ds, ax
    mov     es, ax
    cli
    mov     ss, ax          ! ds = es = ss = Vector segment
    mov     sp, #LOADOFF
    sti

! Copy this code to safety, then jump to it.
    mov     si, sp          ! si = start of this code
    push    si              ! Also where we'll return to eventually
    mov     di, #BUFFER     ! Buffer area
    mov     cx, #512/2      ! One sector
    cld
    rep     movs
    jmpf     BUFFER+migrate, 0 ! To safety
migrate:

! Find the active partition
findactive:
    testb   dl, dl
    jns     nextdisk        ! No bootable partitions on floppies
    mov     si, #BUFFER+PART_TABLE
find:      cmpb   sysind(si), #0 ! Partition type, nonzero when in use
    jz      nextpart
    testb   bootind(si), #0x80 ! Active partition flag in bit 7
    jz      nextpart        ! It's not active
loadpart:
    call    load            ! Load partition bootstrap
    jc      error1          ! Not supposed to fail
bootstrap:
    ret                  ! Jump to the master bootstrap
nextpart:
    add     si, #PENTRYSIZE
    cmp     si, #BUFFER+PART_TABLE+4*PENTRYSIZE
    jb      find
! No active partition, tell 'em

```

```

    call    print
    .ascii  "No active partition\0"
    jmp     reboot

! There are no active partitions on this drive, try the next drive.
nextdisk:
    incb    dl                      ! Increment dl for the next drive
    testb   dl, dl
    js      nexthd                  ! Hard disk if negative
    int     0x11                   ! Get equipment configuration
    shl     ax, #1                  ! Highest floppy drive # in bits 6-7
    shl     ax, #1                  ! Now in bits 0-1 of ah
    andb    ah, #0x03              ! Extract bits
    cmpb    dl, ah                 ! Must be dl ≤ ah for drive to exist
    ja      nextdisk              ! Otherwise try disk 0 eventually
    call    load0                  ! Read the next floppy bootstrap
    jc      nextdisk              ! It failed, next disk please
    ret                                           ! Jump to the next master bootstrap
nexthd:  call    load0              ! Read the hard disk bootstrap
error1:  jc      error              ! No disk?
    ret

! Load sector 0 from the current device.  It's either a floppy bootstrap or
! a hard disk master bootstrap.
load0:
    mov     si, #BUFFER+zero-lowsec ! si = where lowsec(si) is zero
    ! jmp    load

! Load sector lowsec(si) from the current device.  The obvious head, sector,
! and cylinder numbers are ignored in favour of the more trustworthy absolute
! start of partition.
load:
    mov     di, #3                  ! Three retries for floppy spinup
retry:    push    dx                ! Save drive code
    push    es
    push    di                      ! Next call destroys es and di
    movb    ah, #0x08              ! Code for drive parameters
    int     0x13
    pop     di
    pop     es
    andb    cl, #0x3F              ! cl = max sector number (1-origin)
    incb    dh                     ! dh = 1 + max head number (0-origin)
    movb    al, cl                  ! al = cl = sectors per track
    mulb    dh                     ! dh = heads, ax = heads * sectors
    mov     bx, ax                  ! bx = sectors per cylinder = heads * sectors
    mov     ax, lowsec+0(si)
    mov     dx, lowsec+2(si) ! dx:ax = sector within drive
    cmp     dx, #[1024*255*63-255]>>16 ! Near 8G limit?
    jae     bigdisk
    div     bx                      ! ax = cylinder, dx = sector within cylinder
    xchg    ax, dx                 ! ax = sector within cylinder, dx = cylinder
    movb    ch, dl                 ! ch = low 8 bits of cylinder
    divb    cl                     ! al = head, ah = sector (0-origin)
    xorb    dl, dl                 ! About to shift bits 8-9 of cylinder into dl
    shr     dx, #1
    shr     dx, #1                 ! dl[6..7] = high cylinder
    orb     dl, ah                 ! dl[0..5] = sector (0-origin)
    movb    cl, dl                 ! cl[0..5] = sector, cl[6..7] = high cyl
    incb    cl                     ! cl[0..5] = sector (1-origin)
    pop     dx                      ! Restore drive code in dl
    movb    dh, al                 ! dh = al = head
    mov     bx, #LOADOFF           ! es:bx = where sector is loaded
    mov     ax, #0x0201            ! Code for read, just one sector
    int     0x13                  ! Call the BIOS for a read
    jmp     rdeval                 ! Evaluate read result
bigdisk:
    mov     bx, dx                 ! bx:ax = dx:ax = sector to read
    pop     dx                     ! Restore drive code in dl
    push    si                     ! Save si
    mov     si, #BUFFER+ext_rw ! si = extended read/write parameter packet
    mov     8(si), ax              ! Starting block number = bx:ax
    mov     10(si), bx
    movb    ah, #0x42              ! Extended read

```

```

    int     0x13
    pop     si                ! Restore si to point to partition entry
    ! jmp    rdeval
rdeval:
    jnc     rdok              ! Read succeeded
    cmpb    ah, #0x80         ! Disk timed out? (Floppy drive empty)
    je      rdbad
    dec     di
    jl      rdbad            ! Retry count expired
    xorb    ah, ah
    int     0x13             ! Reset
    jnc     retry            ! Try again
rdbad:
    stc
    ret      ! Set carry flag
rdok:
    cmp     LOADOFF+MAGIC, #0xAA55
    jne     nosig            ! Error if signature wrong
    ret      ! Return with carry still clear
nosig:
    call    print
    .ascii  "Not bootable\0"
    jmp     reboot

! A read error occurred, complain and hang
error:
    mov     si, #LOADOFF+errno+1
prnum:
    movb    al, ah           ! Error number in ah
    andb    al, #0x0F        ! Low 4 bits
    cmpb    al, #10          ! A-F?
    jb      digit            ! 0-9!
    addb    al, #7            ! 'A' - ':'
digit:
    addb    (si), al          ! Modify '0' in string
    dec     si
    movb    cl, #4           ! Next 4 bits
    shrb    ah, cl
    jnz     prnum            ! Again if digit > 0
    call    print
    .ascii  "Read error "
errno:
    .ascii  "00\0"
    ! jmp    reboot

reboot:
    call    print
    .ascii  ". Hit any key to reboot.\0"
    xorb    ah, ah           ! Wait for keypress
    int     0x16
    call    print
    .ascii  "\r\n\0"
    int     0x19

! Print a message.
print:
    pop     si                ! si = String following 'call print'
prnext:
    lodsb
    testb   al, al           ! al = *si++ is char to be printed
    jz      prdone           ! Null marks end
    movb    ah, #0x0E        ! Print character in teletype mode
    mov     bx, #0x0001       ! Page 0, foreground color
    int     0x10
    jmp     prnext
prdone:
    jmp     (si)             ! Continue after the string

.data

! Extended read/write commands require a parameter packet.
ext_rw:
    .data1  0x10             ! Length of extended r/w packet
    .data1  0                ! Reserved
    .data2  1                ! Blocks to transfer (just one)
    .data2  LOADOFF          ! Buffer address offset
    .data2  0                ! Buffer address segment
    .data4  0                ! Starting block number low 32 bits (tbfi)
zero:
    .data4  0                ! Starting block number high 32 bits

```

```

!      Mkfhead.s - DOS & BIOS support for mkfile.c      Author: Kees J. Bot
!                                                         9 May 1998
!
! This file contains the startup and low level support for the MKFILE.COM
! utility. See doshead.ack.s for more comments on .COM files.
!
.sect .text; .sect .rom; .sect .data; .sect .bss
.sect .text

.define _PSP
_PSP:
    .space 256                ! Program Segment Prefix

mkfile:
    cld                      ! C compiler wants UP
    xor ax, ax                ! Zero
    mov di, _edata            ! Start of bss is at end of data
    mov cx, _end              ! End of bss (begin of heap)
    sub cx, di                ! Number of bss bytes
    shr cx, 1                 ! Number of words
    rep stos                  ! Clear bss

    xor cx, cx                ! cx = argc
    xor bx, bx
    push bx                   ! argv[argc] = NULL
    movb bl, (_PSP+0x80)      ! Argument byte count
0:   movb _PSP+0x81(bx), ch    ! Null terminate
    dec bx
    js 9f
    cmpb _PSP+0x81(bx), 0x20   ! Whitespace?
    jbe 0b
1:   dec bx                    ! One argument character
    js 2f
    cmpb _PSP+0x81(bx), 0x20   ! More argument characters?
    ja 1b
2:   lea ax, _PSP+0x81+1(bx)   ! Address of argument
    push ax                   ! argv[n]
    inc cx                    ! argc++;
    test bx, bx
    jns 0b                    ! More arguments?
9:   movb _PSP+0x81(bx), ch    ! Make a null string
    lea ax, _PSP+0x81(bx)
    push ax                   ! to use as argv[0]
    inc cx                    ! Final value of argc
    mov ax, sp
    push ax                   ! argv
    push cx                   ! argc
    call _main                 ! main(argc, argv)
    push ax
    call _exit                 ! exit(main(argc, argv))

! int creat(const char *path, mode_t mode)
! Create a file with the old creat() call.
.define _creat
_creat:
    mov bx, sp
    mov dx, 2(bx)             ! Filename
    xor cx, cx                ! Ignore mode, always read-write
    movb ah, 0x3C             ! "CREAT"
dos:  int 0x21                 ! ax = creat(path, 0666);
    jc seterrno
    ret

seterrno:
    mov (_errno), ax          ! Set errno to the DOS error code
    mov ax, -1
    cwd                       ! return -1L;
    ret

! int open(const char *path, int oflag)
! Open a file with the oldfashioned two-argument open() call.
.define _open
_open:
    mov bx, sp

```

```

        mov     dx, 2(bx)           ! Filename
        movb    al, 4(bx)           ! O_RDONLY, O_WRONLY, O_RDWR
        movb    ah, 0x3D            ! "OPEN"
        jmp     dos

! int close(int fd)
!     Close an open file.
.define _close
_close:
        mov     bx, sp
        mov     bx, 2(bx)           ! bx = file handle
        movb    ah, 0x3E            ! "CLOSE"
        jmp     dos

! void exit(int status)
! void _exit(int status)
!     Return to DOS.
.define _exit, __exit, ___exit
_exit:
__exit:
___exit:
        pop     ax
        pop     ax                 ! al = status
        movb    ah, 0x4C            ! "EXIT"
        int     0x21
        hlt

! ssize_t read(int fd, void *buf, size_t n)
!     Read bytes from an open file.
.define _read
_read:
        mov     bx, sp
        mov     cx, 6(bx)
        mov     dx, 4(bx)
        mov     bx, 2(bx)
        movb    ah, 0x3F            ! "READ"
        jmp     dos

! ssize_t write(int fd, const void *buf, size_t n)
!     Write bytes to an open file.
.define _write
_write:
        mov     bx, sp
        mov     cx, 6(bx)
        mov     dx, 4(bx)
        mov     bx, 2(bx)
        movb    ah, 0x40            ! "WRITE"
        jmp     dos

! off_t lseek(int fd, off_t offset, int whence)
!     Set file position for read or write.
.define _lseek
_lseek:
        mov     bx, sp
        movb    al, 8(bx)           ! SEEK_SET, SEEK_CUR, SEEK_END
        mov     dx, 4(bx)
        mov     cx, 6(bx)           ! cx:dx = offset
        mov     bx, 2(bx)
        movb    ah, 0x42            ! "LSEEK"
        jmp     dos

!
! $PchId: mkfhead.ack.s,v 1.3 1999/01/14 21:17:06 philip Exp $

```

```

/*      mkfile 1.0 - create a file under DOS for use as a Minix "disk".
 *
 *      Author: Kees J. Bot
 *      9 May 1998
 */
#define nil 0
#include <sys/types.h>
#include <string.h>
#include <limits.h>

/* Stuff normally found in <unistd.h>, <errno.h>, etc. */
extern int errno;
int creat(const char *file, int mode);
int open(const char *file, int oflag);
off_t lseek(int fd, off_t offset, int whence);
ssize_t write(int fd, const char *buf, size_t len);
void exit(int status);
int printf(const char *fmt, ...);

#define O_WRONLY      1
#define SEEK_SET      0
#define SEEK_END      2

/* Kernel printf requires a putk() function. */
int putk(int c)
{
    char ch = c;

    if (c == 0) return;
    if (c == '\n') putk('\r');
    (void) write(2, &ch, 1);
}

static void usage(void)
{
    printf("Usage: mkfile <size>[gmk] <file>\n"
           "(Example sizes, all 50 meg: 52428800, 51200k, 50m)\n");
    exit(1);
}

char *strerror(int err)
/* Translate some DOS error numbers to text. */
{
    static struct errlist {
        int      err;
        char      *what;
    } errlist[] = {
        { 0, "No error" },
        { 1, "Function number invalid" },
        { 2, "File not found" },
        { 3, "Path not found" },
        { 4, "Too many open files" },
        { 5, "Access denied" },
        { 6, "Invalid handle" },
        { 12, "Access code invalid" },
        { 39, "Insufficient disk space" },
    };
    struct errlist *ep;
    static char unknown[] = "Error 65535";
    unsigned e;
    char *p;

    for (ep= errlist; ep < errlist + sizeof(errlist)/sizeof(errlist[0]); ep++) {
        if (ep->err == err) return ep->what;
    }
    p= unknown + sizeof(unknown) - 1;
    e= err;
    do *--p= '0' + (e % 10); while ((e /= 10) > 0);
    strcpy(unknown + 6, p);
    return unknown;
}

int main(int argc, char **argv)
{

```



```
int i;
static char buf[512];
unsigned long size, mul;
off_t offset;
char *cp;
int fd;
char *file;

if (argc != 3) usage();

cp= argv[1];
size= 0;
while ((unsigned) (*cp - '0') < 10) {
    unsigned d= *cp++ - '0';
    if (size <= (ULONG_MAX-9) / 10) {
        size= size * 10 + d;
    } else {
        size= ULONG_MAX;
    }
}
if (cp == argv[1]) usage();
while (*cp != 0) {
    mul = 1;
    switch (*cp++) {
        case 'G':
        case 'g':      mul *= 1024;
        case 'M':
        case 'm':      mul *= 1024;
        case 'K':
        case 'k':      mul *= 1024;
        case 'B':
        case 'b':      break;
        default:      usage();
    }
    if (size <= ULONG_MAX / mul) {
        size *= mul;
    } else {
        size= ULONG_MAX;
    }
}

if (size > 1024L*1024*1024) {
    printf("mkfile: A file size over 1G is a bit too much\n");
    exit(1);
}

/* Open existing file, or create a new file. */
file= argv[2];
if ((fd= open(file, O_WRONLY)) < 0) {
    if (errno == 2) {
        fd= creat(file, 0666);
    }
}
if (fd < 0) {
    printf("mkfile: Can't open %s: %s\n", file, strerror(errno));
    exit(1);
}

/* How big is the file now? */
if ((offset= lseek(fd, 0, SEEK_END)) == -1) {
    printf("mkfile: Can't seek in %s: %s\n", file, strerror(errno));
    exit(1);
}

if (offset == 0 && size == 0) exit(0); /* Huh? */

/* Write the first bit if the file is zero length. This is necessary
 * to circumvent a DOS bug by extending a new file by lseek. We also
 * want to make sure there are zeros in the first sector.
 */
if (offset == 0) {
    if (write(fd, buf, sizeof(buf)) == -1) {
        printf("mkfile: Can't write to %s: %s\n",
            file, strerror(errno));
    }
}
```

```
        exit(1);
    }
}

/* Seek to the required size and write 0 bytes to extend/truncate the
 * file to that size.
 */
if (lseek(fd, size, SEEK_SET) == -1) {
    printf("mkfile: Can't seek in %s: %s\n", file, strerror(errno));
    exit(1);
}
if (write(fd, buf, 0) == -1) {
    printf("mkfile: Can't write to %s: %s\n",
        file, strerror(errno));
    exit(1);
}

/* Did the file become the required size? */
if ((offset= lseek(fd, 0, SEEK_END)) == -1) {
    printf("mkfile: Can't seek in %s: %s\n", file, strerror(errno));
    exit(1);
}
if (offset != size) {
    printf("mkfile: Failed to extend %s. Disk full?\n", file);
    exit(1);
}
return 0;
}

/*
 * $PchId: mkfile.c,v 1.4 2000/08/13 22:06:40 philip Exp $
 */
```

```

/*      rawfs.c - Raw Minix file system support.      Author: Kees J. Bot
*                                                    23 Dec 1991
*                                                    Based on readfs by Paul Polderman
*/
#define nil 0
#define _POSIX_SOURCE      1
#define _MINIX              1
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include <errno.h>
#include <minix/config.h>
#include <minix/const.h>
#include <minix/type.h>
#include <servers/fs/const.h>
#include <servers/fs/type.h>
#include <servers/fs/buf.h>
#include <servers/fs/super.h>
#include <servers/fs/inode.h>
#include "rawfs.h"

void readblock(off_t blockno, char *buf, int);

/* The following code handles two file system types: Version 1 with small
 * inodes and 16-bit disk addresses and Version 2 with big inodes and 32-bit
 * disk addresses.
#ifdef FLEX
 * To make matters worse, Minix-vmd knows about the normal Unix Version 7
 * directories and directories with flexible entries.
#endif
*/

/* File system parameters. */
static unsigned nr_dzones;      /* Fill these in after reading superblock. */
static unsigned nr_indirects;
static unsigned inodes_per_block;
static int block_size;
#ifdef FLEX
#include <dirent.h>
#define direct _v7_direct
#else
#include <sys/dir.h>
#endif

#if __minix_vmd
static struct vl2_super_block super;      /* Superblock of file system */
#define s_log_zone_size s_dummy          /* Zones are obsolete. */
#else
static struct super_block super;          /* Superblock of file system */
#define SUPER_V1 SUPER_MAGIC             /* V1 magic has a weird name. */
#endif

static struct inode curfil;              /* Inode of file under examination */
static char indir[_MAX_BLOCK_SIZE];      /* Single indirect block. */
static char dindir[_MAX_BLOCK_SIZE];     /* Double indirect block. */
static char dirbuf[_MAX_BLOCK_SIZE];     /* Scratch/Directory block. */
#define scratch dirbuf

static block_t a_indir, a_dindir;        /* Addresses of the indirects. */
static off_t dirpos;                     /* Reading pos in a dir. */

#define fsbuf(b)      (* (struct buf *) (b))

#define zone_shift      (super.s_log_zone_size) /* zone to block ratio */

off_t r_super(int *bs)
/* Initialize variables, return size of file system in blocks,
 * (zero on error).
 */
{
    /* Read superblock. (The superblock is always at 1kB offset,
     * that's why we lie to readblock and say the block size is 1024

```

```

    * and we want block number 1 (the 'second block', at offset 1kB).)
    */
    readblock(1, scratch, 1024);

    memcpy(&super, scratch, sizeof(super));

    /* Is it really a MINIX file system ? */
    if (super.s_magic == SUPER_V2 || super.s_magic == SUPER_V3) {
        if (super.s_magic == SUPER_V2)
            super.s_block_size = 1024;
        *bs = block_size = super.s_block_size;
        if (block_size < _MIN_BLOCK_SIZE ||
            block_size > _MAX_BLOCK_SIZE) {
            return 0;
        }
        nr_dzones= V2_NR_DZONES;
        nr_indirects= V2_INDIRECTS(block_size);
        inodes_per_block= V2_INODES_PER_BLOCK(block_size);
        return (off_t) super.s_zones << zone_shift;
    } else
    if (super.s_magic == SUPER_V1) {
        *bs = block_size = 1024;
        nr_dzones= V1_NR_DZONES;
        nr_indirects= V1_INDIRECTS;
        inodes_per_block= V1_INODES_PER_BLOCK;
        return (off_t) super.s_nzones << zone_shift;
    } else {
        /* Filesystem not recognized as Minix. */
        return 0;
    }
}

```

```

void r_stat(Ino_t inum, struct stat *stp)
/* Return information about a file like stat(2) and remember it. */
{
    block_t block;
    block_t ino_block;
    ino_t ino_offset;

    /* Calculate start of i-list */
    block = START_BLOCK + super.s_imap_blocks + super.s_zmap_blocks;

    /* Calculate block with inode inum */
    ino_block = ((inum - 1) / inodes_per_block);
    ino_offset = ((inum - 1) % inodes_per_block);
    block += ino_block;

    /* Fetch the block */
    readblock(block, scratch, block_size);

    if (super.s_magic == SUPER_V2 || super.s_magic == SUPER_V3) {
        d2_inode *dip;
        int i;

        dip= &fsbuf(scratch).b_v2_ino[ino_offset];

        curfil.i_mode= dip->d2_mode;
        curfil.i_nlinks= dip->d2_nlinks;
        curfil.i_uid= dip->d2_uid;
        curfil.i_gid= dip->d2_gid;
        curfil.i_size= dip->d2_size;
        curfil.i_atime= dip->d2_atime;
        curfil.i_mtime= dip->d2_mtime;
        curfil.i_ctime= dip->d2_ctime;
        for (i= 0; i < V2_NR_TZONES; i++)
            curfil.i_zone[i]= dip->d2_zone[i];
    } else {
        d1_inode *dip;
        int i;

        dip= &fsbuf(scratch).b_v1_ino[ino_offset];

        curfil.i_mode= dip->d1_mode;
        curfil.i_nlinks= dip->d1_nlinks;
    }
}

```

```

        curfil.i_uid= dip->d1_uid;
        curfil.i_gid= dip->d1_gid;
        curfil.i_size= dip->d1_size;
        curfil.i_atime= dip->d1_mtime;
        curfil.i_mtime= dip->d1_mtime;
        curfil.i_ctime= dip->d1_mtime;
        for (i= 0; i < V1_NR_TZONES; i++)
            curfil.i_zone[i]= dip->d1_zone[i];
    }
    curfil.i_dev= -1;        /* Can't fill this in alas. */
    curfil.i_num= inum;

    stp->st_dev= curfil.i_dev;
    stp->st_ino= curfil.i_num;
    stp->st_mode= curfil.i_mode;
    stp->st_nlink= curfil.i_nlinks;
    stp->st_uid= curfil.i_uid;
    stp->st_gid= curfil.i_gid;
    stp->st_rdev= (dev_t) curfil.i_zone[0];
    stp->st_size= curfil.i_size;
    stp->st_atime= curfil.i_atime;
    stp->st_mtime= curfil.i_mtime;
    stp->st_ctime= curfil.i_ctime;

    a_indir= a_dindir= 0;
    dirpos= 0;
}

ino_t r_readdir(char *name)
/* Read next directory entry at "dirpos" from file "curfil". */
{
    ino_t inum= 0;
    int blkpos;
    struct direct *dp;

    if (!S_ISDIR(curfil.i_mode)) { errno= ENOTDIR; return -1; }

    if(!block_size) { errno = 0; return -1; }

    while (inum == 0 && dirpos < curfil.i_size) {
        if ((blkpos= (int) (dirpos % block_size)) == 0) {
            /* Need to fetch a new directory block. */

            readblock(r_vir2abs(dirpos / block_size), dirbuf, block_size);
        }
#ifdef FLEX
        if (super.s_flags & S_FLEX) {
            struct _fl_direct *dp;

            dp= (struct _fl_direct *) (dirbuf + blkpos);
            if ((inum= dp->d_ino) != 0) strcpy(name, dp->d_name);

            dirpos+= (1 + dp->d_extent) * FL_DIR_ENTRY_SIZE;
            continue;
        }
#endif
        /* Let dp point to the next entry. */
        dp= (struct direct *) (dirbuf + blkpos);

        if ((inum= dp->d_ino) != 0) {
            /* This entry is occupied, return name. */
            strncpy(name, dp->d_name, sizeof(dp->d_name));
            name[sizeof(dp->d_name)]= 0;
        }
        dirpos+= DIR_ENTRY_SIZE;
    }
    return inum;
}

off_t r_vir2abs(off_t virblk)
/* Translate a block number in a file to an absolute disk block number.
 * Returns 0 for a hole and -1 if block is past end of file.
 */
{

```

```

    block_t b= virblk;
    zone_t zone, ind_zone;
    block_t z, zone_index;
    int i;

    if(!block_size) return -1;

    /* Check if virblk within file. */
    if (virblk * block_size >= curfil.i_size) return -1;

    /* Calculate zone in which the datablock number is contained */
    zone = (zone_t) (b >> zone_shift);

    /* Calculate index of the block number in the zone */
    zone_index = b - ((block_t) zone << zone_shift);

    /* Go get the zone */
    if (zone < (zone_t) nr_dzones) {          /* direct block */
        zone = curfil.i_zone[(int) zone];
        z = ((block_t) zone << zone_shift) + zone_index;
        return z;
    }

    /* The zone is not a direct one */
    zone -= (zone_t) nr_dzones;

    /* Is it single indirect ? */
    if (zone < (zone_t) nr_indirects) {      /* single indirect block */
        ind_zone = curfil.i_zone[nr_dzones];
    } else {                                /* double indirect block */
        /* Fetch the double indirect block */
        if ((ind_zone = curfil.i_zone[nr_dzones + 1]) == 0) return 0;

        z = (block_t) ind_zone << zone_shift;
        if (a_dindir != z) {
            readblock(z, dindir, block_size);
            a_dindir= z;
        }
        /* Extract the indirect zone number from it */
        zone -= (zone_t) nr_indirects;

        i = zone / (zone_t) nr_indirects;
        ind_zone = (super.s_magic == SUPER_V2 || super.s_magic == SUPER_V3)
            ? fsbuf(dindir).b_v2_ind[i]
            : fsbuf(dindir).b_v1_ind[i];
        zone %= (zone_t) nr_indirects;
    }
    if (ind_zone == 0) return 0;

    /* Extract the datablock number from the indirect zone */
    z = (block_t) ind_zone << zone_shift;
    if (a_indir != z) {
        readblock(z, indir, block_size);
        a_indir= z;
    }
    zone = (super.s_magic == SUPER_V2 || super.s_magic == SUPER_V3)
        ? fsbuf(indir).b_v2_ind[(int) zone]
        : fsbuf(indir).b_v1_ind[(int) zone];

    /* Calculate absolute datablock number */
    z = ((block_t) zone << zone_shift) + zone_index;
    return z;
}

ino_t r_lookup(Ino_t cwd, char *path)
/* Translates a pathname to an inode number. This is just a nice utility
 * function, it only needs r_stat and r_readdir.
 */
{
    char name[NAME_MAX+1], r_name[NAME_MAX+1];
    char *n;
    struct stat st;
    ino_t ino;

```

```
ino= path[0] == '/' ? ROOT_INO : cwd;

for (;;) {
    if (ino == 0) {
        errno= ENOENT;
        return 0;
    }

    while (*path == '/') path++;

    if (*path == 0) return ino;

    r_stat(ino, &st);

    if (!S_ISDIR(st.st_mode)) {
        errno= ENOTDIR;
        return 0;
    }

    n= name;
    while (*path != 0 && *path != '/')
        if (n < name + NAME_MAX) *n++ = *path++;
    *n= 0;

    while ((ino= r_readdir(r_name)) != 0
           && strcmp(name, r_name) != 0) {
    }
}

/*
 * $PchId: rawfs.c,v 1.8 1999/11/05 23:14:15 philip Exp $
 */
```

```
/*      rawfs.h - Raw Minix file system support.      Author: Kees J. Bot
*
*      off_t r_super(int *block_size);
*          Initialize variables, returns the size of a valid Minix
*          file system blocks, but zero on error.
*
*      void r_stat(ino_t file, struct stat *stp);
*          Return information about a file like stat(2) and
*          remembers file for the next two calls.
*
*      off_t r_vir2abs(off_t virblockno);
*          Translate virtual block number in file to absolute
*          disk block number. Returns 0 if the file contains
*          a hole, or -1 if the block lies past the end of file.
*
*      ino_t r_readdir(char *name);
*          Return next directory entry or 0 if there are no more.
*          Returns -1 and sets errno on error.
*
*      ino_t r_lookup(ino_t cwd, char *path);
*          A utility function that translates a pathname to an
*          inode number. It starts from directory "cwd" unless
*          path starts with a '/', then from ROOT_INO.
*          Returns 0 and sets errno on error.
*
*      One function needs to be provided by the outside world:
*
*      void readblock(off_t blockno, char *buf, int block_size);
*          Read a block into the buffer. Outside world handles
*          errors.
*/

#define ROOT_INO      ((ino_t) 1)      /* Inode nr of root dir. */

off_t r_super(int *);
void r_stat(ino_t file, struct stat *stp);
off_t r_vir2abs(off_t virblockno);
ino_t r_readdir(char *name);
ino_t r_lookup(ino_t cwd, char *path);

/*
* $PchId: rawfs.h,v 1.4 1996/04/19 08:16:36 philip Exp $
*/
```



**Table of Contents**

1	<i>addaout.c</i> .....	sheets	1 to	2 ( 2)	pages	1-	2	129	lines
2	<i>a.out2com</i> .....	sheets	3 to	3 ( 1)	pages	3-	3	26	lines
3	<i>bootblock.s</i> .....	sheets	4 to	7 ( 4)	pages	4-	7	232	lines
4	<i>boot.c</i> .....	sheets	8 to	34 (27)	pages	8-	34	1958	lines
5	<i>boot.h</i> .....	sheets	35 to	37 ( 3)	pages	35-	37	213	lines
6	<i>boothead.s</i> .....	sheets	38 to	58 (21)	pages	38-	58	1514	lines
7	<i>bootimage.c</i> .....	sheets	59 to	68 (10)	pages	59-	68	724	lines
8	<i>doshead.s</i> .....	sheets	69 to	87 (19)	pages	69-	87	1370	lines
9	<i>image.h</i> .....	sheets	88 to	88 ( 1)	pages	88-	88	14	lines
10	<i>installboot.c</i> .....	sheets	89 to	100 (12)	pages	89-	100	834	lines
11	<i>jumpboot.s</i> .....	sheets	101 to	104 ( 4)	pages	101-	104	262	lines
12	<i>Makefile</i> .....	sheets	105 to	106 ( 2)	pages	105-	106	117	lines
13	<i>masterboot.s</i> .....	sheets	107 to	109 ( 3)	pages	107-	109	219	lines
14	<i>mkfhead.s</i> .....	sheets	110 to	111 ( 2)	pages	110-	111	138	lines
15	<i>mkfile.c</i> .....	sheets	112 to	114 ( 3)	pages	112-	114	181	lines
16	<i>rawfs.c</i> .....	sheets	115 to	119 ( 5)	pages	115-	119	330	lines
17	<i>rawfs.h</i> .....	sheets	120 to	120 ( 1)	pages	120-	120	44	lines