```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/a.out.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

00000    /* The <a.out> header file describes the format of executable files. */
00001
00002    #ifndef _AOUT_H
00003    #define _AOUT_H
00004
00005    struct  exec {                  /* a.out header */
00006      unsigned char a_magic[2];     /* magic number */
00007      unsigned char a_flags;        /* flags, see below */
00008      unsigned char a_cpu;          /* cpu id */
00009      unsigned char a_hdrlen;       /* length of header */
00010      unsigned char a_unused;       /* reserved for future use */
00011      unsigned short a_version;     /* version stamp (not used at present) */
00012      long          a_text;         /* size of text segement in bytes */
00013      long          a_data;         /* size of data segment in bytes */
00014      long          a_bss;          /* size of bss segment in bytes */
00015      long          a_entry;        /* entry point */
00016      long          a_total;        /* total memory allocated */
00017      long          a_syms;         /* size of symbol table */
00018
00019      /* SHORT FORM ENDS HERE */
00020      long          a_trsize;       /* text relocation size */
00021      long          a_drsize;       /* data relocation size */
00022      long          a_tbase;        /* text relocation base */
00023      long          a_dbase;        /* data relocation base */
00024    };
00025
00026    #define A_MAGIC0      (unsigned char) 0x01
00027    #define A_MAGIC1      (unsigned char) 0x03
00028    #define BADMAG(X)     ((X).a_magic[0] != A_MAGIC0 ||(X).a_magic[1] != A_MAGIC1)
00029
00030    /* CPU Id of TARGET machine (byte order coded in low order two bits) */
00031    #define A_NONE  0x00    /* unknown */
00032    #define A_I8086 0x04    /* intel i8086/8088 */
00033    #define A_M68K  0x0B    /* motorola m68000 */
00034    #define A_NS16K 0x0C    /* national semiconductor 16032 */
00035    #define A_I80386 0x10   /* intel i80386 */
00036    #define A_SPARC 0x17    /* Sun SPARC */
00037
00038    #define A_BLR(cputype)  ((cputype&0x01)!=0) /* TRUE if bytes left-to-right */
00039    #define A_WLR(cputype)  ((cputype&0x02)!=0) /* TRUE if words left-to-right */
00040
00041    /* Flags. */
00042    #define A_UZP   0x01    /* unmapped zero page (pages) */
00043    #define A_PAL   0x02    /* page aligned executable */
00044    #define A_NSYM  0x04    /* new style symbol table */
00045    #define A_EXEC  0x10    /* executable */
00046    #define A_SEP   0x20    /* separate I/D */
00047    #define A_PURE  0x40    /* pure text */         /* not used */
00048    #define A_TOVLY 0x80    /* text overlay */      /* not used */
00049
00050    /* Offsets of various things. */
00051    #define A_MINHDR        32
00052    #define A_TEXTPOS(X)    ((long)(X).a_hdrlen)
00053    #define A_DATAPOS(X)    (A_TEXTPOS(X) + (X).a_text)
00054    #define A_HASRELS(X)    ((X).a_hdrlen > (unsigned char) A_MINHDR)
```

```
00055   #define A_HASEXT(X)     ((X).a_hdrlen > (unsigned char) (A_MINHDR +  8))
00056   #define A_HASLNS(X)     ((X).a_hdrlen > (unsigned char) (A_MINHDR + 16))
00057   #define A_HASTOFF(X)    ((X).a_hdrlen > (unsigned char) (A_MINHDR + 24))
00058   #define A_TRELPOS(X)    (A_DATAPOS(X) + (X).a_data)
00059   #define A_DRELPOS(X)    (A_TRELPOS(X) + (X).a_trsize)
00060   #define A_SYMPOS(X)     (A_TRELPOS(X) + (A_HASRELS(X) ? \
00061                           ((X).a_trsize + (X).a_drsize) : 0))
00062
00063   struct reloc {
00064     long r_vaddr;                 /* virtual address of reference */
00065     unsigned short r_symndx;      /* internal segnum or extern symbol num */
00066     unsigned short r_type;        /* relocation type */
00067   };
00068
00069   /* r_tyep values: */
00070   #define R_ABBS          0
00071   #define R_RELLBYTE      2
00072   #define R_PCRBYTE       3
00073   #define R_RELWORD       4
00074   #define R_PCRWORD       5
00075   #define R_RELLONG       6
00076   #define R_PCRLONG       7
00077   #define R_REL3BYTE      8
00078   #define R_KBRANCHE      9
00079
00080   /* r_symndx for internal segments */
00081   #define S_ABS           ((unsigned short)-1)
00082   #define S_TEXT          ((unsigned short)-2)
00083   #define S_DATA          ((unsigned short)-3)
00084   #define S_BSS           ((unsigned short)-4)
00085
00086   struct nlist {                  /* symbol table entry */
00087     char n_name[8];               /* symbol name */
00088     long n_value;                 /* value */
00089     unsigned char n_sclass;       /* storage class */
00090     unsigned char n_numaux;       /* number of auxiliary entries (not used) */
00091     unsigned short n_type;        /* language base and derived type (not used) */
00092   };
00093
00094   /* Low bits of storage class (section). */
00095   #define N_SECT            07    /* section mask */
00096   #define N_UNDF            00    /* undefined */
00097   #define N_ABS             01    /* absolute */
00098   #define N_TEXT            02    /* text */
00099   #define N_DATA            03    /* data */
00100   #define N_BSS             04    /* bss */
00101   #define N_COMM            05    /* (common) */
00102
00103   /* High bits of storage class. */
00104   #define N_CLASS         0370    /* storage class mask */
00105   #define C_NULL
00106   #define C_EXT           0020    /* external symbol */
00107   #define C_STAT          0030    /* static */
00108
00109   /* Function prototypes. */
00110   #ifndef _ANSI_H
00111   #include <ansi.h>
00112   #endif
00113
00114   _PROTOTYPE( int nlist, (char *_file, struct nlist *_nl)                );
```

```
00115
00116    #endif /* _AOUT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/alloca.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

00200    /*      alloca.h - The dreaded alloca() function.
00201     */
00202
00203    #ifndef _ALLOCA_H
00204    #define _ALLOCA_H
00205
00206    #if __GNUC__
00207
00208    /* The compiler recognizes this special keyword, and inlines the code. */
00209    #define alloca(size)    __builtin_alloca(size)
00210
00211    #endif /* __GCC__ */
00212
00213    #if __ACK__ || __CCC__
00214
00215    #ifndef _ANSI_H
00216    #include <ansi.h>
00217    #endif
00218
00219    _PROTOTYPE(void *alloca, (size_t _size)                                 );
00220
00221    #endif /* __ACK__ || __CCC__ */
00222
00223    #endif /* _ALLOCA_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/ansi.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

00300    /* The <ansi.h> header attempts to decide whether the compiler has enough
00301     * conformance to Standard C for Minix to take advantage of.  If so, the
00302     * symbol _ANSI is defined (as 31415).  Otherwise _ANSI is not defined
00303     * here, but it may be defined by applications that want to bend the rules.
00304     * The magic number in the definition is to inhibit unnecessary bending
00305     * of the rules.  (For consistency with the new '#ifdef _ANSI" tests in
00306     * the headers, _ANSI should really be defined as nothing, but that would
00307     * break many library routines that use "#if _ANSI".)
00308
00309     * If _ANSI ends up being defined, a macro
00310     *
00311     *      _PROTOTYPE(function, params)
00312     *
00313     * is defined.  This macro expands in different ways, generating either
00314     * ANSI Standard C prototypes or old-style K&R (Kernighan & Ritchie)
00315     * prototypes, as needed.  Finally, some programs use _CONST, _VOIDSTAR etc
00316     * in such a way that they are portable over both ANSI and K&R compilers.
00317     * The appropriate macros are defined here.
00318     */
00319
```

```
00320    #ifndef _ANSI_H
00321    #define _ANSI_H
00322
00323    #if __STDC__ == 1
00324    #define _ANSI           31459   /* compiler claims full ANSI conformance */
00325    #endif
00326
00327    #ifdef __GNUC__
00328    #define _ANSI           31459   /* gcc conforms enough even in non-ANSI mode */
00329    #endif
00330
00331    #ifdef _ANSI
00332
00333    /* Keep everything for ANSI prototypes. */
00334    #define _PROTOTYPE(function, params)    function params
00335    #define _ARGS(params)                   params
00336
00337    #define _VOIDSTAR       void *
00338    #define _VOID           void
00339    #define _CONST          const
00340    #define _VOLATILE       volatile
00341    #define _SIZET          size_t
00342
00343    #else
00344
00345    /* Throw away the parameters for K&R prototypes. */
00346    #define _PROTOTYPE(function, params)    function()
00347    #define _ARGS(params)                   ()
00348
00349    #define _VOIDSTAR       void *
00350    #define _VOID           void
00351    #define _CONST
00352    #define _VOLATILE
00353    #define _SIZET          int
00354
00355    #endif /* _ANSI */
00356
00357    #endif /* ANSI_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/assert.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

00400    /* The <assert.h> header contains a macro called "assert" that allows
00401     * programmers to put assertions in the code.  These assertions can be verified
00402     * at run time.  If an assertion fails, an error message is printed and the
00403     * program aborts.
00404     * Assertion checking can be disabled by adding the statement
00405     *
00406     *      #define NDEBUG
00407     *
00408     * to the program before the
00409     *
00410     *      #include <assert.h>
00411     *
00412     * statement.
00413     */
00414
```

```
00415    #undef assert
00416
00417    #ifndef _ANSI_H
00418    #include <ansi.h>
00419    #endif
00420
00421
00422    #ifdef NDEBUG
00423    /* Debugging disabled -- do not evaluate assertions. */
00424    #define assert(expr)  ((void) 0)
00425    #else
00426    /* Debugging enabled -- verify assertions at run time. */
00427    #ifdef _ANSI
00428    #define __str(x)        # x
00429    #define __xstr(x)       __str(x)
00430
00431    _PROTOTYPE( void __bad_assertion, (const char *_mess) );
00432    #define assert(expr)    ((expr)? (void)0 : \
00433                                 __bad_assertion("Assertion \"" #expr \
00434                                     "\" failed, file " __xstr(__FILE__) \
00435                                     ", line " __xstr(__LINE__) "\n"))
00436    #else
00437    #define assert(expr) ((void) ((expr) ? 0 : __assert( __FILE__,  __LINE__)))
00438    #endif /* _ANSI */
00439    #endif
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/ctype.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

00500    /* The <ctype.h> header file defines some macros used to identify characters.
00501     * It works by using a table stored in chartab.c. When a character is presented
00502     * to one of these macros, the character is used as an index into the table
00503     * (__ctype) to retrieve a byte.  The relevant bit is then extracted.
00504     */
00505
00506    #ifndef _CTYPE_H
00507    #define _CTYPE_H
00508
00509    extern char     __ctype[];      /* property array defined in chartab.c */
00510
00511    #define _U              0x01    /* this bit is for upper-case letters [A-Z] */
00512    #define _L              0x02    /* this bit is for lower-case letters [a-z] */
00513    #define _N              0x04    /* this bit is for numbers [0-9] */
00514    #define _S              0x08    /* this bit is for white space \t \n \f etc */
00515    #define _P              0x10    /* this bit is for punctuation characters */
00516    #define _C              0x20    /* this bit is for control characters */
00517    #define _X              0x40    /* this bit is for hex digits [a-f] and [A-F]*/
00518
00519    /* Function Prototypes (have to go before the macros). */
00520    #ifndef _ANSI_H
00521    #include <ansi.h>
00522    #endif
00523
00524    _PROTOTYPE( int isalnum, (int  _c)  );  /* alphanumeric [a-z], [A-Z], [0-9] */
```

```
00525    _PROTOTYPE( int isalpha, (int  _c)  );  /* alphabetic */
00526    _PROTOTYPE( int iscntrl, (int  _c)  );  /* control characters */
00527    _PROTOTYPE( int isdigit, (int  _c)  );  /* digit [0-9] */
00528    _PROTOTYPE( int isgraph, (int  _c)  );  /* graphic character */
00529    _PROTOTYPE( int islower, (int  _c)  );  /* lower-case letter [a-z] */
00530    _PROTOTYPE( int isprint, (int  _c)  );  /* printable character */
00531    _PROTOTYPE( int ispunct, (int  _c)  );  /* punctuation mark */
00532    _PROTOTYPE( int isspace, (int  _c)  );  /* white space sp, \f, \n, \r, \t, \v*/
00533    _PROTOTYPE( int isupper, (int  _c)  );  /* upper-case letter [A-Z] */
00534    _PROTOTYPE( int isxdigit,(int  _c)  );  /* hex digit [0-9], [a-f], [A-F] */
00535    _PROTOTYPE( int tolower, (int  _c)  );  /* convert to lower-case */
00536    _PROTOTYPE( int toupper, (int  _c)  );  /* convert to upper-case */
00537
00538    /* Macros for identifying character classes. */
00539    #define isalnum(c)      ((__ctype+1)[c]&(_U|_L|_N))
00540    #define isalpha(c)      ((__ctype+1)[c]&(_U|_L))
00541    #define iscntrl(c)      ((__ctype+1)[c]&_C)
00542    #define isgraph(c)      ((__ctype+1)[c]&(_P|_U|_L|_N))
00543    #define ispunct(c)      ((__ctype+1)[c]&_P)
00544    #define isspace(c)      ((__ctype+1)[c]&_S)
00545    #define isxdigit(c)     ((__ctype+1)[c]&(_N|_X))
00546
00547    #define isdigit(c)      ((unsigned) ((c)-'0') < 10)
00548    #define islower(c)      ((unsigned) ((c)-'a') < 26)
00549    #define isupper(c)      ((unsigned) ((c)-'A') < 26)
00550    #define isprint(c)      ((unsigned) ((c)-' ') < 95)
00551    #define isascii(c)      ((unsigned) (c) < 128)
00552
00553    #endif /* _CTYPE_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/curses.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

00600    /* curses.h - defines macros and prototypes for curses */
00601
00602    #ifndef _CURSES_H
00603    #define _CURSES_H
00604
00605    #include <termios.h>
00606    #include <stdarg.h>
00607    #include <stdio.h>
00608
00609    typedef int bool;
00610
00611    #ifndef TRUE
00612    #define TRUE 1
00613    #endif
00614    #ifndef FALSE
00615    #define FALSE 0
00616    #endif
00617    #ifndef ERR
00618    #define ERR (-1)        /* general error flag */
00619    #endif
00620    #ifndef OK
00621    #define OK 0            /* general OK flag */
00622    #endif
00623
00624    /* Macros. */
```

```
00625    #define box(win,vc,hc) wbox(win,0,0,0,0,vc,hc)
00626    #define addch(ch) waddch(stdscr,ch)
00627    #define mvaddch(y,x,ch) (wmove(stdscr,y,x)==ERR?ERR:waddch(stdscr,ch))
00628    #define mvwaddch(win,y,x,ch) (wmove(win,y,x)==ERR?ERR:waddch(win,ch))
00629    #define getch() wgetch(stdscr)
00630    #define mvgetch(y,x) (wmove(stdscr,y,x)==ERR?ERR:wgetch(stdscr))
00631    #define mvwgetch(win,y,x) (wmove(win,y,x)==ERR?ERR:wgetch(win))
00632    #define addstr(str) waddstr(stdscr,str)
00633    #define mvaddstr(y,x,str) (wmove(stdscr,y,x)==ERR?ERR:waddstr(stdscr,str))
00634    #define mvwaddstr(win,y,x,str) (wmove(win,y,x)==ERR?ERR:waddstr(win,str))
00635    #define getstr(str) wgetstr(stdscr,str)
00636    #define mvgetstr(y,x,str) (wmove(stdscr,y,x)==ERR?ERR:wgetstr(stdscr,str))
00637    #define mvwgetstr(win,y,x,str) (wmove(win,y,x)==ERR?ERR:wgetstr(win,str))
00638    #define move(y,x) wmove(stdscr,y,x)
00639    #define clear() wclear(stdscr)
00640    #define erase() werase(stdscr)
00641    #define clrtobot() wclrtobot(stdscr)
00642    #define mvclrtobot(y,x) (wmove(stdscr,y,x)==ERR?ERR:wclrtobot(stdscr))
00643    #define mvwclrtobot(win,y,x) (wmove(win,y,x)==ERR?ERR:wclrtobot(win))
00644    #define clrtoeol() wclrtoeol(stdscr)
00645    #define mvclrtoeol(y,x) (wmove(stdscr,y,x)==ERR?ERR:wclrtoeol(stdscr))
00646    #define mvwclrtoeol(win,y,x) (wmove(win,y,x)==ERR?ERR:wclrtoeol(win))
00647    #define insertln() winsertln(stdscr)
00648    #define mvinsertln(y,x) (wmove(stdscr,y,x)==ERR?ERR:winsertln(stdscr))
00649    #define mvwinsertln(win,y,x) (wmove(win,y,x)==ERR?ERR:winsertln(win))
00650    #define deleteln() wdeleteln(stdscr)
00651    #define mvdeleteln(y,x) (wmove(stdscr,y,x)==ERR?ERR:wdeleteln(stdscr))
00652    #define mvwdeleteln(win,y,x) (wmove(win,y,x)==ERR?ERR:wdeleteln(win))
00653    #define refresh() wrefresh(stdscr)
00654    #define inch() winch(stdscr)
00655    #define insch(ch) winsch(stdscr,ch)
00656    #define mvinsch(y,x,ch) (wmove(stdscr,y,x)==ERR?ERR:winsch(stdscr,ch))
00657    #define mvwinsch(win,y,x,ch) (wmove(win,y,x)==ERR?ERR:winsch(win,ch))
00658    #define delch() wdelch(stdscr)
00659    #define mvdelch(y,x) (wmove(stdscr,y,x)==ERR?ERR:wdelch(stdscr))
00660    #define mvwdelch(win,y,x) (wmove(win,y,x)==ERR?ERR:wdelch(win))
00661    #define standout() wstandout(stdscr)
00662    #define wstandout(win) ((win)->_attrs |= A_STANDOUT)
00663    #define standend() wstandend(stdscr)
00664    #define wstandend(win) ((win)->_attrs &= ~A_STANDOUT)
00665    #define attrset(attrs) wattrset(stdscr, attrs)
00666    #define wattrset(win, attrs) ((win)->_attrs = (attrs))
00667    #define attron(attrs) wattron(stdscr, attrs)
00668    #define wattron(win, attrs) ((win)->_attrs |= (attrs))
00669    #define attroff(attrs) wattroff(stdscr,attrs)
00670    #define wattroff(win, attrs) ((win)->_attrs &= ~(attrs))
00671    #define resetty() tcsetattr(1, TCSANOW, &_orig_tty)
00672    #define getyx(win,y,x) (y = (win)->_cury, x = (win)->_curx)
00673
00674    /* Video attribute definitions. */
00675    #define A_BLINK        0x0100
00676    #define A_BLANK        0
00677    #define A_BOLD         0x0200
00678    #define A_DIM          0
00679    #define A_PROTECT      0
00680    #define A_REVERSE      0x0400
00681    #define A_STANDOUT     0x0800
00682    #define A_UNDERLINE    0x1000
00683    #define A_ALTCHARSET   0x2000
00684
```

```
00685   /* Type declarations. */
00686   typedef struct {
00687     int      _cury;                     /* current pseudo-cursor */
00688     int      _curx;
00689     int      _maxy;                     /* max coordinates */
00690     int      _maxx;
00691     int      _begy;                     /* origin on screen */
00692     int      _begx;
00693     int      _flags;                    /* window properties */
00694     int      _attrs;                    /* attributes of written characters */
00695     int      _tabsize;                  /* tab character size */
00696     bool     _clear;                    /* causes clear at next refresh */
00697     bool     _leave;                    /* leaves cursor as it happens */
00698     bool     _scroll;                   /* allows window scrolling */
00699     bool     _nodelay;                  /* input character wait flag */
00700     bool     _keypad;                   /* flags keypad key mode active */
00701     int    **_line;                     /* pointer to line pointer array */
00702     int     *_minchng;                  /* First changed character in line */
00703     int     *_maxchng;                  /* Last changed character in line */
00704     int      _regtop;                   /* Top/bottom of scrolling region */
00705     int      _regbottom;
00706   } WINDOW;
00707
00708   /* External variables */
00709   extern int LINES;                     /* terminal height */
00710   extern int COLS;                      /* terminal width */
00711   extern bool NONL;                     /* \n causes CR too ? */
00712   extern WINDOW *curscr;                /* the current screen image */
00713   extern WINDOW *stdscr;                /* the default screen window */
00714   extern struct termios _orig_tty, _tty;
00715
00716   extern unsigned int ACS_ULCORNER;     /* terminal dependent block grafic */
00717   extern unsigned int ACS_LLCORNER;     /* charcters.  Forget IBM, we are */
00718   extern unsigned int ACS_URCORNER;     /* independent of their charset. :-) */
00719   extern unsigned int ACS_LRCORNER;
00720   extern unsigned int ACS_RTEE;
00721   extern unsigned int ACS_LTEE;
00722   extern unsigned int ACS_BTEE;
00723   extern unsigned int ACS_TTEE;
00724   extern unsigned int ACS_HLINE;
00725   extern unsigned int ACS_VLINE;
00726   extern unsigned int ACS_PLUS;
00727   extern unsigned int ACS_S1;
00728   extern unsigned int ACS_S9;
00729   extern unsigned int ACS_DIAMOND;
00730   extern unsigned int ACS_CKBOARD;
00731   extern unsigned int ACS_DEGREE;
00732   extern unsigned int ACS_PLMINUS;
00733   extern unsigned int ACS_BULLET;
00734   extern unsigned int ACS_LARROW;
00735   extern unsigned int ACS_RARROW;
00736   extern unsigned int ACS_DARROW;
00737   extern unsigned int ACS_UARROW;
00738   extern unsigned int ACS_BOARD;
00739   extern unsigned int ACS_LANTERN;
00740   extern unsigned int ACS_BLOCK;
00741
00742   _PROTOTYPE( char *unctrl, (int _c) );
00743   _PROTOTYPE( int baudrate, (void));
00744   _PROTOTYPE( void beep, (void));
```

```
00745    _PROTOTYPE( void cbreak, (void));
00746    _PROTOTYPE( void clearok, (WINDOW *_win, bool _flag) );
00747    _PROTOTYPE( void clrscr, (void));
00748    _PROTOTYPE( void curs_set, (int _visibility) );
00749    _PROTOTYPE( void delwin, (WINDOW *_win) );
00750    _PROTOTYPE( void doupdate, (void));
00751    _PROTOTYPE( void echo, (void));
00752    _PROTOTYPE( int endwin, (void));
00753    _PROTOTYPE( int erasechar, (void));
00754    _PROTOTYPE( void fatal, (char *_s) );
00755    _PROTOTYPE( int fixterm, (void));
00756    _PROTOTYPE( void flash, (void));
00757    _PROTOTYPE( void gettmode, (void));
00758    _PROTOTYPE( void idlok, (WINDOW *_win, bool _flag) );
00759    _PROTOTYPE( WINDOW *initscr, (void));
00760    _PROTOTYPE( void keypad, (WINDOW *_win, bool _flag) );
00761    _PROTOTYPE( int killchar, (void));
00762    _PROTOTYPE( void leaveok, (WINDOW *_win, bool _flag) );
00763    _PROTOTYPE( char *longname, (void));
00764    _PROTOTYPE( void meta, (WINDOW *_win, bool _flag) );
00765    _PROTOTYPE( int mvcur, (int _oldy, int _oldx, int _newy, int _newx) );
00766    _PROTOTYPE( int mvinch, (int _y, int _x) );
00767    _PROTOTYPE( int mvprintw, (int _y, int _x, const char *_fmt, ...) );
00768    _PROTOTYPE( int mvscanw, (int _y, int _x, const char *_fmt, ...) );
00769    _PROTOTYPE( int mvwin, (WINDOW *_win, int _begy, int _begx) );
00770    _PROTOTYPE( int mvwinch, (WINDOW *_win, int _y, int _x) );
00771    _PROTOTYPE( int mvwprintw, (WINDOW *_win, int _y, int _x, const char *_fmt,
00772                                                                    ...) );
00773    _PROTOTYPE( int mvwscanw, (WINDOW *_win, int _y, int _x, const char *_fmt,
00774                                                                    ...) );
00775    _PROTOTYPE( WINDOW *newwin, (int _num_lines, int _num_cols, int _y, int _x));
00776    _PROTOTYPE( void nl, (void));
00777    _PROTOTYPE( void nocbreak, (void));
00778    _PROTOTYPE( void nodelay, (WINDOW *_win, bool _flag) );
00779    _PROTOTYPE( void noecho, (void));
00780    _PROTOTYPE( void nonl, (void));
00781    _PROTOTYPE( void noraw, (void));
00782    _PROTOTYPE( void outc, (int _c) );
00783    _PROTOTYPE( void  overlay, (WINDOW *_win1, WINDOW *_win2) );
00784    _PROTOTYPE( void  overwrite, (WINDOW *_win1, WINDOW *_win2) );
00785    _PROTOTYPE( void poscur, (int _r, int _c) );
00786    _PROTOTYPE( int printw, (const char *_fmt, ...) );
00787    _PROTOTYPE( void raw, (void));
00788    _PROTOTYPE( int resetterm, (void));
00789    _PROTOTYPE( int saveoldterm, (void));
00790    _PROTOTYPE( int saveterm, (void));
00791    _PROTOTYPE( int savetty, (void));
00792    _PROTOTYPE( int scanw, (const char *_fmt, ...) );
00793    _PROTOTYPE( void scroll, (WINDOW *_win) );
00794    _PROTOTYPE( void scrollok, (WINDOW *_win, bool _flag) );
00795    _PROTOTYPE( int setscrreg, (int _top, int _bottom) );
00796    _PROTOTYPE( int setterm, (char *_type) );
00797    _PROTOTYPE( int setupterm, (void));
00798    _PROTOTYPE( WINDOW *subwin, (WINDOW *_orig, int _nlines, int _ncols, int _y,
00799                                          int _x));
00800    _PROTOTYPE( int tabsize, (int _ts) );
00801    _PROTOTYPE( void touchwin, (WINDOW *_win) );
00802    _PROTOTYPE( int waddch, (WINDOW *_win, int _c) );
00803    _PROTOTYPE( int waddstr, (WINDOW *_win, char *_str) );
00804    _PROTOTYPE( int wbox, (WINDOW *_win, int _ymin, int _xmin, int _ymax,
```

```
00805                                   int _xmax, unsigned int _v, unsigned int _h) );
00806    _PROTOTYPE( void wclear, (WINDOW *_win) );
00807    _PROTOTYPE( int wclrtobot, (WINDOW *_win) );
00808    _PROTOTYPE( int wclrtoeol, (WINDOW *_win) );
00809    _PROTOTYPE( int wdelch, (WINDOW *_win) );
00810    _PROTOTYPE( int wdeleteln, (WINDOW *_win) );
00811    _PROTOTYPE( void werase, (WINDOW *_win) );
00812    _PROTOTYPE( int wgetch, (WINDOW *_win) );
00813    _PROTOTYPE( int wgetstr, (WINDOW *_win, char *_str) );
00814    _PROTOTYPE( int winch, (WINDOW *_win) );
00815    _PROTOTYPE( int winsch, (WINDOW *_win, int _c) );
00816    _PROTOTYPE( int winsertln, (WINDOW *_win) );
00817    _PROTOTYPE( int wmove, (WINDOW *_win, int _y, int _x) );
00818    _PROTOTYPE( void wnoutrefresh, (WINDOW *_win) );
00819    _PROTOTYPE( int wprintw, (WINDOW *_win, const char *_fmt, ...));
00820    _PROTOTYPE( void wrefresh, (WINDOW *_win) );
00821    _PROTOTYPE( int wscanw, (WINDOW *_win, const char *_fmt, ...));
00822    _PROTOTYPE( int wsetscrreg, (WINDOW *_win, int _top, int _bottom) );
00823    _PROTOTYPE( int wtabsize, (WINDOW *_win, int _ts) );
00824
00825    #endif /* _CURSES_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/dirent.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

00900    /*      dirent.h - Declarations for directory reading routines.
00901     *                                                      Author: Kees J. Bot
00902     *                                                              24 Apr 1989
00903     *
00904     * Note: The V7 format directory entries used under Minix must be transformed
00905     * into a struct dirent with a d_name of at least 15 characters.  Given that
00906     * we have to transform V7 entries anyhow it is little trouble to let the
00907     * routines understand the so-called "flex" directory format too.
00908     */
00909
00910    #ifndef _DIRENT_H
00911    #define _DIRENT_H
00912
00913    /* _fl_direct is a flexible directory entry.  Actually it's a union of 8
00914     * characters and the 3 fields defined below.
00915     */
00916
00917    /* Flexible directory entry: */
00918    struct _fl_direct {             /* First slot in an entry */
00919            ino_t           d_ino;
00920            unsigned char   d_extent;
00921            char            d_name[5];  /* Four characters for the shortest name */
00922    };
00923
00924            /* Name of length len needs _EXTENT(len) extra slots. */
00925    #define _EXTENT(len)    (((len) + 3) >> 3)
00926
00927    /* Version 7 directory entry: */
00928    struct _v7_direct {
00929            ino_t           d_ino;
```

```
00930           char            d_name[14];
00931   };
00932
00933   /* Definitions for the directory(3) routines: */
00934   typedef struct {
00935           char            _fd;    /* Filedescriptor of open directory */
00936           char            _v7;    /* Directory is Version 7 */
00937           short           _count; /* This many objects in buf */
00938           off_t           _pos;   /* Position in directory file */
00939           struct _fl_direct  *_ptr;       /* Next slot in buf */
00940           struct _fl_direct  _buf[128];   /* One block of a directory file */
00941           struct _fl_direct  _v7f[3];     /* V7 entry transformed to flex */
00942   } DIR;
00943
00944   struct dirent {         /* Largest entry (8 slots) */
00945           ino_t           d_ino;          /* I-node number */
00946           unsigned char   d_extent;       /* Extended with this many slots */
00947           char            d_name[61];     /* Null terminated name */
00948   };
00949
00950   /* Function Prototypes. */
00951   #ifndef _ANSI_H
00952   #include <ansi.h>
00953   #endif
00954
00955   _PROTOTYPE( int closedir, (DIR *_dirp)                                  );
00956   _PROTOTYPE( DIR *opendir, (const char *_dirname)                        );
00957   _PROTOTYPE( struct dirent *readdir, (DIR *_dirp)                        );
00958   _PROTOTYPE( void rewinddir, (DIR *_dirp)                                );
00959
00960   #ifdef _MINIX
00961   _PROTOTYPE( int seekdir, (DIR *_dirp, off_t _loc)                       );
00962   _PROTOTYPE( off_t telldir, (DIR *_dirp)                                 );
00963   #endif
00964
00965   #endif /* _DIRENT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/errno.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

01000   /* The <errno.h> header defines the numbers of the various errors that can
01001    * occur during program execution.  They are visible to user programs and
01002    * should be small positive integers.  However, they are also used within
01003    * MINIX, where they must be negative.  For example, the READ system call is
01004    * executed internally by calling do_read().  This function returns either a
01005    * (negative) error number or a (positive) number of bytes actually read.
01006    *
01007    * To solve the problem of having the error numbers be negative inside the
01008    * the system and positive outside, the following mechanism is used.  All the
01009    * definitions are are the form:
01010    *
01011    *      #define EPERM           (_SIGN 1)
01012    *
01013    * If the macro _SYSTEM is defined, then  _SIGN is set to "-", otherwise it is
01014    * set to "".  Thus when compiling the operating system, the  macro _SYSTEM
```

```
01015     * will be defined, setting EPERM to (- 1), whereas when when this
01016     * file is included in an ordinary user program, EPERM has the value ( 1).
01017     */
01018
01019    #ifndef _ERRNO_H                /* check if <errno.h> is already included */
01020    #define _ERRNO_H                /* it is not included; note that fact */
01021
01022    /* Now define _SIGN as "" or "-" depending on _SYSTEM. */
01023    #ifdef _SYSTEM
01024    #   define _SIGN         -
01025    #   define OK            0
01026    #else
01027    #   define _SIGN
01028    #endif
01029
01030    extern int errno;                /* place where the error numbers go */
01031
01032    /* Here are the numerical values of the error numbers. */
01033    #define _NERROR               70  /* number of errors */
01034
01035    #define EGENERIC      (_SIGN 99)  /* generic error */
01036    #define EPERM         (_SIGN  1)  /* operation not permitted */
01037    #define ENOENT        (_SIGN  2)  /* no such file or directory */
01038    #define ESRCH         (_SIGN  3)  /* no such process */
01039    #define EINTR         (_SIGN  4)  /* interrupted function call */
01040    #define EIO           (_SIGN  5)  /* input/output error */
01041    #define ENXIO         (_SIGN  6)  /* no such device or address */
01042    #define E2BIG         (_SIGN  7)  /* arg list too long */
01043    #define ENOEXEC       (_SIGN  8)  /* exec format error */
01044    #define EBADF         (_SIGN  9)  /* bad file descriptor */
01045    #define ECHILD        (_SIGN 10)  /* no child process */
01046    #define EAGAIN        (_SIGN 11)  /* resource temporarily unavailable */
01047    #define ENOMEM        (_SIGN 12)  /* not enough space */
01048    #define EACCES        (_SIGN 13)  /* permission denied */
01049    #define EFAULT        (_SIGN 14)  /* bad address */
01050    #define ENOTBLK       (_SIGN 15)  /* Extension: not a block special file */
01051    #define EBUSY         (_SIGN 16)  /* resource busy */
01052    #define EEXIST        (_SIGN 17)  /* file exists */
01053    #define EXDEV         (_SIGN 18)  /* improper link */
01054    #define ENODEV        (_SIGN 19)  /* no such device */
01055    #define ENOTDIR       (_SIGN 20)  /* not a directory */
01056    #define EISDIR        (_SIGN 21)  /* is a directory */
01057    #define EINVAL        (_SIGN 22)  /* invalid argument */
01058    #define ENFILE        (_SIGN 23)  /* too many open files in system */
01059    #define EMFILE        (_SIGN 24)  /* too many open files */
01060    #define ENOTTY        (_SIGN 25)  /* inappropriate I/O control operation */
01061    #define ETXTBSY       (_SIGN 26)  /* no longer used */
01062    #define EFBIG         (_SIGN 27)  /* file too large */
01063    #define ENOSPC        (_SIGN 28)  /* no space left on device */
01064    #define ESPIPE        (_SIGN 29)  /* invalid seek */
01065    #define EROFS         (_SIGN 30)  /* read-only file system */
01066    #define EMLINK        (_SIGN 31)  /* too many links */
01067    #define EPIPE         (_SIGN 32)  /* broken pipe */
01068    #define EDOM          (_SIGN 33)  /* domain error       (from ANSI C std) */
01069    #define ERANGE        (_SIGN 34)  /* result too large   (from ANSI C std) */
01070    #define EDEADLK       (_SIGN 35)  /* resource deadlock avoided */
01071    #define ENAMETOOLONG  (_SIGN 36)  /* file name too long */
01072    #define ENOLCK        (_SIGN 37)  /* no locks available */
01073    #define ENOSYS        (_SIGN 38)  /* function not implemented */
01074    #define ENOTEMPTY     (_SIGN 39)  /* directory not empty */
```

```
01075
01076    /* The following errors relate to networking. */
01077    #define EPACKSIZE      (_SIGN 50)  /* invalid packet size for some protocol */
01078    #define EOUTOFBUFS     (_SIGN 51)  /* not enough buffers left */
01079    #define EBADIOCTL      (_SIGN 52)  /* illegal ioctl for device */
01080    #define EBADMODE       (_SIGN 53)  /* badmode in ioctl */
01081    #define EWOULDBLOCK    (_SIGN 54)
01082    #define EBADDEST       (_SIGN 55)  /* not a valid destination address */
01083    #define EDSTNOTRCH     (_SIGN 56)  /* destination not reachable */
01084    #define EISCONN        (_SIGN 57)  /* all ready connected */
01085    #define EADDRINUSE     (_SIGN 58)  /* address in use */
01086    #define ECONNREFUSED   (_SIGN 59)  /* connection refused */
01087    #define ECONNRESET     (_SIGN 60)  /* connection reset */
01088    #define ETIMEDOUT      (_SIGN 61)  /* connection timed out */
01089    #define EURG           (_SIGN 62)  /* urgent data present */
01090    #define ENOURG         (_SIGN 63)  /* no urgent data present */
01091    #define ENOTCONN       (_SIGN 64)  /* no connection (yet or anymore) */
01092    #define ESHUTDOWN      (_SIGN 65)  /* a write call to a shutdown connection */
01093    #define ENOCONN        (_SIGN 66)  /* no such connection */
01094
01095    /* The following are not POSIX errors, but they can still happen. */
01096    #define ELOCKED       (_SIGN 101)  /* can't send message */
01097    #define EBADCALL      (_SIGN 102)  /* error on send/receive */
01098
01099    /* The following error codes are generated by the kernel itself. */
01100    #ifdef _SYSTEM
01101    #define E_BAD_DEST     -1001    /* destination address illegal */
01102    #define E_BAD_SRC      -1002    /* source address illegal */
01103    #define E_TRY_AGAIN    -1003    /* can't send-- tables full */
01104    #define E_OVERRUN      -1004    /* interrupt for task that is not waiting */
01105    #define E_BAD_BUF      -1005    /* message buf outside caller's addr space */
01106    #define E_TASK         -1006    /* can't send to task */
01107    #define E_NO_MESSAGE   -1007    /* RECEIVE failed: no message present */
01108    #define E_NO_PERM      -1008    /* ordinary users can't send to tasks */
01109    #define E_BAD_FCN      -1009    /* only valid fcns are SEND, RECEIVE, BOTH */
01110    #define E_BAD_ADDR     -1010    /* bad address given to utility routine */
01111    #define E_BAD_PROC     -1011    /* bad proc number given to utility */
01112    #endif /* _SYSTEM */
01113
01114    #endif /* _ERRNO_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/fcntl.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

01200    /* The <fcntl.h> header is needed by the open() and fcntl() system calls,
01201     * which  have a variety of parameters and flags.  They are described here.
01202     * The formats of the calls to each of these are:
01203     *
01204     *      open(path, oflag [,mode])       open a file
01205     *      fcntl(fd, cmd [,arg])           get or set file attributes
01206     *
01207     */
01208
01209    #ifndef _FCNTL_H
```

```
01210    #define _FCNTL_H
01211
01212    /* These values are used for cmd in fcntl().  POSIX Table 6-1.  */
01213    #define F_DUPFD            0    /* duplicate file descriptor */
01214    #define F_GETFD            1    /* get file descriptor flags */
01215    #define F_SETFD            2    /* set file descriptor flags */
01216    #define F_GETFL            3    /* get file status flags */
01217    #define F_SETFL            4    /* set file status flags */
01218    #define F_GETLK            5    /* get record locking information */
01219    #define F_SETLK            6    /* set record locking information */
01220    #define F_SETLKW           7    /* set record locking info; wait if blocked */
01221
01222    /* File descriptor flags used for fcntl().  POSIX Table 6-2. */
01223    #define FD_CLOEXEC         1    /* close on exec flag for third arg of fcntl */
01224
01225    /* L_type values for record locking with fcntl().  POSIX Table 6-3. */
01226    #define F_RDLCK            1    /* shared or read lock */
01227    #define F_WRLCK            2    /* exclusive or write lock */
01228    #define F_UNLCK            3    /* unlock */
01229
01230    /* Oflag values for open().  POSIX Table 6-4. */
01231    #define O_CREAT        00100    /* creat file if it doesn't exist */
01232    #define O_EXCL         00200    /* exclusive use flag */
01233    #define O_NOCTTY       00400    /* do not assign a controlling terminal */
01234    #define O_TRUNC        01000    /* truncate flag */
01235
01236    /* File status flags for open() and fcntl().  POSIX Table 6-5. */
01237    #define O_APPEND       02000    /* set append mode */
01238    #define O_NONBLOCK     04000    /* no delay */
01239
01240    /* File access modes for open() and fcntl().  POSIX Table 6-6. */
01241    #define O_RDONLY           0    /* open(name, O_RDONLY) opens read only */
01242    #define O_WRONLY           1    /* open(name, O_WRONLY) opens write only */
01243    #define O_RDWR             2    /* open(name, O_RDWR) opens read/write */
01244
01245    /* Mask for use with file access modes.  POSIX Table 6-7. */
01246    #define O_ACCMODE         03    /* mask for file access modes */
01247
01248    /* Struct used for locking.  POSIX Table 6-8. */
01249    struct flock {
01250      short l_type;                /* type: F_RDLCK, F_WRLCK, or F_UNLCK */
01251      short l_whence;              /* flag for starting offset */
01252      off_t l_start;               /* relative offset in bytes */
01253      off_t l_len;                 /* size; if 0, then until EOF */
01254      pid_t l_pid;                 /* process id of the locks' owner */
01255    };
01256
01257
01258    /* Function Prototypes. */
01259    #ifndef _ANSI_H
01260    #include <ansi.h>
01261    #endif
01262
01263    _PROTOTYPE( int creat, (const char *_path, Mode_t _mode)               );
01264    _PROTOTYPE( int fcntl, (int _filedes, int _cmd, ...)                   );
01265    _PROTOTYPE( int open,  (const char *_path, int _oflag, ...)            );
01266
01267    #endif /* _FCNTL_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/float.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

01300    /* The <float.h> header defines some implementation limits for (IEEE) floating
01301     * point. Application programs can use it to find out how big and small
01302     * floating-point numbers can be, what epsilon to use for iteration, etc.
01303     */
01304
01305    #ifndef _FLOAT_H
01306    #define _FLOAT_H
01307
01308    #define FLT_DIG                 6
01309    #define FLT_EPSILON             1.19209290e-07F
01310    #define FLT_MANT_DIG            24
01311    #define FLT_MAX                 3.40282347e+38F
01312    #define FLT_MAX_10_EXP          38
01313    #define FLT_MAX_EXP             128
01314    #define FLT_MIN                 1.17549435e-38F
01315    #define FLT_MIN_10_EXP          -37
01316    #define FLT_MIN_EXP             -125
01317
01318    #define DBL_DIG                 15
01319    #define DBL_EPSILON             2.2204460492503131e-16
01320    #define DBL_MANT_DIG            53
01321    #define DBL_MAX                 1.7976931348623157e+308
01322    #define DBL_MAX_10_EXP          308
01323    #define DBL_MAX_EXP             1024
01324    #define DBL_MIN                 2.2250738585072014e-308
01325    #define DBL_MIN_10_EXP          -307
01326    #define DBL_MIN_EXP             -1021
01327
01328    #define LDBL_DIG                15
01329    #define LDBL_EPSILON            2.2204460492503131e-16L
01330    #define LDBL_MANT_DIG           53
01331    #define LDBL_MAX                1.7976931348623157e+308L
01332    #define LDBL_MAX_10_EXP         308
01333    #define LDBL_MAX_EXP            1024
01334    #define LDBL_MIN                2.2250738585072014e-308L
01335    #define LDBL_MIN_10_EXP         -307
01336    #define LDBL_MIN_EXP            -1021
01337
01338    #define FLT_ROUNDS              1
01339    #define FLT_RADIX               2
01340
01341    #endif /* _FLOAT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/grp.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

01400    /* The <grp.h> header is used for the getgrid() and getgrnam() calls. */
01401
01402    #ifndef _GRP_H
01403    #define _GRP_H
01404
01405    struct  group {
01406      char *gr_name;                /* the name of the group */
01407      char *gr_passwd;              /* the group passwd */
01408      gid_t gr_gid;                 /* the numerical group ID */
01409      char **gr_mem;                /* a vector of pointers to the members */
01410    };
01411
01412    /* Function Prototypes. */
01413    #ifndef _ANSI_H
01414    #include <ansi.h>
01415    #endif
01416
01417    _PROTOTYPE( struct group *getgrgid, (Gid_t _gid)                        );
01418    _PROTOTYPE( struct group *getgrnam, (const char *_name)                 );
01419
01420    #ifdef _MINIX
01421    _PROTOTYPE( void endgrent, (void)                                       );
01422    _PROTOTYPE( struct group *getgrent, (void)                              );
01423    _PROTOTYPE( int setgrent, (void)                                        );
01424    _PROTOTYPE( void setgrfile, (const char *_file)                         );
01425    #endif
01426
01427    #endif /* _GRP_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/lib.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

01500    /* The <lib.h> header is the master header used by the library.
01501     * All the C files in the lib subdirectories include it.
01502     */
01503
01504    #ifndef _LIB_H
01505    #define _LIB_H
01506
01507    /* First come the defines. */
01508    #define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
01509    #define _MINIX             1    /* tell headers to include MINIX stuff */
01510
01511    /* The following are so basic, all the lib files get them automatically. */
01512    #include <minix/config.h>       /* must be first */
01513    #include <sys/types.h>
01514    #include <limits.h>
01515    #include <errno.h>
01516    #include <ansi.h>
01517
01518    #include <minix/const.h>
01519    #include <minix/type.h>
```

```
01520    #include <minix/callnr.h>
01521
01522    #define MM                 0
01523    #define FS                 1
01524
01525    _PROTOTYPE( int __execve, (const char *_path, char *const _argv[],
01526                            char *const _envp[], int _nargs, int _nenvps)   );
01527    _PROTOTYPE( int _syscall, (int _who, int _syscallnr, message *_msgptr)  );
01528    _PROTOTYPE( void _loadname, (const char *_name, message *_msgptr)       );
01529    _PROTOTYPE( int _len, (const char *_s)                                  );
01530    _PROTOTYPE( void panic, (const char *_message, int _errnum)             );
01531    _PROTOTYPE( int _sendrec, (int _src_dest, message *_m_ptr)              );
01532    _PROTOTYPE( void _begsig, (int _dummy)                                  );
01533
01534    #endif /* _LIB_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                          include/limits.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

01600    /* The <limits.h> header defines some basic sizes, both of the language types
01601     * (e.g., the number of bits in an integer), and of the operating system (e.g.
01602     * the number of characters in a file name.
01603     */
01604
01605    #ifndef _LIMITS_H
01606    #define _LIMITS_H
01607
01608    /* Definitions about chars (8 bits in MINIX, and signed). */
01609    #define CHAR_BIT           8    /* # bits in a char */
01610    #define CHAR_MIN        -128    /* minimum value of a char */
01611    #define CHAR_MAX         127    /* maximum value of a char */
01612    #define SCHAR_MIN       -128    /* minimum value of a signed char */
01613    #define SCHAR_MAX        127    /* maximum value of a signed char */
01614    #define UCHAR_MAX        255    /* maximum value of an unsigned char */
01615    #define MB_LEN_MAX         1    /* maximum length of a multibyte char */
01616
01617    /* Definitions about shorts (16 bits in MINIX). */
01618    #define SHRT_MIN  (-32767-1)    /* minimum value of a short */
01619    #define SHRT_MAX       32767    /* maximum value of a short */
01620    #define USHRT_MAX     0xFFFF    /* maximum value of unsigned short */
01621
01622    /* _EM_WSIZE is a compiler-generated symbol giving the word size in bytes. */
01623    #if _EM_WSIZE == 2
01624    #define INT_MIN   (-32767-1)    /* minimum value of a 16-bit int */
01625    #define INT_MAX        32767    /* maximum value of a 16-bit int */
01626    #define UINT_MAX      0xFFFF    /* maximum value of an unsigned 16-bit int */
01627    #endif
01628
01629    #if _EM_WSIZE == 4
01630    #define INT_MIN (-2147483647-1) /* minimum value of a 32-bit int */
01631    #define INT_MAX   2147483647    /* maximum value of a 32-bit int */
01632    #define UINT_MAX  0xFFFFFFFF    /* maximum value of an unsigned 32-bit int */
01633    #endif
01634
```

```
01635    /*Definitions about longs (32 bits in MINIX). */
01636    #define LONG_MIN (-2147483647L-1)/* minimum value of a long */
01637    #define LONG_MAX  2147483647L   /* maximum value of a long */
01638    #define ULONG_MAX 0xFFFFFFFFL   /* maximum value of an unsigned long */
01639
01640    /* Minimum sizes required by the POSIX P1003.1 standard (Table 2-3). */
01641    #ifdef _POSIX_SOURCE            /* these are only visible for POSIX */
01642    #define _POSIX_ARG_MAX    4096  /* exec() may have 4K worth of args */
01643    #define _POSIX_CHILD_MAX     6  /* a process may have 6 children */
01644    #define _POSIX_LINK_MAX      8  /* a file may have 8 links */
01645    #define _POSIX_MAX_CANON   255  /* size of the canonical input queue */
01646    #define _POSIX_MAX_INPUT   255  /* you can type 255 chars ahead */
01647    #define _POSIX_NAME_MAX     14  /* a file name may have 14 chars */
01648    #define _POSIX_NGROUPS_MAX   0  /* supplementary group IDs are optional */
01649    #define _POSIX_OPEN_MAX     16  /* a process may have 16 files open */
01650    #define _POSIX_PATH_MAX    255  /* a pathname may contain 255 chars */
01651    #define _POSIX_PIPE_BUF    512  /* pipes writes of 512 bytes must be atomic */
01652    #define _POSIX_STREAM_MAX    8  /* at least 8 FILEs can be open at once */
01653    #define _POSIX_TZNAME_MAX    3  /* time zone names can be at least 3 chars */
01654    #define _POSIX_SSIZE_MAX 32767  /* read() must support 32767 byte reads */
01655
01656    /* Values actually implemented by MINIX (Tables 2-4, 2-5, 2-6, and 2-7). */
01657    /* Some of these old names had better be defined when not POSIX. */
01658    #define _NO_LIMIT        100    /* arbitrary number; limit not enforced */
01659
01660    #define NGROUPS_MAX        0    /* supplemental group IDs not available */
01661    #if _EM_WSIZE > 2
01662    #define ARG_MAX        16384    /* # bytes of args + environ for exec() */
01663    #else
01664    #define ARG_MAX         4096    /* args + environ on small machines */
01665    #endif
01666    #define CHILD_MAX  _NO_LIMIT    /* MINIX does not limit children */
01667    #define OPEN_MAX          20    /* # open files a process may have */
01668    #define LINK_MAX         127    /* # links a file may have */
01669    #define MAX_CANON        255    /* size of the canonical input queue */
01670    #define MAX_INPUT        255    /* size of the type-ahead buffer */
01671    #define NAME_MAX          14    /* # chars in a file name */
01672    #define PATH_MAX         255    /* # chars in a path name */
01673    #define PIPE_BUF        7168    /* # bytes in atomic write to a pipe */
01674    #define STREAM_MAX        20    /* must be the same as FOPEN_MAX in stdio.h */
01675    #define TZNAME_MAX         3    /* maximum bytes in a time zone name is 3 */
01676    #define SSIZE_MAX      32767    /* max defined byte count for read() */
01677
01678    #endif /* _POSIX_SOURCE */
01679
01680    #endif /* _LIMITS_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/locale.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

01700    /* The <locale.h> header is used to custom tailor currency symbols, decimal
01701     * points, and other items to the local style.  It is ANSI's attempt at
01702     * avoiding cultural imperialism.  The locale given below is for C.
01703     */
01704
01705    #ifndef _LOCALE_H
01706    #define _LOCALE_H
01707
01708    struct lconv {
01709      char *decimal_point;          /* "." */
01710      char *thousands_sep;          /* ""  */
01711      char *grouping;               /* ""  */
01712      char *int_curr_symbol;        /* ""  */
01713      char *currency_symbol;        /* ""  */
01714      char *mon_decimal_point;      /* ""  */
01715      char *mon_thousands_sep;      /* ""  */
01716      char *mon_grouping;           /* ""  */
01717      char *positive_sign;          /* ""  */
01718      char *negative_sign;          /* ""  */
01719      char int_frac_digits;         /* CHAR_MAX */
01720      char frac_digits;             /* CHAR_MAX */
01721      char p_cs_precedes;           /* CHAR_MAX */
01722      char p_sep_by_space;          /* CHAR_MAX */
01723      char n_cs_precedes;           /* CHAR_MAX */
01724      char n_sep_by_space;          /* CHAR_MAX */
01725      char p_sign_posn;             /* CHAR_MAX */
01726      char n_sign_posn;             /* CHAR_MAX */
01727    };
01728
01729    #define NULL    ((void *)0)
01730
01731    #define LC_ALL             1
01732    #define LC_COLLATE         2
01733    #define LC_CTYPE           3
01734    #define LC_MONETARY        4
01735    #define LC_NUMERIC         5
01736    #define LC_TIME            6
01737
01738    /* Function Prototypes. */
01739    #ifndef _ANSI_H
01740    #include <ansi.h>
01741    #endif
01742
01743    _PROTOTYPE( char *setlocale, (int _category, const char *_locale)       );
01744    _PROTOTYPE( struct lconv *localeconv, (void)                            );
01745
01746    #endif /* _LOCALE_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/math.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

01800    /* The <math.h> header contains prototypes for mathematical functions. */
01801
01802    #ifndef _MATH_H
01803    #define _MATH_H
01804
01805    #define HUGE_VAL        (__huge_val())          /* may be infinity */
01806
01807    /* Function Prototypes. */
01808    #ifndef _ANSI_H
01809    #include <ansi.h>
01810    #endif
01811
01812    _PROTOTYPE( double __huge_val,  (void)                                  );
01813    _PROTOTYPE( int __IsNan,        (double _x)                             );
01814
01815    _PROTOTYPE( double acos,  (double _x)                                   );
01816    _PROTOTYPE( double asin,  (double _x)                                   );
01817    _PROTOTYPE( double atan,  (double _x)                                   );
01818    _PROTOTYPE( double atan2, (double _y, double _x)                        );
01819    _PROTOTYPE( double ceil,  (double _x)                                   );
01820    _PROTOTYPE( double cos,   (double _x)                                   );
01821    _PROTOTYPE( double cosh,  (double _x)                                   );
01822    _PROTOTYPE( double exp,   (double _x)                                   );
01823    _PROTOTYPE( double fabs,  (double _x)                                   );
01824    _PROTOTYPE( double floor, (double _x)                                   );
01825    _PROTOTYPE( double fmod,  (double _x, double _y)                        );
01826    _PROTOTYPE( double frexp, (double _x, int *_exp)                        );
01827    _PROTOTYPE( double ldexp, (double _x, int _exp)                         );
01828    _PROTOTYPE( double log,   (double _x)                                   );
01829    _PROTOTYPE( double log10, (double _x)                                   );
01830    _PROTOTYPE( double modf,  (double _x, double *_iptr)                    );
01831    _PROTOTYPE( double pow,   (double _x, double _y)                        );
01832    _PROTOTYPE( double sin,   (double _x)                                   );
01833    _PROTOTYPE( double sinh,  (double _x)                                   );
01834    _PROTOTYPE( double sqrt,  (double _x)                                   );
01835    _PROTOTYPE( double tan,   (double _x)                                   );
01836    _PROTOTYPE( double tanh,  (double _x)                                   );
01837
01838    #endif /* _MATH_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/mathconst.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

01900    /*
01901     * mathconst.h - mathematic constants
01902     */
01903    /* $Header: mathconst.h,v 1.3 89/12/18 13:59:33 eck Exp $ */
01904
01905    #ifndef _MATHCONST_H
01906    #define _MATHCONST_H
01907
01908    /* Some constants (Hart & Cheney) */
01909    #define M_PI            3.14159265358979323846264338327950288
```

```
01910    #define M_2PI           6.28318530717958647692528676655900576
01911    #define M_3PI_4         2.35619449019234492884698253745962716
01912    #define M_PI_2          1.57079632679489661923132169163975144
01913    #define M_3PI_8         1.17809724509617246442349126872981358
01914    #define M_PI_4          0.78539816339744830961566084581987572
01915    #define M_PI_8          0.39269908169872415480783042290993786
01916    #define M_1_PI          0.31830988618379067153776752674502872
01917    #define M_2_PI          0.63661977236758134307553505349005744
01918    #define M_4_PI          1.27323954473516268615107010698011488
01919    #define M_E             2.71828182845904523536028747135266250
01920    #define M_LOG2E         1.44269504088896340735992468100189213
01921    #define M_LOG10E        0.43429448190325182765112891891660508
01922    #define M_LN2           0.69314718055994530941723212145817657
01923    #define M_LN10          2.30258509299404568401799145468436421
01924    #define M_SQRT2         1.41421356237309504880168872420969808
01925    #define M_1_SQRT2       0.70710678118654752440084436210484904
01926    #define M_EULER         0.57721566490153286060651209008240243
01927
01928    #endif /* _MATHCONST_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/pwd.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
02000    /* The <pwd.h> header defines the items in the password file. */
02001
02002    #ifndef _PWD_H
02003    #define _PWD_H
02004
02005    struct passwd {
02006      char *pw_name;                /* login name */
02007      uid_t pw_uid;                 /* uid corresponding to the name */
02008      gid_t pw_gid;                 /* gid corresponding to the name */
02009      char *pw_dir;                 /* user's home directory */
02010      char *pw_shell;               /* name of the user's shell */
02011
02012      /* The following members are not defined by POSIX. */
02013      char *pw_passwd;              /* password information */
02014      char *pw_gecos;               /* just in case you have a GE 645 around */
02015    };
02016
02017
02018    /* Function Prototypes. */
02019    #ifndef _ANSI_H
02020    #include <ansi.h>
02021    #endif
02022
02023    _PROTOTYPE( struct passwd *getpwnam, (const char *_name)               );
02024    _PROTOTYPE( struct passwd *getpwuid, (Uid_t _uid)                      );
02025
02026    #ifdef _MINIX
02027    _PROTOTYPE( void endpwent, (void)                                      );
02028    _PROTOTYPE( struct passwd *getpwent, (void)                            );
02029    _PROTOTYPE( int setpwent, (void)                                       );
02030    _PROTOTYPE( void setpwfile, (const char *_file)                        );
02031    #endif
02032
02033    #endif /* _PWD_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/regexp.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

02100    /* The <regexp.h> header is used by the (V8-compatible) regexp(3) routines. */
02101
02102    #ifndef _REGEXP_H
02103    #define _REGEXP_H
02104
02105    #define CHARBITS 0377
02106    #define NSUBEXP  10
02107    typedef struct regexp {
02108            char *startp[NSUBEXP];
02109            char *endp[NSUBEXP];
02110            char regstart;          /* Internal use only. */
02111            char reganch;           /* Internal use only. */
02112            char *regmust;          /* Internal use only. */
02113            int regmlen;            /* Internal use only. */
02114            char program[1];        /* Unwarranted chumminess with compiler. */
02115    } regexp;
02116
02117    /* Function Prototypes. */
02118    #ifndef _ANSI_H
02119    #include <ansi.h>
02120    #endif
02121
02122    _PROTOTYPE( regexp *regcomp, (char *_exp)                               );
02123    _PROTOTYPE( int regexec, (regexp *_prog, char *_string, int _bolflag)   );
02124    _PROTOTYPE( void regsub, (regexp *_prog, char *_source, char *_dest)    );
02125    _PROTOTYPE( void regerror, (char *_message)                             );
02126
02127    #endif /* _REGEXP_H */



++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/setjmp.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

02200    /* The <setjmp.h> header relates to the C phenomenon known as setjmp/longjmp.
02201     * It is used to escape out of the current situation into a previous one.
02202     * A typical example is in an editor, where hitting DEL breaks off the current
02203     * command and puts the editor back in the main loop, though care has to be
02204     * taken when the DEL occurs while executing a library function, since
02205     * some of them are not reentrant.
02206     *
02207     * POSIX does not require the process signal mask to be saved and restored
02208     * during setjmp/longjmp.  However, the current implementation does this
02209     * in order to agree with OSF/1 and other BSD derived systems.
02210     *
02211     * The pair of functions _setjmp/_longjmp may be used when the signal
02212     * mask is not to be saved/restored.  These functions are traditional
02213     * in BSD systems.
02214     *
02215     * There are different ways of implementing setjmp/longjmp.  Probably
02216     * the best way is to unify it with signal handling.  This is true for the
02217     * following reasons:  Both setjmp/longjmp and signal delivery must save
02218     * a context so that it may be restored later.  The jmp_buf necessarily
02219     * contains signal information, namely the signal mask to restore.  Both
```

```
02220     * longjmp and the return of a signal handler must trap to the operating
02221     * system to restore the previous signal mask.  Finally, the jmp_buf
02222     * and the sigcontext structure contain the registers to restore.
02223     *
02224     * Some compilers, namely ACK, will not enregister any variables inside a
02225     * function containing a call to setjmp, even if those variables are
02226     * explicitly declared as register variables.  Thus for ACK, the
02227     * identification of the jmp_buf with a sigcontext structure would cause
02228     * unnecessary overhead: the jmp_buf has room for all the registers, but
02229     * the only registers that need to be saved are the stack pointer,
02230     * frame pointer, and program counter.
02231     *
02232     * So, for ACK a jmp_buf is much smaller than a sigcontext structure, and
02233     * longjmp does not directly call sigreturn.  Instead, longjmp calls a
02234     * front-end function which initializes the appropriate fields of a
02235     * sigcontext structure, marks this structure as containing no valid
02236     * general purpose registers, and then calls sigreturn.
02237     *
02238     * The POSIX sigjmp_buf is identical to the jmp_buf in all cases.
02239     *
02240     * Different compilers have different symbols that they recognize as
02241     * setjmp symbols.  ACK recognizes __setjmp, the GNU C compiler
02242     * recognizes setjmp and _setjmp, and BCC recognizes all three.
02243     * When these symbols occur within a function, the compiler may keep
02244     * all local variables on the stack, avoid certain optimizations, or
02245     * pass hidden arguments to the setjmp function.
02246     *
02247     * Thus, setjmp implementations vary in two independent ways which may
02248     * be identified through the following preprocessor tokens:
02249     *
02250     * _SETJMP_SYMBOL -- If 0, this means the compiler treats setjmp and _setjmp
02251     * specially.  If 1, this means the compiler treats __setjmp specially.
02252     *
02253     * _SETJMP_SAVES_REGS -- If 1, this means setjmp/longjmp must explicitly
02254     * save and restore all registers.  This also implies that a jmp_buf is
02255     * different than a sigcontext structure.  If 0, this means that the compiler
02256     * will not use register variables within a function that calls one of
02257     * its SETJMP_SYMBOLs.
02258     *
02259     * When _SETJMP_SYMBOL = 1, the implementation has a few dozen bytes of
02260     * unnecessary overhead.  This happens in the following manner:  a program uses
02261     * _setjmp/_longjmp because it is not interested in saving and restoring the
02262     * signal mask. Nevertheless, because _setjmp expands to the general purpose
02263     * function __setjmp, code for sigprocmask(2) is linked into the program.
02264     */
02265
02266    #ifndef _SETJMP_H
02267    #define _SETJMP_H
02268
02269    #ifndef _ANSI_H
02270    #include <ansi.h>
02271    #endif
02272
02273    #if !defined(__ACK__) && !defined(__BCC__) && !defined(__GNUC__)
02274    #define __ACK__
02275    #endif
02276
02277    #ifdef __ACK__
02278    #define _SETJMP_SYMBOL 1
02279    #define _SETJMP_SAVES_REGS 0
```

```
02280   #endif
02281   #ifdef __BCC__
02282   #define _SETJMP_SYMBOL 0
02283   #define _SETJMP_SAVES_REGS 1
02284   #endif
02285   #ifdef __GNUC__
02286   #define _SETJMP_SYMBOL 0
02287   #define _SETJMP_SAVES_REGS 1
02288   #endif
02289
02290   /* The jmp_buf data type.  Do not change the order of these fields -- some
02291    * C library code refers to these fields by name.  When _SETJMP_SAVES_REGS
02292    * is 1, the file <sys/jmp_buf.h> gives the usage of the sixteen registers.
02293    */
02294   typedef struct {
02295     int __flags;                  /* XXX - long might give better alignment */
02296     long __mask;                  /* must have size >= sizeof(sigset_t) */
02297   #if (_SETJMP_SAVES_REGS == 0)
02298     _PROTOTYPE(void (*__pc),(void));      /* program counter */
02299     void *__sp;                   /* stack pointer */
02300     void *__lb;                   /* local base (ACKspeak for frame pointer) */
02301   #else
02302     void *__regs[16];             /* size is machine dependent */
02303   #endif
02304   } jmp_buf[1];
02305
02306   #if (_SETJMP_SYMBOL == 1)
02307
02308   _PROTOTYPE( int __setjmp, (jmp_buf _env, int _savemask)                );
02309   _PROTOTYPE( void longjmp, (jmp_buf _env, int _val)                     );
02310   _PROTOTYPE(int sigjmp, (jmp_buf _jb, int _retval)                      );
02311
02312   #define setjmp(env)     __setjmp((env), 1)
02313
02314   #ifdef _MINIX
02315   #define _setjmp(env)    __setjmp((env), 0)
02316   _PROTOTYPE(void _longjmp, (jmp_buf _env, int _val)                     );
02317   #endif
02318
02319   #ifdef _POSIX_SOURCE
02320   typedef jmp_buf sigjmp_buf;
02321   _PROTOTYPE( void siglongjmp, (sigjmp_buf _env, int _val)               );
02322
02323   #define sigsetjmp(env, savemask) __setjmp((env), (savemask))
02324   #endif /* _POSIX_SOURCE */
02325
02326   #endif /* _SETJMP_SYMBOL == 1 */
02327
02328   #if (_SETJMP_SYMBOL == 0)
02329
02330   _PROTOTYPE( int setjmp, (jmp_buf _env)                                 );
02331   _PROTOTYPE( void longjmp, (jmp_buf _env, int _val)                     );
02332
02333   #ifdef _MINIX
02334   _PROTOTYPE( int _setjmp, (jmp_buf _env)                                );
02335   _PROTOTYPE( void _longjmp, (jmp_buf _env, int _val)                    );
02336   #endif
02337
02338   #ifdef _POSIX_SOURCE
02339   #define sigjmp_buf jmp_buf
```

```
02340    _PROTOTYPE( void siglongjmp, (sigjmp_buf _env, int _val)               );
02341    /* XXX - the name _setjmp is no good - that's why ACK used __setjmp. */
02342    #define sigsetjmp(env, savemask) ((savemask) ? setjmp(env) : _setjmp(env))
02343    #endif /* _POSIX_SOURCE */
02344
02345    #endif /* _SETJMP_SYMBOL == 0 */
02346
02347    #endif /* _SETJMP_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/sgtty.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

02400    /* The <sgtty.h> header contains data structures for ioctl(). */
02401
02402    #ifndef _SGTTY_H
02403    #define _SGTTY_H
02404
02405    /* Should not be used, nor extended. Termios.h is the replacement for
02406     * sgtty.h for tty functions, and ioctl replaced code should be moved to
02407     * sys/ioctl.h and specific header files in the sys, or minix directory.
02408     */
02409    #include <sys/ioctl.h>          /* Ouch. */
02410
02411    struct sgttyb {
02412      char sg_ispeed;               /* input speed */
02413      char sg_ospeed;               /* output speed */
02414      char sg_erase;                /* erase character */
02415      char sg_kill;                 /* kill character */
02416      int  sg_flags;                /* mode flags */
02417    };
02418
02419    struct tchars {
02420      char t_intrc;                 /* SIGINT char */
02421      char t_quitc;                 /* SIGQUIT char */
02422      char t_startc;                /* start output (initially CTRL-Q) */
02423      char t_stopc;                 /* stop output  (initially CTRL-S) */
02424      char t_eofc;                  /* EOF (initially CTRL-D) */
02425      char t_brkc;                  /* input delimiter (like nl) */
02426    };
02427
02428    #if !_SYSTEM                    /* the kernel doesn't want to see the rest */
02429
02430    /* Field names */
02431    #define XTABS        0006000    /* do tab expansion */
02432    #define BITS8        0001400    /* 8 bits/char */
02433    #define BITS7        0001000    /* 7 bits/char */
02434    #define BITS6        0000400    /* 6 bits/char */
02435    #define BITS5        0000000    /* 5 bits/char */
02436    #define EVENP        0000200    /* even parity */
02437    #define ODDP         0000100    /* odd parity */
02438    #define RAW          0000040    /* enable raw mode */
02439    #define CRMOD        0000020    /* map lf to cr + lf */
02440    #define ECHO         0000010    /* echo input */
02441    #define CBREAK       0000002    /* enable cbreak mode */
02442    #define COOKED       0000000    /* neither CBREAK nor RAW */
02443
02444    #define DCD          0100000    /* Data Carrier Detect */
```

```
02445
02446   /* Line speeds */
02447   #define B0                 0    /* code for line-hangup */
02448   #define B110               1
02449   #define B300               3
02450   #define B1200             12
02451   #define B2400             24
02452   #define B4800             48
02453   #define B9600             96
02454   #define B19200           192
02455   #define B38400           195
02456   #define B57600           194
02457   #define B115200          193
02458
02459   /* Things Minix supports but not properly */
02460   /* the divide-by-100 encoding ain't too hot */
02461   #define ANYP         0000300
02462   #define B50                0
02463   #define B75                0
02464   #define B134               0
02465   #define B150               0
02466   #define B200               2
02467   #define B600               6
02468   #define B1800             18
02469   #define B3600             36
02470   #define B7200             72
02471   #define EXTA             192
02472   #define EXTB               0
02473
02474   /* Things Minix doesn't support but are fairly harmless if used */
02475   #define NLDELAY      0001400
02476   #define TBDELAY      0006000
02477   #define CRDELAY      0030000
02478   #define VTDELAY      0040000
02479   #define BSDELAY      0100000
02480   #define ALLDELAY     0177400
02481
02482   /* Copied from termios.h: */
02483   struct winsize
02484   {
02485           unsigned short  ws_row;         /* rows, in characters */
02486           unsigned short  ws_col;         /* columns, in characters */
02487           unsigned short  ws_xpixel;      /* horizontal size, pixels */
02488           unsigned short  ws_ypixel;      /* vertical size, pixels */
02489   };
02490   #endif /* !_SYSTEM */
02491   #endif /* _SGTTY_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/signal.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

02500    /* The <signal.h> header defines all the ANSI and POSIX signals.
02501     * MINIX supports all the signals required by POSIX. They are defined below.
02502     * Some additional signals are also supported.
02503     */
02504
02505    #ifndef _SIGNAL_H
02506    #define _SIGNAL_H
02507
02508    #ifndef _ANSI_H
02509    #include <ansi.h>
02510    #endif
02511
02512    /* Here are types that are closely associated with signal handling. */
02513    typedef int sig_atomic_t;
02514
02515    #ifdef _POSIX_SOURCE
02516    #ifndef _SIGSET_T
02517    #define _SIGSET_T
02518    typedef unsigned long sigset_t;
02519    #endif
02520    #endif
02521
02522    #define _NSIG             16    /* number of signals used */
02523
02524    #define SIGHUP             1    /* hangup */
02525    #define SIGINT             2    /* interrupt (DEL) */
02526    #define SIGQUIT            3    /* quit (ASCII FS) */
02527    #define SIGILL             4    /* illegal instruction */
02528    #define SIGTRAP            5    /* trace trap (not reset when caught) */
02529    #define SIGABRT            6    /* IOT instruction */
02530    #define SIGIOT             6    /* SIGABRT for people who speak PDP-11 */
02531    #define SIGUNUSED          7    /* spare code */
02532    #define SIGFPE             8    /* floating point exception */
02533    #define SIGKILL            9    /* kill (cannot be caught or ignored) */
02534    #define SIGUSR1           10    /* user defined signal # 1 */
02535    #define SIGSEGV           11    /* segmentation violation */
02536    #define SIGUSR2           12    /* user defined signal # 2 */
02537    #define SIGPIPE           13    /* write on a pipe with no one to read it */
02538    #define SIGALRM           14    /* alarm clock */
02539    #define SIGTERM           15    /* software termination signal from kill */
02540
02541    #define SIGEMT             7    /* obsolete */
02542    #define SIGBUS            10    /* obsolete */
02543
02544    /* POSIX requires the following signals to be defined, even if they are
02545     * not supported.  Here are the definitions, but they are not supported.
02546     */
02547    #define SIGCHLD           17    /* child process terminated or stopped */
02548    #define SIGCONT           18    /* continue if stopped */
02549    #define SIGSTOP           19    /* stop signal */
02550    #define SIGTSTP           20    /* interactive stop signal */
02551    #define SIGTTIN           21    /* background process wants to read */
02552    #define SIGTTOU           22    /* background process wants to write */
02553
02554    /* The sighandler_t type is not allowed unless _POSIX_SOURCE is defined. */
```

```
02555   #ifdef _POSIX_SOURCE
02556   #define __sighandler_t sighandler_t
02557   #else
02558   typedef void (*__sighandler_t) (int);
02559   #endif
02560
02561   /* Macros used as function pointers. */
02562   #define SIG_ERR    ((__sighandler_t) -1)        /* error return */
02563   #define SIG_DFL    ((__sighandler_t)  0)        /* default signal handling */
02564   #define SIG_IGN    ((__sighandler_t)  1)        /* ignore signal */
02565   #define SIG_HOLD   ((__sighandler_t)  2)        /* block signal */
02566   #define SIG_CATCH  ((__sighandler_t)  3)        /* catch signal */
02567
02568   #ifdef _POSIX_SOURCE
02569   struct sigaction {
02570     __sighandler_t sa_handler;    /* SIG_DFL, SIG_IGN, or pointer to function */
02571     sigset_t sa_mask;             /* signals to be blocked during handler */
02572     int sa_flags;                 /* special flags */
02573   };
02574
02575   /* Fields for sa_flags. */
02576   #define SA_ONSTACK   0x0001     /* deliver signal on alternate stack */
02577   #define SA_RESETHAND 0x0002     /* reset signal handler when signal caught */
02578   #define SA_NODEFER   0x0004     /* don't block signal while catching it */
02579   #define SA_RESTART   0x0008     /* automatic system call restart */
02580   #define SA_SIGINFO   0x0010     /* extended signal handling */
02581   #define SA_NOCLDWAIT 0x0020     /* don't create zombies */
02582   #define SA_NOCLDSTOP 0x0040     /* don't receive SIGCHLD when child stops */
02583
02584   /* POSIX requires these values for use with sigprocmask(2). */
02585   #define SIG_BLOCK          0    /* for blocking signals */
02586   #define SIG_UNBLOCK        1    /* for unblocking signals */
02587   #define SIG_SETMASK        2    /* for setting the signal mask */
02588   #define SIG_INQUIRE        4    /* for internal use only */
02589   #endif  /* _POSIX_SOURCE */
02590
02591   /* POSIX and ANSI function prototypes. */
02592   _PROTOTYPE( int raise, (int _sig)                                       );
02593   _PROTOTYPE( __sighandler_t signal, (int _sig, __sighandler_t _func)     );
02594
02595   #ifdef _POSIX_SOURCE
02596   _PROTOTYPE( int kill, (pid_t _pid, int _sig)                            );
02597   _PROTOTYPE( int sigaction,
02598       (int _sig, const struct sigaction *_act, struct sigaction *_oact)   );
02599   _PROTOTYPE( int sigaddset, (sigset_t *_set, int _sig)                   );
02600   _PROTOTYPE( int sigdelset, (sigset_t *_set, int _sig)                   );
02601   _PROTOTYPE( int sigemptyset, (sigset_t *_set)                           );
02602   _PROTOTYPE( int sigfillset, (sigset_t *_set)                            );
02603   _PROTOTYPE( int sigismember, (sigset_t *_set, int _sig)                 );
02604   _PROTOTYPE( int sigpending, (sigset_t *_set)                            );
02605   _PROTOTYPE( int sigprocmask,
02606               (int _how, const sigset_t *_set, sigset_t *_oset)           );
02607   _PROTOTYPE( int sigsuspend, (const sigset_t *_sigmask)                  );
02608   #endif
02609
02610   #endif /* _SIGNAL_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/stdarg.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

02700    /* The <stdarg.h> header is ANSI's way to handle variable numbers of params.
02701     * Some programming languages require a function that is declared with n
02702     * parameters to be called with n parameters.  C does not.  A function may
02703     * called with more parameters than it is declared with.  The well-known
02704     * printf function, for example, may have arbitrarily many parameters.
02705     * The question arises how one can access all the parameters in a portable
02706     * way.  The C standard defines three macros that programs can use to
02707     * advance through the parameter list.  The definition of these macros for
02708     * MINIX are given in this file.  The three macros are:
02709     *
02710     *      va_start(ap, parmN)     prepare to access parameters
02711     *      va_arg(ap, type)        get next parameter value and type
02712     *      va_end(ap)              access is finished
02713     *
02714     * Ken Thompson's famous line from V6 UNIX is equally applicable to this file:
02715     *
02716     *      "You are not expected to understand this"
02717     *
02718     */
02719
02720    #ifndef _STDARG_H
02721    #define _STDARG_H
02722
02723
02724    #ifdef __GNUC__
02725    /* The GNU C-compiler uses its own, but similar varargs mechanism. */
02726
02727    typedef char *va_list;
02728
02729    /* Amount of space required in an argument list for an arg of type TYPE.
02730     * TYPE may alternatively be an expression whose type is used.
02731     */
02732
02733    #define __va_rounded_size(TYPE)  \
02734      (((sizeof (TYPE) + sizeof (int) - 1) / sizeof (int)) * sizeof (int))
02735
02736    #if __GNUC__ < 2
02737
02738    #ifndef __sparc__
02739    #define va_start(AP, LASTARG)                                           \
02740     (AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
02741    #else
02742    #define va_start(AP, LASTARG)                                           \
02743     (__builtin_saveregs (),                                                \
02744      AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
02745    #endif
02746
02747    void va_end (va_list);          /* Defined in gnulib */
02748    #define va_end(AP)
02749
02750    #define va_arg(AP, TYPE)                                                \
02751     (AP += __va_rounded_size (TYPE),                                       \
02752      *((TYPE *) (AP - __va_rounded_size (TYPE))))
02753
02754    #else   /* __GNUC__ >= 2 */
```

```
02755
02756    #ifndef __sparc__
02757    #define va_start(AP, LASTARG)                                          \
02758     (AP = ((char *) __builtin_next_arg ()))
02759    #else
02760    #define va_start(AP, LASTARG)                                         \
02761      (__builtin_saveregs (), AP = ((char *) __builtin_next_arg ()))
02762    #endif
02763
02764    void va_end (va_list);          /* Defined in libgcc.a */
02765    #define va_end(AP)
02766
02767    #define va_arg(AP, TYPE)                                               \
02768     (AP = ((char *) (AP)) += __va_rounded_size (TYPE),                    \
02769      *((TYPE *) ((char *) (AP) - __va_rounded_size (TYPE))))
02770
02771    #endif  /* __GNUC__ >= 2 */
02772
02773    #else   /* not __GNUC__ */
02774
02775
02776    typedef char *va_list;
02777
02778    #define __vasz(x)               ((sizeof(x)+sizeof(int)-1) & ~(sizeof(int) -1))
02779
02780    #define va_start(ap, parmN)     ((ap) = (va_list)&parmN + __vasz(parmN))
02781    #define va_arg(ap, type)      \
02782      (*((type *)((va_list)((ap) = (void *)((va_list)(ap) + __vasz(type))) \
02783                                                    - __vasz(type))))
02784    #define va_end(ap)
02785
02786
02787    #endif /* __GNUC__ */
02788
02789    #endif /* _STDARG_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/stddef.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

02800    /* The <stddef.h> header defines certain commonly used macros. */
02801
02802    #ifndef _STDDEF_H
02803    #define _STDDEF_H
02804
02805    #define NULL   ((void *)0)
02806
02807    /* The following is not portable, but the compiler accepts it. */
02808    #define offsetof(type, ident)   ((size_t) (unsigned long) &((type *)0)->ident)
02809
02810    #if _EM_PSIZE == _EM_WSIZE
02811    typedef int ptrdiff_t;          /* result of subtracting two pointers */
02812    #else   /* _EM_PSIZE == _EM_LSIZE */
02813    typedef long ptrdiff_t;
02814    #endif
```

```
02815
02816    #ifndef _SIZE_T
02817    #define _SIZE_T
02818    typedef unsigned int size_t;    /* type returned by sizeof */
02819    #endif
02820
02821    #ifndef _WCHAR_T
02822    #define _WCHAR_T
02823    typedef char wchar_t;           /* type expanded character set */
02824    #endif
02825
02826    #endif /* _STDDEF_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/stdio.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

02900    /*
02901     * stdio.h - input/output definitions
02902     *
02903     * (c) copyright 1987 by the Vrije Universiteit, Amsterdam, The Netherlands.
02904     * See the copyright notice in the ACK home directory, in the file "Copyright".
02905     */
02906    /* $Header: stdio.h,v 1.3 89/12/18 14:00:10 eck Exp $ */
02907
02908    #ifndef _STDIO_H
02909    #define _STDIO_H
02910
02911    /*
02912     * Focus point of all stdio activity.
02913     */
02914    typedef struct __iobuf {
02915            int             _count;
02916            int             _fd;
02917            int             _flags;
02918            int             _bufsiz;
02919            unsigned char   *_buf;
02920            unsigned char   *_ptr;
02921    } FILE;
02922
02923    #define _IOFBF          0x000
02924    #define _IOREAD         0x001
02925    #define _IOWRITE        0x002
02926    #define _IONBF          0x004
02927    #define _IOMYBUF        0x008
02928    #define _IOEOF          0x010
02929    #define _IOERR          0x020
02930    #define _IOLBF          0x040
02931    #define _IOREADING      0x080
02932    #define _IOWRITING      0x100
02933    #define _IOAPPEND       0x200
02934
02935    /* The following definitions are also in <unistd.h>. They should not
02936     * conflict.
02937     */
02938    #define SEEK_SET        0
02939    #define SEEK_CUR        1
```

```
02940    #define SEEK_END        2
02941
02942    #define stdin           (&__stdin)
02943    #define stdout          (&__stdout)
02944    #define stderr          (&__stderr)
02945
02946    #define BUFSIZ          1024
02947    #define NULL            ((void *)0)
02948    #define EOF             (-1)
02949
02950    #define FOPEN_MAX       20
02951
02952    #define FILENAME_MAX    14
02953    #define TMP_MAX         999
02954    #define L_tmpnam        (sizeof("/tmp/") + FILENAME_MAX)
02955    #define __STDIO_VA_LIST__       void *
02956
02957    typedef long int        fpos_t;
02958
02959    #ifndef _SIZE_T
02960    #define _SIZE_T
02961    typedef unsigned int    size_t;         /* type returned by sizeof */
02962    #endif /* _SIZE_T */
02963
02964    extern FILE     *__iotab[FOPEN_MAX];
02965    extern FILE     __stdin, __stdout, __stderr;
02966
02967    #ifndef _ANSI_H
02968    #include <ansi.h>
02969    #endif
02970
02971    _PROTOTYPE( int remove, (const char *_filename)                         );
02972    _PROTOTYPE( int rename, (const char *_old, const char *_new)            );
02973    _PROTOTYPE( FILE *tmpfile, (void)                                       );
02974    _PROTOTYPE( char *tmpnam, (char *_s)                                    );
02975    _PROTOTYPE( int fclose, (FILE *_stream)                                 );
02976    _PROTOTYPE( int fflush, (FILE *_stream)                                 );
02977    _PROTOTYPE( FILE *fopen, (const char *_filename, const char *_mode)     );
02978    _PROTOTYPE( FILE *freopen,
02979                (const char *_filename, const char *_mode, FILE *_stream)   );
02980    _PROTOTYPE( void setbuf, (FILE *_stream, char *_buf)                    );
02981    _PROTOTYPE( int setvbuf,
02982                    (FILE *_stream, char *_buf, int _mode, size_t _size)    );
02983    _PROTOTYPE( int fprintf, (FILE *_stream, const char *_format, ...)      );
02984    _PROTOTYPE( int printf, (const char *_format, ...)                      );
02985    _PROTOTYPE( int sprintf, (char *_s, const char *_format, ...)           );
02986    _PROTOTYPE( int vfprintf,
02987                    (FILE *_stream, const char *_format, char *_arg)        );
02988    _PROTOTYPE( int vprintf, (const char *_format, char *_arg)              );
02989    _PROTOTYPE( int vsprintf, (char *_s, const char *_format, char *_arg)   );
02990    _PROTOTYPE( int fscanf, (FILE *_stream, const char *_format, ...)       );
02991    _PROTOTYPE( int scanf, (const char *_format, ...)                       );
02992    _PROTOTYPE( int sscanf, (const char *_s, const char *_format, ...)      );
02993    #define vfscanf _doscan
02994    _PROTOTYPE( int vfscanf, (FILE *_stream, const char *_format, char *_arg));
02995    _PROTOTYPE( int vscanf, (const char *_format, char *_arg)               );
02996    _PROTOTYPE( int vsscanf, (const char *_s, const char *_format, char *_arg));
02997    _PROTOTYPE( int fgetc, (FILE *_stream)                                  );
02998    _PROTOTYPE( char *fgets, (char *_s, int _n, FILE *_stream)              );
02999    _PROTOTYPE( int fputc, (int _c, FILE *_stream)                          );
```

```
03000    _PROTOTYPE( int fputs, (const char *_s, FILE *_stream)                 );
03001    _PROTOTYPE( int getc, (FILE *_stream)                                  );
03002    _PROTOTYPE( int getchar, (void)                                        );
03003    _PROTOTYPE( char *gets, (char *_s)                                     );
03004    _PROTOTYPE( int putc, (int _c, FILE *_stream)                          );
03005    _PROTOTYPE( int putchar, (int _c)                                      );
03006    _PROTOTYPE( int puts, (const char *_s)                                 );
03007    _PROTOTYPE( int ungetc, (int _c, FILE *_stream)                        );
03008    _PROTOTYPE( size_t fread,
03009                (void *_ptr, size_t _size, size_t _nmemb, FILE *_stream)    );
03010    _PROTOTYPE( size_t fwrite,
03011            (const void *_ptr, size_t _size, size_t _nmemb, FILE *_stream)  );
03012    _PROTOTYPE( int fgetpos, (FILE *_stream, fpos_t *_pos)                 );
03013    _PROTOTYPE( int fseek, (FILE *_stream, long _offset, int _whence)      );
03014    _PROTOTYPE( int fsetpos, (FILE *_stream, fpos_t *_pos)                 );
03015    _PROTOTYPE( long ftell, (FILE *_stream)                                );
03016    _PROTOTYPE( void rewind, (FILE *_stream)                               );
03017    _PROTOTYPE( void clearerr, (FILE *_stream)                             );
03018    _PROTOTYPE( int feof, (FILE *_stream)                                  );
03019    _PROTOTYPE( int ferror, (FILE *_stream)                                );
03020    _PROTOTYPE( void perror, (const char *_s)                              );
03021    _PROTOTYPE( int __fillbuf, (FILE *_stream)                             );
03022    _PROTOTYPE( int __flushbuf, (int _c, FILE *_stream)                    );
03023
03024    #define getchar()       getc(stdin)
03025    #define putchar(c)      putc(c,stdout)
03026    #define getc(p)         (--(p)->_count >= 0 ? (int) (*(p)->_ptr++) : \
03027                                 __fillbuf(p))
03028    #define putc(c, p)      (--(p)->_count >= 0 ? \
03029                             (int) (*(p)->_ptr++ = (c)) : \
03030                             __flushbuf((c),(p)))
03031
03032    #define feof(p)         (((p)->_flags & _IOEOF) != 0)
03033    #define ferror(p)       (((p)->_flags & _IOERR) != 0)
03034    #define clearerr(p)     ((p)->_flags &= ~(_IOERR|_IOEOF))
03035
03036    #ifdef _POSIX_SOURCE
03037    _PROTOTYPE( int fileno, (FILE *_stream)                                );
03038    _PROTOTYPE (FILE *fdopen, (int _fildes, const char *_types) );
03039    #define fileno(stream)          ((stream)->_fd)
03040    #define L_ctermid 255   /* required by POSIX */
03041    #define L_cuserid 255   /* required by POSIX */
03042    #endif
03043
03044    #ifdef _MINIX
03045    _PROTOTYPE(FILE *popen, (const char *_command, const char *_type));
03046    _PROTOTYPE(int pclose, (FILE *_stream));
03047    #endif
03048
03049    #endif /* _STDIO_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/stdlib.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

03100    /* The <stdlib.h> header defines certain common macros, types, and functions.*/
03101
03102    #ifndef _STDLIB_H
03103    #define _STDLIB_H
03104
03105    /* The macros are NULL, EXIT_FAILURE, EXIT_SUCCESS, RAND_MAX, and MB_CUR_MAX.*/
03106    #define NULL    ((void *)0)
03107
03108    #define EXIT_FAILURE       1    /* standard error return using exit() */
03109    #define EXIT_SUCCESS       0    /* successful return using exit() */
03110    #define RAND_MAX       32767    /* largest value generated by rand() */
03111    #define MB_CUR_MAX         1    /* max value of multibyte character in MINIX */
03112
03113    typedef struct { int quot, rem; } div_t;
03114    typedef struct { long quot, rem; } ldiv_t;
03115
03116    /* The types are size_t, wchar_t, div_t, and ldiv_t. */
03117    #ifndef _SIZE_T
03118    #define _SIZE_T
03119    typedef unsigned int size_t;    /* type returned by sizeof */
03120    #endif
03121
03122    #ifndef _WCHAR_T
03123    #define _WCHAR_T
03124    typedef char wchar_t;           /* type expanded character set */
03125    #endif
03126
03127    /* Function Prototypes. */
03128    #ifndef _ANSI_H
03129    #include <ansi.h>
03130    #endif
03131
03132    _PROTOTYPE( void abort, (void)                                          );
03133    _PROTOTYPE( int abs, (int _j)                                           );
03134    _PROTOTYPE( int atexit, (void (*_func)(void))                           );
03135    _PROTOTYPE( double atof, (const char *_nptr)                            );
03136    _PROTOTYPE( int atoi, (const char *_nptr)                               );
03137    _PROTOTYPE( long atol, (const char *_nptr)                              );
03138    _PROTOTYPE( void *calloc, (size_t _nmemb, size_t _size)                 );
03139    _PROTOTYPE( div_t div, (int _numer, int _denom)                         );
03140    _PROTOTYPE( void exit, (int _status)                                    );
03141    _PROTOTYPE( void free, (void *_ptr)                                     );
03142    _PROTOTYPE( char *getenv, (const char *_name)                           );
03143    _PROTOTYPE( long labs, (long _j)                                        );
03144    _PROTOTYPE( ldiv_t ldiv, (long _numer, long _denom)                     );
03145    _PROTOTYPE( void *malloc, (size_t _size)                                );
03146    _PROTOTYPE( int mblen, (const char *_s, size_t _n)                      );
03147    _PROTOTYPE( size_t mbstowcs, (wchar_t *_pwcs, const char *_s, size_t _n));
03148    _PROTOTYPE( int mbtowc, (wchar_t *_pwc, const char *_s, size_t _n)      );
03149    _PROTOTYPE( int rand, (void)                                            );
03150    _PROTOTYPE( void *realloc, (void *_ptr, size_t _size)                   );
03151    _PROTOTYPE( void srand, (unsigned int _seed)                            );
03152    _PROTOTYPE( double strtod, (const char *_nptr, char **_endptr)          );
03153    _PROTOTYPE( long strtol, (const char *_nptr, char **_endptr, int _base) );
03154    _PROTOTYPE( int system, (const char *_string)                           );
```

```
03155    _PROTOTYPE( size_t wcstombs, (char *_s, const wchar_t *_pwcs, size_t _n));
03156    _PROTOTYPE( int wctomb, (char *_s, wchar_t _wchar)                      );
03157    _PROTOTYPE( void *bsearch, (const void *_key, const void *_base,
03158            size_t _nmemb, size_t _size,
03159            int (*compar) (const void *, const void *))                     );
03160    _PROTOTYPE( void qsort, (void *_base, size_t _nmemb, size_t _size,
03161            int (*compar) (const void *, const void *))                     );
03162    _PROTOTYPE( unsigned long int strtoul,
03163                            (const char *_nptr, char **_endptr, int _base)  );
03164
03165    #ifdef _MINIX
03166    _PROTOTYPE( int putenv, (const char *_name)                             );
03167    _PROTOTYPE(int getopt, (int _argc, char **_argv, char *_opts));
03168    extern char *optarg;
03169    extern int optind, opterr, optopt;
03170    #endif /* _MINIX */
03171
03172    #endif /* STDLIB_H */
```

++
include/string.h
++

```
03200    /* The <string.h> header contains prototypes for the string handling
03201     * functions.
03202     */
03203
03204    #ifndef _STRING_H
03205    #define _STRING_H
03206
03207    #define NULL    ((void *)0)
03208
03209    #ifndef _SIZE_T
03210    #define _SIZE_T
03211    typedef unsigned int size_t;    /* type returned by sizeof */
03212    #endif /*_SIZE_T */
03213
03214    /* Function Prototypes. */
03215    #ifndef _ANSI_H
03216    #include <ansi.h>
03217    #endif
03218
03219    _PROTOTYPE( void *memchr, (const void *_s, int _c, size_t _n)           );
03220    _PROTOTYPE( int memcmp, (const void *_s1, const void *_s2, size_t _n)   );
03221    _PROTOTYPE( void *memcpy, (void *_s1, const void *_s2, size_t _n)       );
03222    _PROTOTYPE( void *memmove, (void *_s1, const void *_s2, size_t _n)      );
03223    _PROTOTYPE( void *memset, (void *_s, int _c, size_t _n)                 );
03224    _PROTOTYPE( char *strcat, (char *_s1, const char *_s2)                  );
03225    _PROTOTYPE( char *strchr, (const char *_s, int _c)                      );
03226    _PROTOTYPE( int strncmp, (const char *_s1, const char *_s2, size_t _n)  );
03227    _PROTOTYPE( int strcmp, (const char *_s1, const char *_s2)              );
03228    _PROTOTYPE( int strcoll, (const char *_s1, const char *_s2)             );
03229    _PROTOTYPE( char *strcpy, (char *_s1, const char *_s2)                  );
03230    _PROTOTYPE( size_t strcspn, (const char *_s1, const char *_s2)          );
03231    _PROTOTYPE( char *strerror, (int _errnum)                               );
03232    _PROTOTYPE( size_t strlen, (const char *_s)                             );
03233    _PROTOTYPE( char *strncat, (char *_s1, const char *_s2, size_t _n)      );
03234    _PROTOTYPE( char *strncpy, (char *_s1, const char *_s2, size_t _n)      );
```

```
03235    _PROTOTYPE( char *strpbrk, (const char *_s1, const char *_s2)          );
03236    _PROTOTYPE( char *strrchr, (const char *_s, int _c)                    );
03237    _PROTOTYPE( size_t strspn, (const char *_s1, const char *_s2)          );
03238    _PROTOTYPE( char *strstr, (const char *_s1, const char *_s2)           );
03239    _PROTOTYPE( char *strtok, (char *_s1, const char *_s2)                 );
03240    _PROTOTYPE( size_t strxfrm, (char *_s1, const char *_s2, size_t _n)    );
03241
03242    #ifdef _MINIX
03243    /* For backward compatibility. */
03244    _PROTOTYPE( char *index, (const char *_s, int _charwanted)             );
03245    _PROTOTYPE( char *rindex, (const char *_s, int _charwanted)            );
03246    _PROTOTYPE( void bcopy, (const void *_src, void *_dst, size_t _length) );
03247    _PROTOTYPE( int bcmp, (const void *_s1, const void *_s2, size_t _length));
03248    _PROTOTYPE( void bzero, (void *_dst, size_t _length)                   );
03249    _PROTOTYPE( void *memccpy, (char *_dst, const char *_src, int _ucharstop,
03250                                                        size_t _size)      );
03251    /* BSD functions */
03252    _PROTOTYPE( int strcasecmp, (const char *_s1, const char *_s2)         );
03253    #endif
03254
03255    #endif /* _STRING_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/tar.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

03300    /* The <tar.h> header is used with the tape archiver, tar. */
03301
03302    #ifndef _TAR_H
03303    #define _TAR_H
03304
03305    #define TBLOCK          512
03306    #define NAMSIZ          100
03307    #define PFXSIZ          155
03308
03309    #define TMODLEN         8
03310    #define TUIDLEN         8
03311    #define TGIDLEN         8
03312    #define TSIZLEN         12
03313    #define TMTMLEN         12
03314    #define TCKSLEN         8
03315
03316    #define TMAGIC          "ustar"
03317    #define TMAGLEN         6
03318    #define TVERSION        "00"
03319    #define TVERSLEN        2
03320    #define TUNMLEN         32
03321    #define TGNMLEN         32
03322    #define TDEVLEN         8
03323
03324    #define REGTYPE         '0'
03325    #define AREGTYPE        '\0'
03326    #define LNKTYPE         '1'
03327    #define SYMTYPE         '2'
03328    #define CHRTYPE         '3'
03329    #define BLKTYPE         '4'
```

```
03330    #define DIRTYPE         '5'
03331    #define FIFOTYPE        '6'
03332    #define CONTTYPE        '7'
03333
03334    #define TSUID           04000
03335    #define TSGID           02000
03336    #define TSVTX           01000
03337
03338    #define TUREAD          00400
03339    #define TUWRITE         00200
03340    #define TUEXEC          00100
03341    #define TGREAD          00040
03342    #define TGWRITE         00020
03343    #define TGEXEC          00010
03344    #define TOREAD          00004
03345    #define TOWRITE         00002
03346    #define TOEXEC          00001
03347
03348    union hblock {
03349      char dummy[TBLOCK];
03350      struct header {
03351            char name[NAMSIZ];
03352            char mode[TMODLEN];
03353            char uid[TUIDLEN];
03354            char gid[TGIDLEN];
03355            char size[TSIZLEN];
03356            char mtime[TMTMLEN];
03357            char chksum[TCKSLEN];
03358            char typeflag;
03359            char linkname[NAMSIZ];
03360            char magic[TMAGLEN];
03361            char version[TVERSLEN];
03362            char uname[TUNMLEN];
03363            char gname[TGNMLEN];
03364            char devmajor[TDEVLEN];
03365            char devminor[TDEVLEN];
03366            char prefix[PFXSIZ];
03367      } dbuf;
03368    };
03369
03370    #endif /* _TAR_H */
```

++

include/termcap.h

++

```
03400    #ifndef _TERMCAP_H
03401    #define _TERMCAP_H
03402
03403    #include <ansi.h>
03404
03405    _PROTOTYPE( int tgetent, (char *_bp, char *_name)                    );
03406    _PROTOTYPE( int tgetflag, (char *_id)                                );
03407    _PROTOTYPE( int tgetnum, (char *_id)                                 );
03408    _PROTOTYPE( char *tgetstr, (char *_id, char **_area)                 );
03409    _PROTOTYPE( char *tgoto, (char *_cm, int _destcol, int _destline)    );
```

```
03410    _PROTOTYPE( int tputs, (char *_cp, int _affcnt, void (*_outc)(int))     );
03411
03412    #endif /* _TERMCAP_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/termios.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
03500    /* The <termios.h> header is used for controlling tty modes. */
03501
03502    #ifndef _TERMIOS_H
03503    #define _TERMIOS_H
03504
03505    typedef unsigned short tcflag_t;
03506    typedef unsigned char cc_t;
03507    typedef unsigned int speed_t;
03508
03509    #define NCCS            20      /* size of cc_c array, some extra space
03510                                     * for extensions. */
03511
03512    /* Primary terminal control structure. POSIX Table 7-1. */
03513    struct termios {
03514      tcflag_t c_iflag;             /* input modes */
03515      tcflag_t c_oflag;             /* output modes */
03516      tcflag_t c_cflag;             /* control modes */
03517      tcflag_t c_lflag;             /* local modes */
03518      speed_t  c_ispeed;            /* input speed */
03519      speed_t  c_ospeed;            /* output speed */
03520      cc_t c_cc[NCCS];              /* control characters */
03521    };
03522
03523    /* Values for termios c_iflag bit map.  POSIX Table 7-2. */
03524    #define BRKINT          0x0001  /* signal interrupt on break */
03525    #define ICRNL           0x0002  /* map CR to NL on input */
03526    #define IGNBRK          0x0004  /* ignore break */
03527    #define IGNCR           0x0008  /* ignore CR */
03528    #define IGNPAR          0x0010  /* ignore characters with parity errors */
03529    #define INLCR           0x0020  /* map NL to CR on input */
03530    #define INPCK           0x0040  /* enable input parity check */
03531    #define ISTRIP          0x0080  /* mask off 8th bit */
03532    #define IXOFF           0x0100  /* enable start/stop input control */
03533    #define IXON            0x0200  /* enable start/stop output control */
03534    #define PARMRK          0x0400  /* mark parity errors in the input queue */
03535
03536    /* Values for termios c_oflag bit map.  POSIX Sec. 7.1.2.3. */
03537    #define OPOST           0x0001  /* perform output processing */
03538
03539    /* Values for termios c_cflag bit map.  POSIX Table 7-3. */
03540    #define CLOCAL          0x0001  /* ignore modem status lines */
03541    #define CREAD           0x0002  /* enable receiver */
03542    #define CSIZE           0x000C  /* number of bits per character */
03543    #define         CS5     0x0000  /* if CSIZE is CS5, characters are 5 bits */
03544    #define         CS6     0x0004  /* if CSIZE is CS6, characters are 6 bits */
03545    #define         CS7     0x0008  /* if CSIZE is CS7, characters are 7 bits */
03546    #define         CS8     0x000C  /* if CSIZE is CS8, characters are 8 bits */
03547    #define CSTOPB          0x0010  /* send 2 stop bits if set, else 1 */
03548    #define HUPCL           0x0020  /* hang up on last close */
03549    #define PARENB          0x0040  /* enable parity on output */
```

```
03550    #define PARODD          0x0080  /* use odd parity if set, else even */
03551
03552    /* Values for termios c_lflag bit map.  POSIX Table 7-4. */
03553    #define ECHO            0x0001  /* enable echoing of input characters */
03554    #define ECHOE           0x0002  /* echo ERASE as backspace */
03555    #define ECHOK           0x0004  /* echo KILL */
03556    #define ECHONL          0x0008  /* echo NL */
03557    #define ICANON          0x0010  /* canonical input (erase and kill enabled) */
03558    #define IEXTEN          0x0020  /* enable extended functions */
03559    #define ISIG            0x0040  /* enable signals */
03560    #define NOFLSH          0x0080  /* disable flush after interrupt or quit */
03561    #define TOSTOP          0x0100  /* send SIGTTOU (job control, not implemented*/
03562
03563    /* Indices into c_cc array.  Default values in parentheses. POSIX Table 7-5. */
03564    #define VEOF               0    /* cc_c[VEOF] = EOF char (^D) */
03565    #define VEOL               1    /* cc_c[VEOL] = EOL char (undef) */
03566    #define VERASE             2    /* cc_c[VERASE] = ERASE char (^H) */
03567    #define VINTR              3    /* cc_c[VINTR] = INTR char (DEL) */
03568    #define VKILL              4    /* cc_c[VKILL] = KILL char (^U) */
03569    #define VMIN               5    /* cc_c[VMIN] = MIN value for timer */
03570    #define VQUIT              6    /* cc_c[VQUIT] = QUIT char (^\) */
03571    #define VTIME              7    /* cc_c[VTIME] = TIME value for timer */
03572    #define VSUSP              8    /* cc_c[VSUSP] = SUSP (^Z, ignored) */
03573    #define VSTART             9    /* cc_c[VSTART] = START char (^S) */
03574    #define VSTOP             10    /* cc_c[VSTOP] = STOP char (^Q) */
03575
03576    #define _POSIX_VDISABLE   (cc_t)0xFF    /* You can't even generate this
03577                                             * character with 'normal' keyboards.
03578                                             * But some language specific keyboards
03579                                             * can generate 0xFF. It seems that all
03580                                             * 256 are used, so cc_t should be a
03581                                             * short...
03582                                             */
03583
03584    /* Values for the baud rate settings.  POSIX Table 7-6. */
03585    #define B0              0x0000  /* hang up the line */
03586    #define B50             0x1000  /* 50 baud */
03587    #define B75             0x2000  /* 75 baud */
03588    #define B110            0x3000  /* 110 baud */
03589    #define B134            0x4000  /* 134.5 baud */
03590    #define B150            0x5000  /* 150 baud */
03591    #define B200            0x6000  /* 200 baud */
03592    #define B300            0x7000  /* 300 baud */
03593    #define B600            0x8000  /* 600 baud */
03594    #define B1200           0x9000  /* 1200 baud */
03595    #define B1800           0xA000  /* 1800 baud */
03596    #define B2400           0xB000  /* 2400 baud */
03597    #define B4800           0xC000  /* 4800 baud */
03598    #define B9600           0xD000  /* 9600 baud */
03599    #define B19200          0xE000  /* 19200 baud */
03600    #define B38400          0xF000  /* 38400 baud */
03601
03602    /* Optional actions for tcsetattr().  POSIX Sec. 7.2.1.2. */
03603    #define TCSANOW            1    /* changes take effect immediately */
03604    #define TCSADRAIN          2    /* changes take effect after output is done */
03605    #define TCSAFLUSH          3    /* wait for output to finish and flush input */
03606
03607    /* Queue_selector values for tcflush().  POSIX Sec. 7.2.2.2. */
03608    #define TCIFLUSH           1    /* flush accumulated input data */
03609    #define TCOFLUSH           2    /* flush accumulated output data */
```

```
03610   #define TCIOFLUSH          3    /* flush accumulated input and output data */
03611
03612   /* Action values for tcflow().  POSIX Sec. 7.2.2.2. */
03613   #define TCOOFF             1    /* suspend output */
03614   #define TCOON              2    /* restart suspended output */
03615   #define TCIOFF             3    /* transmit a STOP character on the line */
03616   #define TCION              4    /* transmit a START character on the line */
03617
03618
03619   /* Function Prototypes. */
03620   #ifndef _ANSI_H
03621   #include <ansi.h>
03622   #endif
03623
03624   _PROTOTYPE( int tcsendbreak, (int _fildes, int _duration)                    );
03625   _PROTOTYPE( int tcdrain, (int _filedes)                                      );
03626   _PROTOTYPE( int tcflush, (int _filedes, int _queue_selector)                 );
03627   _PROTOTYPE( int tcflow, (int _filedes, int _action)                          );
03628   _PROTOTYPE( speed_t cfgetispeed, (const struct termios *_termios_p)          );
03629   _PROTOTYPE( speed_t cfgetospeed, (const struct termios *_termios_p)          );
03630   _PROTOTYPE( int cfsetispeed, (struct termios *_termios_p, speed_t _speed)    );
03631   _PROTOTYPE( int cfsetospeed, (struct termios *_termios_p, speed_t _speed)    );
03632   _PROTOTYPE( int tcgetattr, (int _filedes, struct termios *_termios_p)        );
03633   _PROTOTYPE( int tcsetattr, \
03634           (int _filedes, int _opt_actions, const struct termios *_termios_p)   );
03635
03636   #define cfgetispeed(termios_p)          ((termios_p)->c_ispeed)
03637   #define cfgetospeed(termios_p)          ((termios_p)->c_ospeed)
03638   #define cfsetispeed(termios_p, speed)   ((termios_p)->c_ispeed = (speed), 0)
03639   #define cfsetospeed(termios_p, speed)   ((termios_p)->c_ospeed = (speed), 0)
03640
03641   #ifdef _MINIX
03642   /* Here are the local extensions to the POSIX standard for Minix. Posix
03643    * conforming programs are not able to access these, and therefore they are
03644    * only defined when a Minix program is compiled.
03645    */
03646
03647   /* Extensions to the termios c_iflag bit map.  */
03648   #define IXANY           0x0800  /* allow any key to continue ouptut */
03649
03650   /* Extensions to the termios c_oflag bit map. They are only active iff
03651    * OPOST is enabled. */
03652   #define ONLCR           0x0002  /* Map NL to CR-NL on output */
03653   #define XTABS           0x0004  /* Expand tabs to spaces */
03654   #define ONOEOT          0x0008  /* discard EOT's (^D) on output) */
03655
03656   /* Extensions to the termios c_lflag bit map.  */
03657   #define LFLUSHO         0x0200  /* Flush output. */
03658
03659   /* Extensions to the c_cc array. */
03660   #define VREPRINT          11    /* cc_c[VREPRINT] (^R) */
03661   #define VLNEXT            12    /* cc_c[VLNEXT] (^V) */
03662   #define VDISCARD          13    /* cc_c[VDISCARD] (^O) */
03663
03664   /* Extensions to baud rate settings. */
03665   #define B57600          0x0100  /* 57600 baud */
03666   #define B115200         0x0200  /* 115200 baud */
03667
03668   /* These are the default settings used by the kernel and by 'stty sane' */
03669
```

```
03670    #define TCTRL_DEF       (CREAD | CS8 | HUPCL)
03671    #define TINPUT_DEF      (BRKINT | ICRNL | IXON | IXANY)
03672    #define TOUTPUT_DEF     (OPOST | ONLCR)
03673    #define TLOCAL_DEF      (ISIG | IEXTEN | ICANON | ECHO | ECHOE)
03674    #define TSPEED_DEF      B9600
03675
03676    #define TEOF_DEF        '\4'    /* ^D */
03677    #define TEOL_DEF        _POSIX_VDISABLE
03678    #define TERASE_DEF      '\10'   /* ^H */
03679    #define TINTR_DEF       '\177'  /* ^? */
03680    #define TKILL_DEF       '\25'   /* ^U */
03681    #define TMIN_DEF        1
03682    #define TQUIT_DEF       '\34'   /* ^\ */
03683    #define TSTART_DEF      '\21'   /* ^Q */
03684    #define TSTOP_DEF       '\23'   /* ^S */
03685    #define TSUSP_DEF       '\32'   /* ^Z */
03686    #define TTIME_DEF       0
03687    #define TREPRINT_DEF    '\22'   /* ^R */
03688    #define TLNEXT_DEF      '\26'   /* ^V */
03689    #define TDISCARD_DEF    '\17'   /* ^O */
03690
03691    /* Window size. This information is stored in the TTY driver but not used.
03692     * This can be used for screen based applications in a window environment.
03693     * The ioctls TIOCGWINSZ and TIOCSWINSZ can be used to get and set this
03694     * information.
03695     */
03696
03697    struct winsize
03698    {
03699            unsigned short  ws_row;         /* rows, in characters */
03700            unsigned short  ws_col;         /* columns, in characters */
03701            unsigned short  ws_xpixel;      /* horizontal size, pixels */
03702            unsigned short  ws_ypixel;      /* vertical size, pixels */
03703    };
03704    #endif /* _MINIX */
03705
03706    #endif /* _TERMIOS_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/time.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

03800    /* The <time.h> header is used by the procedures that deal with time.
03801     * Handling time is surprisingly complicated, what with GMT, local time
03802     * and other factors.  Although the Bishop of Ussher (1581-1656) once
03803     * calculated that based on the Bible, the world began on 12 Oct. 4004 BC
03804     * at 9 o'clock in the morning, in the UNIX world time begins at midnight,
03805     * 1 Jan. 1970 GMT.  Before that, all was NULL and (void).
03806     */
03807
03808    #ifndef _TIME_H
03809    #define _TIME_H
03810
03811    #define CLOCKS_PER_SEC    60    /* MINIX always uses 60 Hz, even in Europe */
03812
03813    #ifdef _POSIX_SOURCE
03814    #define CLK_TCK CLOCKS_PER_SEC  /* obsolescent mame for CLOCKS_PER_SEC */
```

```
03815   #endif
03816
03817   #define NULL    ((void *)0)
03818
03819   #ifndef _SIZE_T
03820   #define _SIZE_T
03821   typedef unsigned int size_t;
03822   #endif
03823
03824   #ifndef _TIME_T
03825   #define _TIME_T
03826   typedef long time_t;            /* time in sec since 1 Jan 1970 0000 GMT */
03827   #endif
03828
03829   #ifndef _CLOCK_T
03830   #define _CLOCK_T
03831   typedef long clock_t;           /* time in ticks since process started */
03832   #endif
03833
03834   struct tm {
03835     int tm_sec;                   /* seconds after the minute [0, 59] */
03836     int tm_min;                   /* minutes after the hour [0, 59] */
03837     int tm_hour;                  /* hours since midnight [0, 23] */
03838     int tm_mday;                  /* day of the month [1, 31] */
03839     int tm_mon;                   /* months since January [0, 11] */
03840     int tm_year;                  /* years since 1900 */
03841     int tm_wday;                  /* days since Sunday [0, 6] */
03842     int tm_yday;                  /* days since January 1 [0, 365] */
03843     int tm_isdst;                 /* Daylight Saving Time flag */
03844   };
03845
03846   extern char *tzname[];
03847
03848   /* Function Prototypes. */
03849   #ifndef _ANSI_H
03850   #include <ansi.h>
03851   #endif
03852
03853   _PROTOTYPE( clock_t clock, (void)                                       );
03854   _PROTOTYPE( double difftime, (time_t _time1, time_t _time0)             );
03855   _PROTOTYPE( time_t mktime, (struct tm *_timeptr)                        );
03856   _PROTOTYPE( time_t time, (time_t *_timeptr)                             );
03857   _PROTOTYPE( char *asctime, (const struct tm *_timeptr)                  );
03858   _PROTOTYPE( char *ctime, (const time_t *_timer)                 );
03859   _PROTOTYPE( struct tm *gmtime, (const time_t *_timer)                   );
03860   _PROTOTYPE( struct tm *localtime, (const time_t *_timer)                );
03861   _PROTOTYPE( size_t strftime, (char *_s, size_t _max, const char *_fmt,
03862                                 const struct tm *_timep)                  );
03863
03864   #ifdef _POSIX_SOURCE
03865   _PROTOTYPE( void tzset, (void)                                          );
03866   #endif
03867
03868   #ifdef _MINIX
03869   _PROTOTYPE( int stime, (time_t *_top)                                   );
03870   #endif
03871
03872   #endif /* _TIME_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/tools.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

03900    /* Constants describing the disk */
03901    #define SECTOR_SIZE     512
03902    #define SECTOR_SHIFT    9
03903    #define RATIO           (BLOCK_SIZE / SECTOR_SIZE)
03904    #define HRATIO          (SECTOR_SIZE / HCLICK_SIZE)
03905    #define PARAMSEC        1       /* sector containing boot parameters */
03906    #define DSKBASE         0x1E    /* floppy disk parameter vector */
03907    #define DSKPARSIZE      11      /* there are this many bytes of parameters */
03908    #define ESC             '\33'   /* escape key */
03909    #define HEADERSEG       0x0060  /* place for an array of struct exec's */
03910    #define MINIXSEG        0x0080  /* MINIX loaded here (rounded up to a click) */
03911    #define BOOTSEG         0x07C0  /* bootstraps are loaded here */
03912    #define SIGNATURE       0xAA55  /* proper bootstraps have this signature */
03913    #define SIGNATPOS       510     /* offset within bootblock */
03914    #define FREESEG         0x0800  /* Memory from FREESEG to cseg is free */
03915    #define MSEC_PER_TICK   55      /* 18.2 ticks per second */
03916
03917    /* Scan codes for four different keyboards (from kernel/keyboard.c) */
03918    #define DUTCH_EXT_SCAN    32    /* 'd' */
03919    #define OLIVETTI_SCAN     12    /* '=' key on olivetti */
03920    #define STANDARD_SCAN     13    /* '=' key on IBM */
03921    #define US_EXT_SCAN       22    /* 'u' */
03922
03923    /* Other */
03924    #define ROOT_INO ((ino_t) 1)    /* Inode nr of root dir. */
03925    #define IM_NAME_MAX       63
03926
03927    /* Variables */
03928    #ifndef EXTERN
03929    #define EXTERN extern
03930    #endif
03931
03932    typedef struct vector {
03933      u16_t offset;
03934      u16_t segment;
03935    } vector;
03936
03937    struct image_header {
03938      char name[IM_NAME_MAX + 1];   /* Null terminated. */
03939      struct exec process;
03940    };
03941
03942    EXTERN vector rem_part;         /* boot partition table entry */
03943    EXTERN u16_t cseg, dseg;        /* code and data segment of the boot program */
03944    EXTERN u32_t runsize;           /* size of this program */
03945    EXTERN u16_t device;            /* drive being booted from */
03946    EXTERN u16_t heads, sectors;    /* the drive's number of heads and sectors */
03947    extern u16_t eqscancode;        /* Set by peek/getch() if they see a '=' */
03948
03949    /* Sticky attributes */
03950    #define E_SPECIAL       0x01    /* These are known to the program */
03951    #define E_DEV           0x02    /* The value is a device name */
03952    #define E_RESERVED      0x04    /* May not be set by user, e.g. scancode */
03953    #define E_STICKY        0x07    /* Don't go once set */
03954
```

```
03955   /* Volatile attributes */
03956   #define E_VAR           0x08    /* Variable */
03957   #define E_FUNCTION      0x10    /* Function definition */
03958
03959   typedef struct environment {
03960     struct environment *next;
03961     char flags;
03962     char *name;                   /* name = value */
03963     char *arg;                    /* name(arg) {value} */
03964     char *value;
03965     char *defval;                 /* Safehouse for default values */
03966   } environment;
03967
03968   /* External variables */
03969   EXTERN environment *env;        /* Lists the environment */
03970   EXTERN int fsok;                /* True if the boot device contains an FS */
03971   EXTERN u32_t lowsec;            /* Offset to the file system on the boot dev */
03972
03973   /* Prototypes */
03974   _PROTOTYPE( off_t r_super, (void));
03975   _PROTOTYPE( void r_stat, (Ino_t _inum, struct stat *_stp ));
03976   _PROTOTYPE( ino_t r_readdir, (char *_name ));
03977   _PROTOTYPE( off_t r_vir2abs, (off_t _virblk ));
03978   _PROTOTYPE( ino_t r_lookup, (Ino_t _cwd, char *_path ));
03979
03980   #ifdef _MONHEAD
03981   _PROTOTYPE( void readerr, (off_t _sec, int _err ));
03982   _PROTOTYPE( int numprefix, (char *_s, char **_ps ));
03983   _PROTOTYPE( int numeric, (char *_s ));
03984   _PROTOTYPE( dev_t name2dev, (char *_name ));
03985   _PROTOTYPE( int delay, (char *_msec ));
03986   _PROTOTYPE( char *unix_err, (int _err ));
03987   _PROTOTYPE( void init_cache, (void));
03988   _PROTOTYPE( void invalidate_cache, (void));
03989   _PROTOTYPE( char *b_value, (char *_name ));
03990   _PROTOTYPE( void raw_copy, (int _doff, int _dseg, int _soff, int _sseg,
03991                                                 int _count));
03992   _PROTOTYPE( void raw_clear, (int _off, int _seg, int _count));
03993   _PROTOTYPE( void bootstrap, (int _device, int _partoff, int _partseg));
03994
03995   _PROTOTYPE( long a2l, (char *_a ));
03996   _PROTOTYPE( char *ul2a, (u32_t _n ));
03997   _PROTOTYPE( char *u2a, (int _n1 ));
03998
03999   /* Functions defined in monhead.s and usable by other files. */
04000   _PROTOTYPE( void reset_video, (int color));
04001   _PROTOTYPE( int dev_geometry, (void));
04002   _PROTOTYPE( u16_t get_ext_memsize, (void));
04003   _PROTOTYPE( u16_t get_low_memsize, (void));
04004   _PROTOTYPE( u16_t get_processor, (void));
04005   _PROTOTYPE( u32_t get_tick, (void));
04006   _PROTOTYPE( u16_t get_video, (void));
04007   _PROTOTYPE( u16_t get_word, (int _off, int _seg));
04008   _PROTOTYPE( int getchar, (void));
04009   _PROTOTYPE( void minix, (void));
04010   _PROTOTYPE( void minix86, (int _kcs, int _kds, char *_bpar, int _psize));
04011   _PROTOTYPE( void minix386, (int _kcs, int _kds, char *_bpar, int _psize));
04012   _PROTOTYPE( int peekchar, (void));
04013   _PROTOTYPE( void put_word, (int _off, int _seg, int _word));
04014   _PROTOTYPE( int putchar, (char _c));
```

```
04015    _PROTOTYPE( int readsectors, (int _off, int _seg, off_t _adr, int _ct));
04016    _PROTOTYPE( void reboot, (void));
04017    _PROTOTYPE( void relocate, (void));
04018    _PROTOTYPE( int writesectors, (int _off, int _seg, off_t _adr, int _ct));
04019    #endif
04020
04021
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/ttyent.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

04100    /* <ttyent.h> is used by getttyent(3).                  Author: Kees J. Bot
04101     *                                                             28 Oct 1995
04102     */
04103    #ifndef _TTYENT_H
04104    #define _TTYENT_H
04105
04106    #ifndef _ANSI_H
04107    #include <ansi.h>
04108    #endif
04109
04110    struct ttyent {
04111            char    *ty_name;       /* Name of the terminal device. */
04112            char    *ty_type;       /* Terminal type name (termcap(3)). */
04113            char    **ty_getty;     /* Program to run, normally getty. */
04114            char    **ty_init;      /* Initialization command, normally stty. */
04115    };
04116
04117    _PROTOTYPE( struct ttyent *getttyent, (void)                            );
04118    _PROTOTYPE( struct ttyent *getttynam, (const char *_name)               );
04119    _PROTOTYPE( int setttyent, (void)                                       );
04120    _PROTOTYPE( void endttyent, (void)                                      );
04121
04122    #endif /* _TTYENT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/unistd.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

04200    /* The <unistd.h> header contains a few miscellaneous manifest constants. */
04201
04202    #ifndef _UNISTD_H
04203    #define _UNISTD_H
04204
04205    /* POSIX requires size_t and ssize_t in <unistd.h> and elsewhere. */
04206    #ifndef _SIZE_T
04207    #define _SIZE_T
04208    typedef unsigned int size_t;
04209    #endif
04210
04211    #ifndef _SSIZE_T
04212    #define _SSIZE_T
04213    typedef int ssize_t;
04214    #endif
```

```
04215
04216   /* Values used by access().  POSIX Table 2-8. */
04217   #define F_OK               0    /* test if file exists */
04218   #define X_OK               1    /* test if file is executable */
04219   #define W_OK               2    /* test if file is writable */
04220   #define R_OK               4    /* test if file is readable */
04221
04222   /* Values used for whence in lseek(fd, offset, whence).  POSIX Table 2-9. */
04223   #define SEEK_SET           0    /* offset is absolute  */
04224   #define SEEK_CUR           1    /* offset is relative to current position */
04225   #define SEEK_END           2    /* offset is relative to end of file */
04226
04227   /* This value is required by POSIX Table 2-10. */
04228   #define _POSIX_VERSION 199009L  /* which standard is being conformed to */
04229
04230   /* These three definitions are required by POSIX Sec. 8.2.1.2. */
04231   #define STDIN_FILENO       0    /* file descriptor for stdin */
04232   #define STDOUT_FILENO      1    /* file descriptor for stdout */
04233   #define STDERR_FILENO      2    /* file descriptor for stderr */
04234
04235   #ifdef _MINIX
04236   /* How to exit the system. */
04237   #define RBT_HALT           0
04238   #define RBT_REBOOT         1
04239   #define RBT_PANIC          2    /* for servers */
04240   #define RBT_MONITOR        3    /* let the monitor do this */
04241   #define RBT_RESET          4    /* hard reset the system */
04242   #endif
04243
04244   /* NULL must be defined in <unistd.h> according to POSIX Sec. 2.7.1. */
04245   #define NULL    ((void *)0)
04246
04247   /* The following relate to configurable system variables. POSIX Table 4-2. */
04248   #define _SC_ARG_MAX             1
04249   #define _SC_CHILD_MAX           2
04250   #define _SC_CLOCKS_PER_SEC      3
04251   #define _SC_CLK_TCK             3
04252   #define _SC_NGROUPS_MAX         4
04253   #define _SC_OPEN_MAX            5
04254   #define _SC_JOB_CONTROL         6
04255   #define _SC_SAVED_IDS           7
04256   #define _SC_VERSION             8
04257   #define _SC_STREAM_MAX          9
04258   #define _SC_TZNAME_MAX         10
04259
04260   /* The following relate to configurable pathname variables. POSIX Table 5-2. */
04261   #define _PC_LINK_MAX            1       /* link count */
04262   #define _PC_MAX_CANON           2       /* size of the canonical input queue */
04263   #define _PC_MAX_INPUT           3       /* type-ahead buffer size */
04264   #define _PC_NAME_MAX            4       /* file name size */
04265   #define _PC_PATH_MAX            5       /* pathname size */
04266   #define _PC_PIPE_BUF            6       /* pipe size */
04267   #define _PC_NO_TRUNC            7       /* treatment of long name components */
04268   #define _PC_VDISABLE            8       /* tty disable */
04269   #define _PC_CHOWN_RESTRICTED    9       /* chown restricted or not */
04270
04271   /* POSIX defines several options that may be implemented or not, at the
04272    * implementer's whim.  This implementer has made the following choices:
04273    *
04274    * _POSIX_JOB_CONTROL       not defined:        no job control
```

```
04275     * _POSIX_SAVED_IDS         not defined:        no saved uid/gid
04276     * _POSIX_NO_TRUNC          defined as -1:      long path names are truncated
04277     * _POSIX_CHOWN_RESTRICTED  defined:            you can't give away files
04278     * _POSIX_VDISABLE          defined:            tty functions can be disabled
04279     */
04280    #define _POSIX_NO_TRUNC       (-1)
04281    #define _POSIX_CHOWN_RESTRICTED  1
04282
04283    /* Function Prototypes. */
04284    #ifndef _ANSI_H
04285    #include <ansi.h>
04286    #endif
04287
04288    _PROTOTYPE( void _exit, (int _status)                                    );
04289    _PROTOTYPE( int access, (const char *_path, int _amode)                  );
04290    _PROTOTYPE( unsigned int alarm, (unsigned int _seconds)                  );
04291    _PROTOTYPE( int chdir, (const char *_path)                               );
04292    _PROTOTYPE( int chown, (const char *_path, Uid_t _owner, Gid_t _group)   );
04293    _PROTOTYPE( int close, (int _fd)                                         );
04294    _PROTOTYPE( char *ctermid, (char *_s)                                    );
04295    _PROTOTYPE( char *cuserid, (char *_s)                                    );
04296    _PROTOTYPE( int dup, (int _fd)                                           );
04297    _PROTOTYPE( int dup2, (int _fd, int _fd2)                                );
04298    _PROTOTYPE( int execl, (const char *_path, const char *_arg, ...)        );
04299    _PROTOTYPE( int execle, (const char *_path, const char *_arg, ...)       );
04300    _PROTOTYPE( int execlp, (const char *_file, const char *arg, ...)        );
04301    _PROTOTYPE( int execv, (const char *_path, char *const _argv[])          );
04302    _PROTOTYPE( int execve, (const char *_path, char *const _argv[],
04303                                                 char *const _envp[])        );
04304    _PROTOTYPE( int execvp, (const char *_file, char *const _argv[])         );
04305    _PROTOTYPE( pid_t fork, (void)                                           );
04306    _PROTOTYPE( long fpathconf, (int _fd, int _name)                         );
04307    _PROTOTYPE( char *getcwd, (char *_buf, size_t _size)                     );
04308    _PROTOTYPE( gid_t getegid, (void)                                        );
04309    _PROTOTYPE( uid_t geteuid, (void)                                        );
04310    _PROTOTYPE( gid_t getgid, (void)                                         );
04311    _PROTOTYPE( int getgroups, (int _gidsetsize, gid_t _grouplist[])         );
04312    _PROTOTYPE( char *getlogin, (void)                                       );
04313    _PROTOTYPE( pid_t getpgrp, (void)                                        );
04314    _PROTOTYPE( pid_t getpid, (void)                                         );
04315    _PROTOTYPE( pid_t getppid, (void)                                        );
04316    _PROTOTYPE( uid_t getuid, (void)                                         );
04317    _PROTOTYPE( int isatty, (int _fd)                                        );
04318    _PROTOTYPE( int link, (const char *_existing, const char *_new)          );
04319    _PROTOTYPE( off_t lseek, (int _fd, off_t _offset, int _whence)           );
04320    _PROTOTYPE( long pathconf, (const char *_path, int _name)                );
04321    _PROTOTYPE( int pause, (void)                                            );
04322    _PROTOTYPE( int pipe, (int _fildes[2])                                   );
04323    _PROTOTYPE( ssize_t read, (int _fd, void *_buf, size_t _n)               );
04324    _PROTOTYPE( int rmdir, (const char *_path)                               );
04325    _PROTOTYPE( int setgid, (Gid_t _gid)                                     );
04326    _PROTOTYPE( int setpgid, (pid_t _pid, pid_t _pgid)                       );
04327    _PROTOTYPE( pid_t setsid, (void)                                         );
04328    _PROTOTYPE( int setuid, (Uid_t _uid)                                     );
04329    _PROTOTYPE( unsigned int sleep, (unsigned int _seconds)                  );
04330    _PROTOTYPE( long sysconf, (int _name)                                    );
04331    _PROTOTYPE( pid_t tcgetpgrp, (int _fd)                                   );
04332    _PROTOTYPE( int tcsetpgrp, (int _fd, pid_t _pgrp_id)                     );
04333    _PROTOTYPE( char *ttyname, (int _fd)                                     );
04334    _PROTOTYPE( int unlink, (const char *_path)                              );
```

```
04335    _PROTOTYPE( ssize_t write, (int _fd, const void *_buf, size_t _n)       );
04336
04337    #ifdef _MINIX
04338    _PROTOTYPE( int brk, (char *_addr)                                      );
04339    _PROTOTYPE( int chroot, (const char *_name)                             );
04340    _PROTOTYPE( int mknod, (const char *_name, Mode_t _mode, Dev_t _addr)   );
04341    _PROTOTYPE( int mknod4, (const char *_name, Mode_t _mode, Dev_t _addr,
04342                long _size)                                                 );
04343    _PROTOTYPE( char *mktemp, (char *_template)                             );
04344    _PROTOTYPE( int mount, (char *_spec, char *_name, int _flag)            );
04345    _PROTOTYPE( long ptrace, (int _req, pid_t _pid, long _addr, long _data) );
04346    _PROTOTYPE( char *sbrk, (int _incr)                                     );
04347    _PROTOTYPE( int sync, (void)                                            );
04348    _PROTOTYPE( int umount, (const char *_name)                             );
04349    _PROTOTYPE( int reboot, (int _how, ...)                                 );
04350    _PROTOTYPE( int gethostname, (char *_hostname, size_t _len)             );
04351    _PROTOTYPE( int getdomainname, (char *_domain, size_t _len)             );
04352    _PROTOTYPE( int ttyslot, (void)                                         );
04353    _PROTOTYPE( int fttyslot, (int _fd)                                     );
04354    _PROTOTYPE( char *crypt, (const char *_key, const char *_salt)          );
04355    #endif
04356
04357    #endif /* _UNISTD_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/utime.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

04400    /* The <utime.h> header is used for the utime() system call. */
04401
04402    #ifndef _UTIME_H
04403    #define _UTIME_H
04404
04405    struct utimbuf {
04406      time_t actime;                /* access time */
04407      time_t modtime;               /* modification time */
04408    };
04409
04410
04411    /* Function Prototypes. */
04412    #ifndef _ANSI_H
04413    #include <ansi.h>
04414    #endif
04415
04416    _PROTOTYPE( int utime, (const char *_path, const struct utimbuf *_times)     );
04417
04418    #endif /* _UTIME_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/utmp.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

04500    /* The <utmp.h> header is used by init, login, who, etc. */
04501
04502    #ifndef _UTMP_H
04503    #define _UTMP_H
04504
04505    #define WTMP  "/usr/adm/wtmp"   /* the login history file */
04506    #define BTMP  "/usr/adm/btmp"   /* the bad-login history file */
04507    #define UTMP  "/etc/utmp"       /* the user accouting file */
04508
04509
04510    struct utmp {
04511      char ut_user[8];              /* user name */
04512      char ut_id[4];                /* /etc/inittab ID */
04513      char ut_line[12];             /* terminal name */
04514      char ut_host[16];             /* host name, when remote */
04515      short ut_pid;                 /* process id */
04516      short int ut_type;            /* type of entry */
04517      long ut_time;                 /* login/logout time */
04518    };
04519
04520    #define ut_name ut_user         /* for compatibility with other systems */
04521
04522
04523    /* Definitions for ut_type. */
04524    #define RUN_LVL            1    /* this is a RUN_LEVEL record */
04525    #define BOOT_TIME          2    /* this is a REBOOT record */
04526    #define INIT_PROCESS       5    /* this process was spawned by INIT */
04527    #define LOGIN_PROCESS      6    /* this is a 'getty' process waiting */
04528    #define USER_PROCESS       7    /* any other user process */
04529    #define DEAD_PROCESS       8    /* this process has died (wtmp only) */
04530
04531    #endif /* _UTMP_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/sys/asynchio.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

04600    /*      asynchio.h - Asynchronous I/O                   Author: Kees J. Bot
04601     *                                                              26 Jan 1995
04602     * This is just a fake async I/O library to be used for programs
04603     * written for Minix-vmd that must also run under standard Minix.
04604     * This limits the number of ugly #ifdefs somewhat.  The programs must
04605     * be restricted to performing just one service, of course.
04606     */
04607    #ifndef _SYS__ASYNCHIO_H
04608    #define _SYS__ASYNCHIO_H
04609
04610    typedef int asynchio_t;
04611
04612    #define ASYN_NONBLOCK   0x01
04613
04614    struct timeval { long tv_sec, tv_usec; };
```

```
04615
04616    #define EINPROGRESS     EINTR
04617
04618    void asyn_init(asynchio_t *_asyn);
04619    ssize_t asyn_read(asynchio_t *_asyn, int _fd, void *_buf, size_t _len);
04620    ssize_t asyn_write(asynchio_t *_asyn, int _fd, const void *_buf, size_t _len);
04621    int asyn_ioctl(asynchio_t *_asyn, int _fd, unsigned long _request, void *_data);
04622    int asyn_wait(asynchio_t *_asyn, int _flags, struct timeval *to);
04623    int asyn_synch(asynchio_t *_asyn, int _fd);
04624    int asyn_close(asynchio_t *_asyn, int _fd);
04625
04626    #endif /* _SYS__ASYNCHIO_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/sys/dir.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

04700    /* The <dir.h> header gives the layout of a directory. */
04701
04702    #ifndef _DIR_H
04703    #define _DIR_H
04704
04705    #define DIRBLKSIZ       512     /* size of directory block */
04706
04707    #ifndef DIRSIZ
04708    #define DIRSIZ  14
04709    #endif
04710
04711    struct direct {
04712      ino_t d_ino;
04713      char d_name[DIRSIZ];
04714    };
04715
04716    #endif /* _DIR_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/sys/ioctl.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

04800    /* The ioctl.h header declares device controlling operations. */
04801
04802    #ifndef _IOCTL_H
04803    #define _IOCTL_H
04804
04805    #if _EM_WSIZE >= 4
04806    /* Ioctls have the command encoded in the low-order word, and the size
04807     * of the parameter in the high-order word. The 3 high bits of the high-
04808     * order word are used to encode the in/out/void status of the parameter.
04809     */
04810
04811    #define _IOCPARM_MASK   0x1FFF
04812    #define _IOC_VOID       0x20000000
04813    #define _IOCTYPE_MASK   0xFFFF
04814    #define _IOC_IN         0x40000000
```

```
04815    #define _IOC_OUT        0x80000000
04816    #define _IOC_INOUT      (_IOC_IN | _IOC_OUT)
04817
04818    #define _IO(x,y)        ((x << 8) | y | _IOC_VOID)
04819    #define _IOR(x,y,t)     ((x << 8) | y | ((sizeof(t) & _IOCPARM_MASK) << 16) |\
04820                                 _IOC_OUT)
04821    #define _IOW(x,y,t)     ((x << 8) | y | ((sizeof(t) & _IOCPARM_MASK) << 16) |\
04822                                 _IOC_IN)
04823    #define _IORW(x,y,t)    ((x << 8) | y | ((sizeof(t) & _IOCPARM_MASK) << 16) |\
04824                                 _IOC_INOUT)
04825    #else
04826    /* No fancy encoding on a 16-bit machine. */
04827
04828    #define _IO(x,y)        ((x << 8) | y)
04829    #define _IOR(x,y,t)     _IO(x,y)
04830    #define _IOW(x,y,t)     _IO(x,y)
04831    #define _IORW(x,y,t)    _IO(x,y)
04832    #endif
04833
04834
04835    /* Terminal ioctls. */
04836    #define TCGETS          _IOR('T',  8, struct termios) /* tcgetattr */
04837    #define TCSETS          _IOW('T',  9, struct termios) /* tcsetattr, TCSANOW */
04838    #define TCSETSW         _IOW('T', 10, struct termios) /* tcsetattr, TCSADRAIN */
04839    #define TCSETSF         _IOW('T', 11, struct termios) /* tcsetattr, TCSAFLUSH */
04840    #define TCSBRK          _IOW('T', 12, int)            /* tcsendbreak */
04841    #define TCDRAIN         _IO ('T', 13)                 /* tcdrain */
04842    #define TCFLOW          _IOW('T', 14, int)            /* tcflow */
04843    #define TCFLSH          _IOW('T', 15, int)            /* tcflush */
04844    #define TIOCGWINSZ      _IOR('T', 16, struct winsize)
04845    #define TIOCSWINSZ      _IOW('T', 17, struct winsize)
04846    #define TIOCGPGRP       _IOW('T', 18, int)
04847    #define TIOCSPGRP       _IOW('T', 19, int)
04848    #define TIOCSFON        _IOW('T', 20, u8_t [8192])
04849
04850    #define TIOCGETP        _IOR('t',  1, struct sgttyb)
04851    #define TIOCSETP        _IOW('t',  2, struct sgttyb)
04852    #define TIOCGETC        _IOR('t',  3, struct tchars)
04853    #define TIOCSETC        _IOW('t',  4, struct tchars)
04854
04855
04856    /* Network ioctls. */
04857    #define NWIOSETHOPT     _IOW('n', 16, struct nwio_ethopt)
04858    #define NWIOGETHOPT     _IOR('n', 17, struct nwio_ethopt)
04859    #define NWIOGETHSTAT    _IOR('n', 18, struct nwio_ethstat)
04860
04861    #define NWIOSIPCONF     _IOW('n', 32, struct nwio_ipconf)
04862    #define NWIOGIPCONF     _IOR('n', 33, struct nwio_ipconf)
04863    #define NWIOSIPOPT      _IOW('n', 34, struct nwio_ipopt)
04864    #define NWIOGIPOPT      _IOR('n', 35, struct nwio_ipopt)
04865
04866    #define NWIOIPGROUTE    _IORW('n', 40, struct nwio_route)
04867    #define NWIOIPSROUTE    _IOW ('n', 41, struct nwio_route)
04868    #define NWIOIPDROUTE    _IOW ('n', 42, struct nwio_route)
04869
04870    #define NWIOSTCPCONF    _IOW('n', 48, struct nwio_tcpconf)
04871    #define NWIOGTCPCONF    _IOR('n', 49, struct nwio_tcpconf)
04872    #define NWIOTCPCONN     _IOW('n', 50, struct nwio_tcpcl)
04873    #define NWIOTCPLISTEN   _IOW('n', 51, struct nwio_tcpcl)
04874    #define NWIOTCPATTACH   _IOW('n', 52, struct nwio_tcpatt)
```

```
04875    #define NWIOTCPSHUTDOWN _IO ('n', 53)
04876    #define NWIOSTCPOPT     _IOW('n', 54, struct nwio_tcpopt)
04877    #define NWIOGTCPOPT     _IOR('n', 55, struct nwio_tcpopt)
04878
04879    #define NWIOSUDPOPT     _IOW('n', 64, struct nwio_udpopt)
04880    #define NWIOGUDPOPT     _IOR('n', 65, struct nwio_udpopt)
04881
04882    /* Disk ioctls. */
04883    #define DIOCEJECT       _IO ('d', 5)
04884    #define DIOCSETP        _IOW('d', 6, struct partition)
04885    #define DIOCGETP        _IOR('d', 7, struct partition)
04886
04887    /* Keyboard ioctls. */
04888    #define KIOCSMAP        _IOW('k', 3, keymap_t)
04889
04890    /* Memory ioctls. */
04891    #define MIOCRAMSIZE     _IOW('m', 3, u32_t)     /* Size of the ramdisk */
04892    #define MIOCSPSINFO     _IOW('m', 4, void *)
04893    #define MIOCGPSINFO     _IOR('m', 5, struct psinfo)
04894
04895    /* Magnetic tape ioctls. */
04896    #define MTIOCTOP        _IOW('M', 1, struct mtop)
04897    #define MTIOCGET        _IOR('M', 2, struct mtget)
04898
04899    /* SCSI command. */
04900    #define SCIOCCMD        _IOW('S', 1, struct scsicmd)
04901
04902    /* CD-ROM ioctls. */
04903    #define CDIOPLAYTI      _IOR('c', 1, struct cd_play_track)
04904    #define CDIOPLAYMSS     _IOR('c', 2, struct cd_play_mss)
04905    #define CDIOREADTOCHDR  _IOW('c', 3, struct cd_toc_entry)
04906    #define CDIOREADTOC     _IOW('c', 4, struct cd_toc_entry)
04907    #define CDIOREADSUBCH   _IOW('c', 5, struct cd_toc_entry)
04908    #define CDIOSTOP        _IO ('c', 10)
04909    #define CDIOPAUSE       _IO ('c', 11)
04910    #define CDIORESUME      _IO ('c', 12)
04911    #define CDIOEJECT       DIOCEJECT
04912
04913    /* Soundcard DSP ioctls. */
04914    #define DSPIORATE       _IOR('s', 1, unsigned int)
04915    #define DSPIOSTEREO     _IOR('s', 2, unsigned int)
04916    #define DSPIOSIZE       _IOR('s', 3, unsigned int)
04917    #define DSPIOBITS       _IOR('s', 4, unsigned int)
04918    #define DSPIOSIGN       _IOR('s', 5, unsigned int)
04919    #define DSPIOMAX        _IOW('s', 6, unsigned int)
04920    #define DSPIORESET      _IO ('s', 7)
04921
04922    /* Soundcard mixer ioctls. */
04923    #define MIXIOGETVOLUME          _IORW('s', 10, struct volume_level)
04924    #define MIXIOGETINPUTLEFT       _IORW('s', 11, struct inout_ctrl)
04925    #define MIXIOGETINPUTRIGHT      _IORW('s', 12, struct inout_ctrl)
04926    #define MIXIOGETOUTPUT          _IORW('s', 13, struct inout_ctrl)
04927    #define MIXIOSETVOLUME          _IORW('s', 20, struct volume_level)
04928    #define MIXIOSETINPUTLEFT       _IORW('s', 21, struct inout_ctrl)
04929    #define MIXIOSETINPUTRIGHT      _IORW('s', 22, struct inout_ctrl)
04930    #define MIXIOSETOUTPUT          _IORW('s', 23, struct inout_ctrl)
04931
04932    #ifndef _ANSI
04933    #include <ansi.h>
04934    #endif
```

```
04935
04936    _PROTOTYPE( int ioctl, (int _fd, int _request, void *_data)             );
04937
04938    #endif /* _IOCTL_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/sys/mtio.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

05000    /* <sys/mtio.h> magnetic tape commands                 Author: Kees J. Bot
05001     */
05002
05003    #ifndef _SYS__MTIO_H
05004    #define _SYS__MTIO_H
05005
05006    /* Tape operations: ioctl(fd, MTIOCTOP, &struct mtop) */
05007
05008    struct mtop {
05009            short   mt_op;          /* Operation (MTWEOF, etc.) */
05010            int     mt_count;       /* Repeat count. */
05011    };
05012
05013    #define MTWEOF   0      /* Write End-Of-File Marker */
05014    #define MTFSF    1      /* Forward Space File mark */
05015    #define MTBSF    2      /* Backward Space File mark */
05016    #define MTFSR    3      /* Forward Space Record */
05017    #define MTBSR    4      /* Backward Space Record */
05018    #define MTREW    5      /* Rewind tape */
05019    #define MTOFFL   6      /* Rewind and take Offline */
05020    #define MTNOP    7      /* No-Operation, set status only */
05021    #define MTRETEN  8      /* Retension (completely wind and rewind) */
05022    #define MTERASE  9      /* Erase the tape and rewind */
05023    #define MTEOM   10      /* Position at End-Of-Media */
05024    #define MTMODE  11      /* Select tape density */
05025    #define MTBLKZ  12      /* Select tape block size */
05026
05027    /* Tape status: ioctl(fd, MTIOCGET, &struct mtget) */
05028
05029    struct mtget {
05030            short   mt_type;        /* Type of tape device. */
05031
05032            /* Device dependent "registers". */
05033            short   mt_dsreg;       /* Drive status register. */
05034            short   mt_erreg;       /* Error register. */
05035            short   dummy;          /* (alignment) */
05036
05037            /* Misc info. */
05038            off_t   mt_resid;       /* Residual count. */
05039            off_t   mt_fileno;      /* Current File Number. */
05040            off_t   mt_blkno;       /* Current Block Number within file. */
05041            off_t   mt_blksize;     /* Current block size. */
05042    };
05043
05044    #endif /* _SYS__MTIO_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/sys/ptrace.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

05100    /* <sys/ptrace.h>
05101     * definitions for ptrace(2)
05102     */
05103
05104    #ifndef _PTRACE_H
05105    #define _PTRACE_H
05106
05107    #define T_STOP         -1       /* stop the process */
05108    #define T_OK            0       /* enable tracing by parent for this process */
05109    #define T_GETINS        1       /* return value from instruction space */
05110    #define T_GETDATA       2       /* return value from data space */
05111    #define T_GETUSER       3       /* return value from user process table */
05112    #define T_SETINS        4       /* set value from instruction space */
05113    #define T_SETDATA       5       /* set value from data space */
05114    #define T_SETUSER       6       /* set value in user process table */
05115    #define T_RESUME        7       /* resume execution */
05116    #define T_EXIT          8       /* exit */
05117    #define T_STEP          9       /* set trace bit */
05118
05119    /* Function Prototypes. */
05120    #ifndef _ANSI_H
05121    #include <ansi.h>
05122    #endif
05123
05124    _PROTOTYPE( long ptrace, (int _req, pid_t _pid, long _addr, long _data) );
05125
05126    #endif /* _PTRACE_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/sys/sigcontext.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

05200    #ifndef _SIGCONTEXT_H
05201    #define _SIGCONTEXT_H
05202
05203    /* The sigcontext structure is used by the sigreturn(2) system call.
05204     * sigreturn() is seldom called by user programs, but it is used internally
05205     * by the signal catching mechanism.
05206     */
05207
05208    #ifndef _ANSI_H
05209    #include <ansi.h>
05210    #endif
05211
05212    #ifndef _CONFIG_H
05213    #include <minix/config.h>
05214    #endif
05215
05216    #if !defined(CHIP)
05217    #include "error, configuration is not known"
05218    #endif
05219
```

```
05220   /* The following structure should match the stackframe_s structure used
05221    * by the kernel's context switching code.  Floating point registers should
05222    * be added in a different struct.
05223    */
05224   #if (CHIP == INTEL)
05225   struct sigregs {
05226   #if _WORD_SIZE == 4
05227     short sr_gs;
05228     short sr_fs;
05229   #endif /* _WORD_SIZE == 4 */
05230     short sr_es;
05231     short sr_ds;
05232     int sr_di;
05233     int sr_si;
05234     int sr_bp;
05235     int sr_st;                    /* stack top -- used in kernel */
05236     int sr_bx;
05237     int sr_dx;
05238     int sr_cx;
05239     int sr_retreg;
05240     int sr_retadr;                /* return address to caller of save -- used
05241                                    * in kernel */
05242     int sr_pc;
05243     int sr_cs;
05244     int sr_psw;
05245     int sr_sp;
05246     int sr_ss;
05247   };
05248
05249   struct sigframe {               /* stack frame created for signalled process */
05250     _PROTOTYPE( void (*sf_retadr), (void) );
05251     int sf_signo;
05252     int sf_code;
05253     struct sigcontext *sf_scp;
05254     int sf_fp;
05255     _PROTOTYPE( void (*sf_retadr2), (void) );
05256     struct sigcontext *sf_scpcopy;
05257   };
05258
05259   #else
05260   #if (CHIP == M68000)
05261   struct sigregs {
05262     long sr_retreg;                       /* d0 */
05263     long sr_d1;
05264     long sr_d2;
05265     long sr_d3;
05266     long sr_d4;
05267     long sr_d5;
05268     long sr_d6;
05269     long sr_d7;
05270     long sr_a0;
05271     long sr_a1;
05272     long sr_a2;
05273     long sr_a3;
05274     long sr_a4;
05275     long sr_a5;
05276     long sr_a6;
05277     long sr_sp;                   /* also known as a7 */
05278     long sr_pc;
05279     short sr_psw;
```

```
05280     short sr_dummy;               /* make size multiple of 4 for system.c */
05281   };
05282   #else
05283   #include "error, CHIP is not supported"
05284   #endif
05285   #endif /* CHIP == INTEL */
05286
05287   struct sigcontext {
05288     int sc_flags;                 /* sigstack state to restore */
05289     long sc_mask;                 /* signal mask to restore */
05290     struct sigregs sc_regs;       /* register set to restore */
05291   };
05292
05293   #if (CHIP == INTEL)
05294   #if _WORD_SIZE == 4
05295   #define sc_gs sc_regs.sr_gs
05296   #define sc_fs sc_regs.sr_fs
05297   #endif /* _WORD_SIZE == 4 */
05298   #define sc_es sc_regs.sr_es
05299   #define sc_ds sc_regs.sr_ds
05300   #define sc_di sc_regs.sr_di
05301   #define sc_si sc_regs.sr_si
05302   #define sc_fp sc_regs.sr_bp
05303   #define sc_st sc_regs.sr_st             /* stack top -- used in kernel */
05304   #define sc_bx sc_regs.sr_bx
05305   #define sc_dx sc_regs.sr_dx
05306   #define sc_cx sc_regs.sr_cx
05307   #define sc_retreg sc_regs.sr_retreg
05308   #define sc_retadr sc_regs.sr_retadr     /* return address to caller of
05309                                             save -- used in kernel */
05310   #define sc_pc sc_regs.sr_pc
05311   #define sc_cs sc_regs.sr_cs
05312   #define sc_psw sc_regs.sr_psw
05313   #define sc_sp sc_regs.sr_sp
05314   #define sc_ss sc_regs.sr_ss
05315   #endif /* CHIP == INTEL */
05316
05317   #if (CHIP == M68000)
05318   #define sc_retreg sc_regs.sr_retreg
05319   #define sc_d1 sc_regs.sr_d1
05320   #define sc_d2 sc_regs.sr_d2
05321   #define sc_d3 sc_regs.sr_d3
05322   #define sc_d4 sc_regs.sr_d4
05323   #define sc_d5 sc_regs.sr_d5
05324   #define sc_d6 sc_regs.sr_d6
05325   #define sc_d7 sc_regs.sr_d7
05326   #define sc_a0 sc_regs.sr_a0
05327   #define sc_a1 sc_regs.sr_a1
05328   #define sc_a2 sc_regs.sr_a2
05329   #define sc_a3 sc_regs.sr_a3
05330   #define sc_a4 sc_regs.sr_a4
05331   #define sc_a5 sc_regs.sr_a5
05332   #define sc_fp sc_regs.sr_a6
05333   #define sc_sp sc_regs.sr_sp
05334   #define sc_pc sc_regs.sr_pc
05335   #define sc_psw sc_regs.sr_psw
05336   #endif /* CHIP == M68000 */
05337
05338   /* Values for sc_flags.  Must agree with <minix/jmp_buf.h>. */
05339   #define SC_SIGCONTEXT   2       /* nonzero when signal context is included */
```

```
05340    #define SC_NOREGLOCALS  4       /* nonzero when registers are not to be
05341                                            saved and restored */
05342
05343    _PROTOTYPE( int sigreturn, (struct sigcontext *_scp)                    );
05344
05345    #endif /* _SIGCONTEXT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/sys/stat.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

05400    /* The <sys/stat.h> header defines a struct that is used in the stat() and
05401     * fstat functions.  The information in this struct comes from the i-node of
05402     * some file.  These calls are the only approved way to inspect i-nodes.
05403     */
05404
05405    #ifndef _STAT_H
05406    #define _STAT_H
05407
05408    struct stat {
05409      dev_t st_dev;                 /* major/minor device number */
05410      ino_t st_ino;                 /* i-node number */
05411      mode_t st_mode;               /* file mode, protection bits, etc. */
05412      short int st_nlink;           /* # links; TEMPORARY HACK: should be nlink_t*/
05413      uid_t st_uid;                 /* uid of the file's owner */
05414      short int st_gid;             /* gid; TEMPORARY HACK: should be gid_t */
05415      dev_t st_rdev;
05416      off_t st_size;                /* file size */
05417      time_t st_atime;              /* time of last access */
05418      time_t st_mtime;              /* time of last data modification */
05419      time_t st_ctime;              /* time of last file status change */
05420    };
05421
05422    /* Traditional mask definitions for st_mode. */
05423    /* The ugly casts on only some of the definitions are to avoid suprising sign
05424     * extensions such as S_IFREG != (mode_t) S_IFREG when ints are 32 bits.
05425     */
05426    #define S_IFMT  ((mode_t) 0170000)      /* type of file */
05427    #define S_IFREG ((mode_t) 0100000)      /* regular */
05428    #define S_IFBLK 0060000         /* block special */
05429    #define S_IFDIR 0040000         /* directory */
05430    #define S_IFCHR 0020000         /* character special */
05431    #define S_IFIFO 0010000         /* this is a FIFO */
05432    #define S_ISUID 0004000         /* set user id on execution */
05433    #define S_ISGID 0002000         /* set group id on execution */
05434                                    /* next is reserved for future use */
05435    #define S_ISVTX   01000         /* save swapped text even after use */
05436
05437    /* POSIX masks for st_mode. */
05438    #define S_IRWXU   00700         /* owner:  rwx------ */
05439    #define S_IRUSR   00400         /* owner:  r-------- */
05440    #define S_IWUSR   00200         /* owner:  -w------- */
05441    #define S_IXUSR   00100         /* owner:  --x------ */
05442
05443    #define S_IRWXG   00070         /* group:  ---rwx--- */
05444    #define S_IRGRP   00040         /* group:  ---r----- */
```

```
05445    #define S_IWGRP   00020         /* group:  ----w---- */
05446    #define S_IXGRP   00010         /* group:  -----x--- */
05447
05448    #define S_IRWXO   00007         /* others: ------rwx */
05449    #define S_IROTH   00004         /* others: ------r-- */
05450    #define S_IWOTH   00002         /* others: -------w- */
05451    #define S_IXOTH   00001         /* others: --------x */
05452
05453    /* The following macros test st_mode (from POSIX Sec. 5.6.1.1). */
05454    #define S_ISREG(m)      (((m) & S_IFMT) == S_IFREG)     /* is a reg file */
05455    #define S_ISDIR(m)      (((m) & S_IFMT) == S_IFDIR)     /* is a directory */
05456    #define S_ISCHR(m)      (((m) & S_IFMT) == S_IFCHR)     /* is a char spec */
05457    #define S_ISBLK(m)      (((m) & S_IFMT) == S_IFBLK)     /* is a block spec */
05458    #define S_ISFIFO(m)     (((m) & S_IFMT) == S_IFIFO)     /* is a pipe/FIFO */
05459
05460
05461    /* Function Prototypes. */
05462    #ifndef _ANSI_H
05463    #include <ansi.h>
05464    #endif
05465
05466    _PROTOTYPE( int chmod, (const char *_path, Mode_t _mode)                );
05467    _PROTOTYPE( int fstat, (int _fildes, struct stat *_buf)                 );
05468    _PROTOTYPE( int mkdir, (const char *_path, Mode_t _mode)                );
05469    _PROTOTYPE( int mkfifo, (const char *_path, Mode_t _mode)               );
05470    _PROTOTYPE( int stat, (const char *_path, struct stat *_buf)            );
05471    _PROTOTYPE( mode_t umask, (Mode_t _cmask)                               );
05472
05473    #endif /* _STAT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/sys/times.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

05500    /* The <times.h> header is for time times() system call. */
05501
05502    #ifndef _TIMES_H
05503    #define _TIMES_H
05504
05505    #ifndef _CLOCK_T
05506    #define _CLOCK_T
05507    typedef long clock_t;           /* unit for system accounting */
05508    #endif
05509
05510    struct tms {
05511      clock_t tms_utime;
05512      clock_t tms_stime;
05513      clock_t tms_cutime;
05514      clock_t tms_cstime;
05515    };
05516
05517    /* Function Prototypes. */
05518    #ifndef _ANSI_H
05519    #include <ansi.h>
05520    #endif
05521
05522    _PROTOTYPE( clock_t times, (struct tms *_buffer)                        );
05523
05524    #endif /* _TIMES_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/sys/types.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

05600    /* The <sys/types.h> header contains important data type definitions.
05601     * It is considered good programming practice to use these definitions,
05602     * instead of the underlying base type.  By convention, all type names end
05603     * with _t.
05604     */
05605
05606    #ifndef _TYPES_H
05607    #define _TYPES_H
05608
05609    /* _ANSI is somehow used to determine whether or not the compiler is a
05610     * 16 bit compiler
05611     */
05612    #ifndef _ANSI
05613    #include <ansi.h>
05614    #endif
05615
05616    /* The type size_t holds all results of the sizeof operator.  At first glance,
05617     * it seems obvious that it should be an unsigned int, but this is not always
05618     * the case. For example, MINIX-ST (68000) has 32-bit pointers and 16-bit
05619     * integers. When one asks for the size of a 70K struct or array, the result
05620     * requires 17 bits to express, so size_t must be a long type.  The type
05621     * ssize_t is the signed version of size_t.
05622     */
05623    #ifndef _SIZE_T
05624    #define _SIZE_T
05625    typedef unsigned int size_t;
05626    #endif
05627
05628    #ifndef _SSIZE_T
05629    #define _SSIZE_T
05630    typedef int ssize_t;
05631    #endif
05632
05633    #ifndef _TIME_T
05634    #define _TIME_T
05635    typedef long time_t;               /* time in sec since 1 Jan 1970 0000 GMT */
05636    #endif
05637
05638    #ifndef _CLOCK_T
05639    #define _CLOCK_T
05640    typedef long clock_t;              /* unit for system accounting */
05641    #endif
05642
05643    #ifndef _SIGSET_T
05644    #define _SIGSET_T
05645    typedef unsigned long sigset_t;
05646    #endif
05647
05648    /* Types used in disk, inode, etc. data structures. */
05649    typedef short          dev_t;      /* holds (major|minor) device pair */
05650    typedef char           gid_t;      /* group id */
05651    typedef unsigned short ino_t;      /* i-node number */
05652    typedef unsigned short mode_t;     /* file type and permissions bits */
05653    typedef char         nlink_t;      /* number of links to a file */
05654    typedef unsigned long  off_t;      /* offset within a file */
```

```
05655   typedef int            pid_t;      /* process id (must be signed) */
05656   typedef short          uid_t;      /* user id */
05657   typedef unsigned long zone_t;      /* zone number */
05658   typedef unsigned long block_t;     /* block number */
05659   typedef unsigned long  bit_t;      /* bit number in a bit map */
05660   typedef unsigned short zone1_t;    /* zone number for V1 file systems */
05661   typedef unsigned short bitchunk_t; /* collection of bits in a bitmap */
05662
05663   typedef unsigned char   u8_t;      /* 8 bit type */
05664   typedef unsigned short u16_t;      /* 16 bit type */
05665   typedef unsigned long  u32_t;      /* 32 bit type */
05666
05667   typedef char            i8_t;      /* 8 bit signed type */
05668   typedef short          i16_t;      /* 16 bit signed type */
05669   typedef long           i32_t;      /* 32 bit signed type */
05670
05671   /* The following types are needed because MINIX uses K&R style function
05672    * definitions (for maximum portability).  When a short, such as dev_t, is
05673    * passed to a function with a K&R definition, the compiler automatically
05674    * promotes it to an int.  The prototype must contain an int as the parameter,
05675    * not a short, because an int is what an old-style function definition
05676    * expects.  Thus using dev_t in a prototype would be incorrect.  It would be
05677    * sufficient to just use int instead of dev_t in the prototypes, but Dev_t
05678    * is clearer.
05679    */
05680   typedef int            Dev_t;
05681   typedef int            Gid_t;
05682   typedef int          Nlink_t;
05683   typedef int            Uid_t;
05684   typedef int             U8_t;
05685   typedef unsigned long  U32_t;
05686   typedef int             I8_t;
05687   typedef int            I16_t;
05688   typedef long            I32_t;
05689
05690   /* ANSI C makes writing down the promotion of unsigned types very messy.  When
05691    * sizeof(short) == sizeof(int), there is no promotion, so the type stays
05692    * unsigned.  When the compiler is not ANSI, there is usually no loss of
05693    * unsignedness, and there are usually no prototypes so the promoted type
05694    * doesn't matter.  The use of types like Ino_t is an attempt to use ints
05695    * (which are not promoted) while providing information to the reader.
05696    */
05697
05698   #ifndef _ANSI_H
05699   #include <ansi.h>
05700   #endif
05701
05702   #if _EM_WSIZE == 2 || !defined(_ANSI)
05703   typedef unsigned int      Ino_t;
05704   typedef unsigned int    Zone1_t;
05705   typedef unsigned int Bitchunk_t;
05706   typedef unsigned int      U16_t;
05707   typedef unsigned int  Mode_t;
05708
05709   #else /* _EM_WSIZE == 4, or _EM_WSIZE undefined, or _ANSI defined */
05710   typedef int               Ino_t;
05711   typedef int             Zone1_t;
05712   typedef int          Bitchunk_t;
05713   typedef int               U16_t;
05714   typedef int           Mode_t;
```

```
05715
05716    #endif /* _EM_WSIZE == 2, etc */
05717
05718    /* Signal handler type, e.g. SIG_IGN */
05719    #if defined(_ANSI)
05720    typedef void (*sighandler_t) (int);
05721    #else
05722    typedef void (*sighandler_t)();
05723    #endif
05724
05725    #endif /* _TYPES_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/sys/utsname.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

05800    /* The <sys/utsname.h> header gives the system name. */
05801
05802    #ifndef _UTSNAME_H
05803    #define _UTSNAME_H
05804
05805    struct utsname {
05806      char sysname[15+1];
05807      char nodename[255+1];
05808      char release[11+1];
05809      char version[7+1];
05810      char machine[11+1];
05811      char arch[11+1];
05812    };
05813
05814    /* Function Prototypes. */
05815    #ifndef _ANSI_H
05816    #include <ansi.h>
05817    #endif
05818
05819    _PROTOTYPE( int uname, (struct utsname *_name)                          );
05820
05821    #endif /* _UTSNAME_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/sys/wait.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

05900    /* The <sys/wait.h> header contains macros related to wait(). The value
05901     * returned by wait() and waitpid() depends on whether the process
05902     * terminated by an exit() call, was killed by a signal, or was stopped
05903     * due to job control, as follows:
05904     *
05905     *                               High byte   Low byte
05906     *                              +----------------------+
05907     *      exit(status)            |  status  |    0      |
05908     *                              +----------------------+
05909     *      killed by signal        |    0     |  signal   |
```

```
05910     *                               +----------------------+
05911     *      stopped (job control)    |  signal  |   0177    |
05912     *                               +----------------------+
05913     */
05914
05915    #ifndef _WAIT_H
05916    #define _WAIT_H
05917
05918    #define _LOW(v)         ( (v) & 0377)
05919    #define _HIGH(v)        ( ((v) >> 8) & 0377)
05920
05921    #define WNOHANG         1       /* do not wait for child to exit */
05922    #define WUNTRACED       2       /* for job control; not implemented */
05923
05924    #define WIFEXITED(s)    (_LOW(s) == 0)                      /* normal exit */
05925    #define WEXITSTATUS(s)  (_HIGH(s))                          /* exit status */
05926    #define WTERMSIG(s)     (_LOW(s) & 0177)                    /* sig value */
05927    #define WIFSIGNALED(s)  (((unsigned int)(s)-1 & 0xFFFF) < 0xFF) /* signaled */
05928    #define WIFSTOPPED(s)   (_LOW(s) == 0177)                   /* stopped */
05929    #define WSTOPSIG(s)     (_HIGH(s) & 0377)                   /* stop signal */
05930
05931    /* Function Prototypes. */
05932    #ifndef _ANSI_H
05933    #include <ansi.h>
05934    #endif
05935
05936    _PROTOTYPE( pid_t wait, (int *_stat_loc)                                );
05937    _PROTOTYPE( pid_t waitpid, (pid_t _pid, int *_stat_loc, int _options)   );
05938
05939    #endif /* _WAIT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/minix/boot.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

06000    /* boot.h */
06001
06002    #ifndef _BOOT_H
06003    #define _BOOT_H
06004
06005    /* Redefine root and root image devices as variables.
06006     * This keeps the diffs small but may cause future confusion.
06007     */
06008    #define ROOT_DEV    (boot_parameters.bp_rootdev)
06009    #define IMAGE_DEV   (boot_parameters.bp_ramimagedev)
06010
06011    /* Device numbers of RAM, floppy and hard disk devices.
06012     * h/com.h defines RAM_DEV but only as the minor number.
06013     */
06014    #define DEV_FD0   0x200
06015    #define DEV_HD0   0x300
06016    #define DEV_RAM   0x100
06017    #define DEV_SCSI  0x700 /* Atari TT only */
06018
06019    /* Structure to hold boot parameters. */
```

```
06020    struct bparam_s
06021    {
06022      dev_t bp_rootdev;
06023      dev_t bp_ramimagedev;
06024      unsigned short bp_ramsize;
06025      unsigned short bp_processor;
06026    };
06027
06028    extern struct bparam_s boot_parameters;
06029    #endif /* _BOOT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/minix/callnr.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

06100    #define NCALLS            77    /* number of system calls allowed */
06101
06102    #define EXIT               1
06103    #define FORK               2
06104    #define READ               3
06105    #define WRITE              4
06106    #define OPEN               5
06107    #define CLOSE              6
06108    #define WAIT               7
06109    #define CREAT              8
06110    #define LINK               9
06111    #define UNLINK            10
06112    #define WAITPID           11
06113    #define CHDIR             12
06114    #define TIME              13
06115    #define MKNOD             14
06116    #define CHMOD             15
06117    #define CHOWN             16
06118    #define BRK               17
06119    #define STAT              18
06120    #define LSEEK             19
06121    #define GETPID            20
06122    #define MOUNT             21
06123    #define UMOUNT            22
06124    #define SETUID            23
06125    #define GETUID            24
06126    #define STIME             25
06127    #define PTRACE            26
06128    #define ALARM             27
06129    #define FSTAT             28
06130    #define PAUSE             29
06131    #define UTIME             30
06132    #define ACCESS            33
06133    #define SYNC              36
06134    #define KILL              37
06135    #define RENAME            38
06136    #define MKDIR             39
06137    #define RMDIR             40
06138    #define DUP               41
06139    #define PIPE              42
```

```
06140    #define TIMES             43
06141    #define SETGID            46
06142    #define GETGID            47
06143    #define SIGNAL            48
06144    #define IOCTL             54
06145    #define FCNTL             55
06146    #define EXEC              59
06147    #define UMASK             60
06148    #define CHROOT            61
06149    #define SETSID            62
06150    #define GETPGRP           63
06151
06152    /* The following are not system calls, but are processed like them. */
06153    #define KSIG              64    /* kernel detected a signal */
06154    #define UNPAUSE           65    /* to MM or FS: check for EINTR */
06155    #define REVIVE            67    /* to FS: revive a sleeping process */
06156    #define TASK_REPLY        68    /* to FS: reply code from tty task */
06157
06158    /* Posix signal handling. */
06159    #define SIGACTION         71
06160    #define SIGSUSPEND        72
06161    #define SIGPENDING        73
06162    #define SIGPROCMASK       74
06163    #define SIGRETURN         75
06164
06165    #define REBOOT            76
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/minix/cdrom.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

06200    /* This file contains some structures used by the Mitsumi cdrom driver.
06201     *
06202     *  Feb 13 1995                Author: Michel R. Prevenier
06203     */
06204
06205    /* Index into the mss arrays */
06206    #define MINUTES 0
06207    #define SECONDS 1
06208    #define SECTOR  2
06209
06210    struct cd_play_mss
06211    {
06212            u8_t    begin_mss[3];
06213            u8_t    end_mss[3];
06214    };
06215
06216
06217    struct cd_play_track
06218    {
06219            u8_t    begin_track;
06220            u8_t    end_track;
06221    };
06222
06223
06224    struct cd_disk_info
```

```
06225    {
06226            u8_t    first_track;
06227            u8_t    last_track;
06228            u8_t    disk_length_mss[3];
06229            u8_t    first_track_mss[3];
06230    };
06231
06232
06233    struct cd_toc_entry
06234    {
06235            u8_t    control_address;
06236            u8_t    track_nr;
06237            u8_t    index_nr;
06238            u8_t    track_time_mss[3];
06239            u8_t    reserved;
06240            u8_t    position_mss[3];
06241    };
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/minix/com.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

06300    /* System calls. */
06301    #define SEND               1    /* function code for sending messages */
06302    #define RECEIVE            2    /* function code for receiving messages */
06303    #define BOTH               3    /* function code for SEND + RECEIVE */
06304    #define ANY   (NR_PROCS+100)    /* receive(ANY, buf) accepts from any source */
06305
06306    /* Task numbers, function codes and reply codes. */
06307
06308    /* The values of several task numbers depend on whether they or other tasks
06309     * are enabled.  They are defined as (PREVIOUS_TASK - ENABLE_TASK) in general.
06310     * ENABLE_TASK is either 0 or 1, so a task either gets a new number, or gets
06311     * the same number as the previous task and is further unused.
06312     * The TTY task must always have the most negative number so that it is
06313     * initialized first.  Many of the TTY function codes are shared with other
06314     * tasks.
06315     */
06316
06317    #define TTY             (DL_ETH - 1)
06318                                    /* terminal I/O class */
06319    #       define CANCEL       0   /* general req to force a task to cancel */
06320    #       define HARD_INT     2   /* fcn code for all hardware interrupts */
06321    #       define DEV_READ     3   /* fcn code for reading from tty */
06322    #       define DEV_WRITE    4   /* fcn code for writing to tty */
06323    #       define DEV_IOCTL    5   /* fcn code for ioctl */
06324    #       define DEV_OPEN     6   /* fcn code for opening tty */
06325    #       define DEV_CLOSE    7   /* fcn code for closing tty */
06326    #       define SCATTERED_IO 8   /* fcn code for multiple reads/writes */
06327    #       define TTY_SETPGRP  9   /* fcn code for setpgroup */
06328    #       define TTY_EXIT    10   /* a process group leader has exited */
06329    #       define OPTIONAL_IO 16   /* modifier to DEV_* codes within vector */
06330    #       define SUSPEND   -998   /* used in interrupts when tty has no data */
06331
06332    #define DL_ETH          (CDROM - ENABLE_NETWORKING)
06333                                    /* networking task */
06334
```

```
06335   /* Message type for data link layer reqests. */
06336   #       define DL_WRITE         3
06337   #       define DL_WRITEV        4
06338   #       define DL_READ          5
06339   #       define DL_READV         6
06340   #       define DL_INIT          7
06341   #       define DL_STOP          8
06342   #       define DL_GETSTAT       9
06343
06344   /* Message type for data link layer replies. */
06345   #       define DL_INIT_REPLY    20
06346   #       define DL_TASK_REPLY    21
06347
06348   #       define DL_PORT          m2_i1
06349   #       define DL_PROC          m2_i2
06350   #       define DL_COUNT         m2_i3
06351   #       define DL_MODE          m2_l1
06352   #       define DL_CLCK          m2_l2
06353   #       define DL_ADDR          m2_p1
06354   #       define DL_STAT          m2_l1
06355
06356   /* Bits in 'DL_STAT' field of DL replies. */
06357   #       define DL_PACK_SEND     0x01
06358   #       define DL_PACK_RECV     0x02
06359   #       define DL_READ_IP       0x04
06360
06361   /* Bits in 'DL_MODE' field of DL requests. */
06362   #       define DL_NOMODE        0x0
06363   #       define DL_PROMISC_REQ   0x2
06364   #       define DL_MULTI_REQ     0x4
06365   #       define DL_BROAD_REQ     0x8
06366
06367   #       define NW_OPEN          DEV_OPEN
06368   #       define NW_CLOSE         DEV_CLOSE
06369   #       define NW_READ          DEV_READ
06370   #       define NW_WRITE         DEV_WRITE
06371   #       define NW_IOCTL         DEV_IOCTL
06372   #       define NW_CANCEL        CANCEL
06373
06374   #define CDROM           (AUDIO - ENABLE_CDROM)
06375                                   /* cd-rom device task */
06376
06377   #define AUDIO           (MIXER - ENABLE_AUDIO)
06378   #define MIXER           (SCSI - ENABLE_AUDIO)
06379                                   /* audio & mixer device tasks */
06380
06381   #define SCSI            (WINCHESTER - ENABLE_SCSI)
06382                                   /* scsi device task */
06383
06384   #define WINCHESTER      (SYN_ALRM_TASK - ENABLE_WINI)
06385                                   /* winchester (hard) disk class */
06386
06387   #define SYN_ALRM_TASK     -8    /* task to send CLOCK_INT messages */
06388
06389   #define IDLE              -7    /* task to run when there's nothing to run */
06390
06391   #define PRINTER           -6    /* printer I/O class */
06392
06393   #define FLOPPY            -5    /* floppy disk class */
06394
```

```
06395    #define MEM               -4    /* /dev/ram, /dev/(k)mem and /dev/null class */
06396    #       define NULL_MAJOR  1    /* major device for /dev/null */
06397    #       define RAM_DEV     0    /* minor device for /dev/ram */
06398    #       define MEM_DEV     1    /* minor device for /dev/mem */
06399    #       define KMEM_DEV    2    /* minor device for /dev/kmem */
06400    #       define NULL_DEV    3    /* minor device for /dev/null */
06401
06402    #define CLOCK             -3    /* clock class */
06403    #       define SET_ALARM   1    /* fcn code to CLOCK, set up alarm */
06404    #       define GET_TIME    3    /* fcn code to CLOCK, get real time */
06405    #       define SET_TIME    4    /* fcn code to CLOCK, set real time */
06406    #       define GET_UPTIME  5    /* fcn code to CLOCK, get uptime */
06407    #       define SET_SYNC_AL 6    /* fcn code to CLOCK, set up alarm which */
06408                                    /* times out with a send */
06409    #       define REAL_TIME   1    /* reply from CLOCK: here is real time */
06410    #       define CLOCK_INT   HARD_INT
06411                                    /* this code will only be sent by */
06412                                    /* SYN_ALRM_TASK to a task that requested a */
06413                                    /* synchronous alarm */
06414
06415    #define SYSTASK           -2    /* internal functions */
06416    #       define SYS_XIT        1 /* fcn code for sys_xit(parent, proc) */
06417    #       define SYS_GETSP      2 /* fcn code for sys_sp(proc, &new_sp) */
06418    #       define SYS_OLDSIG     3 /* fcn code for sys_oldsig(proc, sig) */
06419    #       define SYS_FORK       4 /* fcn code for sys_fork(parent, child) */
06420    #       define SYS_NEWMAP     5 /* fcn code for sys_newmap(procno, map_ptr) */
06421    #       define SYS_COPY       6 /* fcn code for sys_copy(ptr) */
06422    #       define SYS_EXEC       7 /* fcn code for sys_exec(procno, new_sp) */
06423    #       define SYS_TIMES      8 /* fcn code for sys_times(procno, bufptr) */
06424    #       define SYS_ABORT      9 /* fcn code for sys_abort() */
06425    #       define SYS_FRESH     10 /* fcn code for sys_fresh()  (Atari only) */
06426    #       define SYS_KILL      11 /* fcn code for sys_kill(proc, sig) */
06427    #       define SYS_GBOOT     12 /* fcn code for sys_gboot(procno, bootptr) */
06428    #       define SYS_UMAP      13 /* fcn code for sys_umap(procno, etc) */
06429    #       define SYS_MEM       14 /* fcn code for sys_mem() */
06430    #       define SYS_TRACE     15 /* fcn code for sys_trace(req,pid,addr,data) */
06431    #       define SYS_VCOPY     16 /* fnc code for sys_vcopy(src_proc, dest_proc,
06432                                       vcopy_s, vcopy_ptr) */
06433    #       define SYS_SENDSIG   17 /* fcn code for sys_sendsig(&sigmsg) */
06434    #       define SYS_SIGRETURN 18 /* fcn code for sys_sigreturn(&sigmsg) */
06435    #       define SYS_ENDSIG    19 /* fcn code for sys_endsig(procno) */
06436    #       define SYS_GETMAP    20 /* fcn code for sys_getmap(procno, map_ptr) */
06437
06438    #define HARDWARE          -1    /* used as source on interrupt generated msgs*/
06439
06440    /* Names of message fields for messages to CLOCK task. */
06441    #define DELTA_TICKS    m6_l1    /* alarm interval in clock ticks */
06442    #define FUNC_TO_CALL   m6_f1    /* pointer to function to call */
06443    #define NEW_TIME       m6_l1    /* value to set clock to (SET_TIME) */
06444    #define CLOCK_PROC_NR  m6_i1    /* which proc (or task) wants the alarm? */
06445    #define SECONDS_LEFT   m6_l1    /* how many seconds were remaining */
06446
06447    /* Names of message fields used for messages to block and character tasks. */
06448    #define DEVICE         m2_i1    /* major-minor device */
06449    #define PROC_NR        m2_i2    /* which (proc) wants I/O? */
06450    #define COUNT          m2_i3    /* how many bytes to transfer */
06451    #define REQUEST        m2_i3    /* ioctl request code */
06452    #define POSITION       m2_l1    /* file offset */
06453    #define ADDRESS        m2_p1    /* core buffer address */
06454
```

```
06455   /* Names of message fields for messages to TTY task. */
06456   #define TTY_LINE       DEVICE   /* message parameter: terminal line */
06457   #define TTY_REQUEST    COUNT    /* message parameter: ioctl request code */
06458   #define TTY_SPEK       POSITION /* message parameter: ioctl speed, erasing */
06459   #define TTY_FLAGS      m2_l2    /* message parameter: ioctl tty mode */
06460   #define TTY_PGRP       m2_i3    /* message parameter: process group */
06461
06462   /* Names of the message fields for QIC 02 status reply from tape driver */
06463   #define TAPE_STAT0      m2_l1
06464   #define TAPE_STAT1      m2_l2
06465
06466   /* Names of messages fields used in reply messages from tasks. */
06467   #define REP_PROC_NR    m2_i1    /* # of proc on whose behalf I/O was done */
06468   #define REP_STATUS     m2_i2    /* bytes transferred or error number */
06469
06470   /* Names of fields for copy message to SYSTASK. */
06471   #define SRC_SPACE      m5_c1    /* T or D space (stack is also D) */
06472   #define SRC_PROC_NR    m5_i1    /* process to copy from */
06473   #define SRC_BUFFER     m5_l1    /* virtual address where data come from */
06474   #define DST_SPACE      m5_c2    /* T or D space (stack is also D) */
06475   #define DST_PROC_NR    m5_i2    /* process to copy to */
06476   #define DST_BUFFER     m5_l2    /* virtual address where data go to */
06477   #define COPY_BYTES     m5_l3    /* number of bytes to copy */
06478
06479   /* Field names for accounting, SYSTASK and miscellaneous. */
06480   #define USER_TIME      m4_l1    /* user time consumed by process */
06481   #define SYSTEM_TIME    m4_l2    /* system time consumed by process */
06482   #define CHILD_UTIME    m4_l3    /* user time consumed by process' children */
06483   #define CHILD_STIME    m4_l4    /* sys time consumed by process' children */
06484   #define BOOT_TICKS     m4_l5    /* number of clock ticks since boot time */
06485
06486   #define PROC1          m1_i1    /* indicates a process */
06487   #define PROC2          m1_i2    /* indicates a process */
06488   #define PID            m1_i3    /* process id passed from MM to kernel */
06489   #define STACK_PTR      m1_p1    /* used for stack ptr in sys_exec, sys_getsp */
06490   #define PR             m6_i1    /* process number for sys_sig */
06491   #define SIGNUM         m6_i2    /* signal number for sys_sig */
06492   #define FUNC           m6_f1    /* function pointer for sys_sig */
06493   #define MEM_PTR        m1_p1    /* tells where memory map is for sys_newmap */
06494   #define NAME_PTR       m1_p2    /* tells where program name is for dmp */
06495   #define IP_PTR         m1_p3    /* initial value for ip after exec */
06496   #define SIG_PROC       m2_i1    /* process number for inform */
06497   #define SIG_MAP        m2_l1    /* used by kernel for passing signal bit map */
06498   #define SIG_MSG_PTR    m1_i1    /* pointer to info to build sig catch stack */
06499   #define SIG_CTXT_PTR   m1_p1    /* pointer to info to restore signal context */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/minix/config.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

06500    #ifndef _CONFIG_H
06501    #define _CONFIG_H
06502
06503    /* Minix release and version numbers. */
06504    #define OS_RELEASE "2.0"
06505    #define OS_VERSION "0"
06506
06507    /* This file sets configuration parameters for the MINIX kernel, FS, and MM.
06508     * It is divided up into two main sections.  The first section contains
06509     * user-settable parameters.  In the second section, various internal system
06510     * parameters are set based on the user-settable parameters.
06511     */
06512
06513    /*===========================================================================*
06514     *              This section contains user-settable parameters               *
06515     *===========================================================================*/
06516    #define MACHINE       IBM_PC    /* Must be one of the names listed below */
06517
06518    #define IBM_PC             1    /* any  8088 or 80x86-based system */
06519    #define SUN_4             40    /* any Sun SPARC-based system */
06520    #define SUN_4_60          40    /* Sun-4/60 (aka SparcStation 1 or Campus) */
06521    #define ATARI             60    /* ATARI ST/STe/TT (68000/68030) */
06522    #define AMIGA             61    /* Commodore Amiga (68000) */
06523    #define MACINTOSH         62    /* Apple Macintosh (68000) */
06524
06525    /* Word size in bytes (a constant equal to sizeof(int)). */
06526    #if __ACK__
06527    #define _WORD_SIZE      _EM_WSIZE
06528    #endif
06529
06530
06531    /* If ROBUST is set to 1, writes of i-node, directory, and indirect blocks
06532     * from the cache happen as soon as the blocks are modified.  This gives a more
06533     * robust, but slower, file system.  If it is set to 0, these blocks are not
06534     * given any special treatment, which may cause problems if the system crashes.
06535     */
06536    #define ROBUST             0    /* 0 for speed, 1 for robustness */
06537
06538    /* Number of slots in the process table for user processes. */
06539    #define NR_PROCS          32
06540
06541    /* The buffer cache should be made as large as you can afford. */
06542    #if (MACHINE == IBM_PC && _WORD_SIZE == 2)
06543    #define NR_BUFS           40    /* # blocks in the buffer cache */
06544    #define NR_BUF_HASH       64    /* size of buf hash table; MUST BE POWER OF 2*/
06545    #endif
06546
06547    #if (MACHINE == IBM_PC && _WORD_SIZE == 4)
06548    #define NR_BUFS         1024    /* # blocks in the buffer cache */
06549    #define NR_BUF_HASH     2048    /* size of buf hash table; MUST BE POWER OF 2*/
06550    #endif
06551
06552    #if (MACHINE == SUN_4_60)
06553    #define NR_BUFS          512    /* # blocks in the buffer cache (<=1536) */
06554    #define NR_BUF_HASH      512    /* size of buf hash table; MUST BE POWER OF 2*/
```

```
06555    #endif
06556
06557    #if (MACHINE == ATARI)
06558    #define NR_BUFS         1536    /* # blocks in the buffer cache (<=1536) */
06559    #define NR_BUF_HASH     2048    /* size of buf hash table; MUST BE POWER OF 2*/
06560    #endif
06561
06562    /* Defines for kernel configuration. */
06563    #define AUTO_BIOS          0    /* xt_wini.c - use Western's autoconfig BIOS */
06564    #define LINEWRAP           1    /* console.c - wrap lines at column 80 */
06565    #define ALLOW_GAP_MESSAGES 1    /* proc.c - allow messages in the gap between
06566                                     * the end of bss and lowest stack address */
06567
06568    /* Enable or disable the second level file system cache on the RAM disk. */
06569    #define ENABLE_CACHE2      1
06570
06571    /* Include or exclude device drivers.  Set to 1 to include, 0 to exclude. */
06572    #define ENABLE_NETWORKING  1    /* enable TCP/IP code */
06573    #define ENABLE_AT_WINI     1    /* enable AT winchester driver */
06574    #define ENABLE_BIOS_WINI   1    /* enable BIOS winchester driver */
06575    #define ENABLE_ESDI_WINI   1    /* enable ESDI winchester driver */
06576    #define ENABLE_XT_WINI     0    /* enable XT winchester driver */
06577    #define ENABLE_ADAPTEC_SCSI 1   /* enable ADAPTEC SCSI driver */
06578    #define ENABLE_MITSUMI_CDROM 0  /* enable Mitsumi CD-ROM driver */
06579    #define ENABLE_SB_AUDIO    0    /* enable Soundblaster audio driver */
06580
06581    /* DMA_SECTORS may be increased to speed up DMA based drivers. */
06582    #define DMA_SECTORS        1    /* DMA buffer size (must be >= 1) */
06583
06584    /* Include or exclude backwards compatibility code. */
06585    #define ENABLE_BINCOMPAT   0    /* for binaries using obsolete calls */
06586    #define ENABLE_SRCCOMPAT   0    /* for sources using obsolete calls */
06587
06588    /* Determine which device to use for pipes. */
06589    #define PIPE_DEV    ROOT_DEV    /* put pipes on root device */
06590
06591    /* NR_CONS, NR_RS_LINES, and NR_PTYS determine the number of terminals the
06592     * system can handle.
06593     */
06594    #define NR_CONS            2    /* # system consoles (1 to 8) */
06595    #define NR_RS_LINES        2    /* # rs232 terminals (0, 1, or 2) */
06596    #define NR_PTYS            4    /* # pseudo terminals (0 to 64) */
06597
06598    #if (MACHINE == ATARI)
06599    /* The next define says if you have an ATARI ST or TT */
06600    #define ATARI_TYPE        TT
06601    #define ST                 1    /* all ST's and Mega ST's */
06602    #define STE                2    /* all STe and Mega STe's */
06603    #define TT                 3
06604
06605    /* if SCREEN is set to 1 graphical screen operations are possible */
06606    #define SCREEN             1
06607
06608    /* This define says whether the keyboard generates VT100 or IBM_PC escapes. */
06609    #define KEYBOARD       VT100    /* either VT100 or IBM_PC */
06610    #define VT100            100
06611
06612    /* The next define determines the kind of partitioning. */
06613    #define PARTITIONING   SUPRA    /* one of the following or ATARI */
06614    #define SUPRA              1    /*ICD, SUPRA and BMS are all the same */
```

```
06615    #define BMS               1
06616    #define ICD               1
06617    #define CBHD              2
06618    #define EICKMANN          3
06619
06620    /* Define the number of hard disk drives on your system. */
06621    #define NR_ACSI_DRIVES     3    /* typically 0 or 1 */
06622    #define NR_SCSI_DRIVES     1    /* typically 0 (ST, STe) or 1 (TT) */
06623
06624    /* Some systems need to have a little delay after each winchester
06625     * commands. These systems need FAST_DISK set to 0. Other disks do not
06626     * need this delay, and thus can have FAST_DISK set to 1 to avoid this delay.
06627     */
06628    #define FAST_DISK          1    /* 0 or 1 */
06629
06630    /* Note: if you want to make your kernel smaller, you can set NR_FD_DRIVES
06631     * to 0. You will still be able to boot minix.img from floppy. However, you
06632     * MUST fetch both the root and usr filesystem from a hard disk
06633     */
06634
06635    /* Define the number of floppy disk drives on your system. */
06636    #define NR_FD_DRIVES       1    /* 0, 1, 2 */
06637
06638    /* This configuration define controls parallel printer code. */
06639    #define PAR_PRINTER        1    /* disable (0) / enable (1) parallel printer */
06640
06641    /* This configuration define controls disk controller clock code. */
06642    #define HD_CLOCK           1    /* disable (0) / enable (1) hard disk clock */
06643
06644    #endif
06645
06646
06647    /*===========================================================================*
06648     *      There are no user-settable parameters after this line                *
06649     *===========================================================================*/
06650    /* Set the CHIP type based on the machine selected. The symbol CHIP is actually
06651     * indicative of more than just the CPU.  For example, machines for which
06652     * CHIP == INTEL are expected to have 8259A interrrupt controllers and the
06653     * other properties of IBM PC/XT/AT/386 types machines in general. */
06654    #define INTEL             1     /* CHIP type for PC, XT, AT, 386 and clones */
06655    #define M68000            2     /* CHIP type for Atari, Amiga, Macintosh    */
06656    #define SPARC             3     /* CHIP type for SUN-4 (e.g. SPARCstation)  */
06657
06658    /* Set the FP_FORMAT type based on the machine selected, either hw or sw    */
06659    #define FP_NONE           0     /* no floating point support                */
06660    #define FP_IEEE           1     /* conform IEEE floating point standard     */
06661
06662    #if (MACHINE == IBM_PC)
06663    #define CHIP          INTEL
06664    #define SHADOWING         0
06665    #define ENABLE_WINI     (ENABLE_AT_WINI || ENABLE_BIOS_WINI || \
06666                             ENABLE_ESDI_WINI || ENABLE_XT_WINI)
06667    #define ENABLE_SCSI     (ENABLE_ADAPTEC_SCSI)
06668    #define ENABLE_CDROM    (ENABLE_MITSUMI_CDROM)
06669    #define ENABLE_AUDIO    (ENABLE_SB_AUDIO)
06670    #endif
06671
06672    #if (MACHINE == ATARI) || (MACHINE == AMIGA) || (MACHINE == MACINTOSH)
06673    #define CHIP         M68000
06674    #define SHADOWING         1
```

```
06675    #endif
06676
06677    #if (MACHINE == SUN_4) || (MACHINE == SUN_4_60)
06678    #define CHIP          SPARC
06679    #define FP_FORMAT   FP_IEEE
06680    #define SHADOWING         0
06681    #endif
06682
06683    #if (MACHINE == ATARI) || (MACHINE == SUN_4)
06684    #define ASKDEV            1     /* ask for boot device */
06685    #define FASTLOAD          1     /* use multiple block transfers to init ram */
06686    #endif
06687
06688    #if (ATARI_TYPE == TT) /* and all other 68030's */
06689    #define FPP
06690    #undef SHADOWING
06691    #define SHADOWING 0
06692    #endif
06693
06694    #ifndef FP_FORMAT
06695    #define FP_FORMAT   FP_NONE
06696    #endif
06697
06698    /* The file buf.h uses MAYBE_WRITE_IMMED. */
06699    #if ROBUST
06700    #define MAYBE_WRITE_IMMED  WRITE_IMMED  /* slower but perhaps safer */
06701    #else
06702    #define MAYBE_WRITE_IMMED 0             /* faster */
06703    #endif
06704
06705    #ifndef MACHINE
06706    error "In <minix/config.h> please define MACHINE"
06707    #endif
06708
06709    #ifndef CHIP
06710    error "In <minix/config.h> please define MACHINE to have a legal value"
06711    #endif
06712
06713    #if (MACHINE == 0)
06714    error "MACHINE has incorrect value (0)"
06715    #endif
06716
06717    #endif /* _CONFIG_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/minix/const.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

06800    /* Copyright (C) 1997 by Prentice-Hall, Inc.  Permission is hereby granted
06801     * to redistribute the binary and source programs of this system for
06802     * educational or research purposes.  For other use, written permission from
06803     * Prentice-Hall is required.
06804     */
06805
06806    #define EXTERN        extern    /* used in *.h files */
06807    #define PRIVATE       static    /* PRIVATE x limits the scope of x */
06808    #define PUBLIC                  /* PUBLIC is the opposite of PRIVATE */
06809    #define FORWARD       static    /* some compilers require this to be 'static'*/
```

```
06810
06811    #define TRUE               1    /* used for turning integers into Booleans */
06812    #define FALSE              0    /* used for turning integers into Booleans */
06813
06814    #define HZ                60    /* clock freq (software settable on IBM-PC) */
06815    #define BLOCK_SIZE      1024    /* # bytes in a disk block */
06816    #define SUPER_USER (uid_t) 0    /* uid_t of superuser */
06817
06818    #define MAJOR              8    /* major device = (dev>>MAJOR) & 0377 */
06819    #define MINOR              0    /* minor device = (dev>>MINOR) & 0377 */
06820
06821    #define NULL     ((void *)0)    /* null pointer */
06822    #define CPVEC_NR          16    /* max # of entries in a SYS_VCOPY request */
06823    #define NR_IOREQS       MIN(NR_BUFS, 64)
06824                                  /* maximum number of entries in an iorequest */
06825
06826    #define NR_SEGS            3    /* # segments per process */
06827    #define T                  0    /* proc[i].mem_map[T] is for text */
06828    #define D                  1    /* proc[i].mem_map[D] is for data */
06829    #define S                  2    /* proc[i].mem_map[S] is for stack */
06830
06831    /* Process numbers of some important processes. */
06832    #define MM_PROC_NR         0    /* process number of memory manager */
06833    #define FS_PROC_NR         1    /* process number of file system */
06834    #define INET_PROC_NR       2    /* process number of the TCP/IP server */
06835    #define INIT_PROC_NR    (INET_PROC_NR + ENABLE_NETWORKING)
06836                                  /* init -- the process that goes multiuser */
06837    #define LOW_USER        (INET_PROC_NR + ENABLE_NETWORKING)
06838                                  /* first user not part of operating system */
06839
06840    /* Miscellaneous */
06841    #define BYTE            0377    /* mask for 8 bits */
06842    #define READING            0    /* copy data to user */
06843    #define WRITING            1    /* copy data from user */
06844    #define NO_NUM        0x8000    /* used as numerical argument to panic() */
06845    #define NIL_PTR   (char *) 0    /* generally useful expression */
06846    #define HAVE_SCATTERED_IO  1    /* scattered I/O is now standard */
06847
06848    /* Macros. */
06849    #define MAX(a, b)   ((a) > (b) ? (a) : (b))
06850    #define MIN(a, b)   ((a) < (b) ? (a) : (b))
06851
06852    /* Number of tasks. */
06853    #define NR_TASKS        (9 + ENABLE_WINI + ENABLE_SCSI + ENABLE_CDROM \
06854                            + ENABLE_NETWORKING + 2 * ENABLE_AUDIO)
06855
06856    /* Memory is allocated in clicks. */
06857    #if (CHIP == INTEL)
06858    #define CLICK_SIZE       256    /* unit in which memory is allocated */
06859    #define CLICK_SHIFT        8    /* log2 of CLICK_SIZE */
06860    #endif
06861
06862    #if (CHIP == SPARC) || (CHIP == M68000)
06863    #define CLICK_SIZE      4096    /* unit in which memory is alocated */
06864    #define CLICK_SHIFT       12    /* 2log of CLICK_SIZE */
06865    #endif
06866
06867    #define click_to_round_k(n) \
06868            ((unsigned) ((((unsigned long) (n) << CLICK_SHIFT) + 512) / 1024))
06869    #if CLICK_SIZE < 1024
```

```
06870    #define k_to_click(n) ((n) * (1024 / CLICK_SIZE))
06871    #else
06872    #define k_to_click(n) ((n) / (CLICK_SIZE / 1024))
06873    #endif
06874
06875    #define ABS             -999    /* this process means absolute memory */
06876
06877    /* Flag bits for i_mode in the inode. */
06878    #define I_TYPE          0170000 /* this field gives inode type */
06879    #define I_REGULAR       0100000 /* regular file, not dir or special */
06880    #define I_BLOCK_SPECIAL 0060000 /* block special file */
06881    #define I_DIRECTORY     0040000 /* file is a directory */
06882    #define I_CHAR_SPECIAL  0020000 /* character special file */
06883    #define I_NAMED_PIPE    0010000 /* named pipe (FIFO) */
06884    #define I_SET_UID_BIT   0004000 /* set effective uid_t on exec */
06885    #define I_SET_GID_BIT   0002000 /* set effective gid_t on exec */
06886    #define ALL_MODES       0006777 /* all bits for user, group and others */
06887    #define RWX_MODES       0000777 /* mode bits for RWX only */
06888    #define R_BIT           0000004 /* Rwx protection bit */
06889    #define W_BIT           0000002 /* rWx protection bit */
06890    #define X_BIT           0000001 /* rwX protection bit */
06891    #define I_NOT_ALLOC     0000000 /* this inode is free */
06892
06893    /* Some limits. */
06894    #define MAX_BLOCK_NR  ((block_t) 077777777)     /* largest block number */
06895    #define HIGHEST_ZONE   ((zone_t) 077777777)     /* largest zone number */
06896    #define MAX_INODE_NR      ((ino_t) 0177777)     /* largest inode number */
06897    #define MAX_FILE_POS ((off_t) 037777777777)     /* largest legal file offset */
06898
06899    #define NO_BLOCK              ((block_t) 0)     /* absence of a block number */
06900    #define NO_ENTRY                ((ino_t) 0)     /* absence of a dir entry */
06901    #define NO_ZONE                ((zone_t) 0)     /* absence of a zone number */
06902    #define NO_DEV                  ((dev_t) 0)     /* absence of a device numb */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/minix/dl_eth.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

07000    /* The eth_stat struct is used in a DL_GETSTAT request the the ehw_task. */
07001
07002    #ifndef _ETH_HW_H
07003    #define _ETH_HW_H
07004
07005    typedef struct eth_stat
07006    {
07007      unsigned long ets_recvErr,    /* # receive errors */
07008            ets_sendErr,            /* # send error */
07009            ets_OVW,                /* # buffer overwrite warnings */
07010            ets_CRCerr,             /* # crc errors of read */
07011            ets_frameAll,           /* # frames not alligned (# bits % 8 != 0) */
07012            ets_missedP,            /* # packets missed due to slow processing */
07013            ets_packetR,            /* # packets received */
07014            ets_packetT,            /* # packets transmitted */
07015            ets_transDef,           /* # transmission defered (Tx was busy) */
07016            ets_collision,          /* # collissions */
07017            ets_transAb,            /* # Tx aborted due to excess collisions */
07018            ets_carrSense,          /* # carrier sense lost */
07019            ets_fifoUnder,          /* # FIFO underruns (processor too busy) */
```

```
07020             ets_fifoOver,           /* # FIFO overruns (processor too busy) */
07021             ets_CDheartbeat,        /* # times unable to transmit collision sig*/
07022             ets_OWC;                /* # times out of window collision */
07023     } eth_stat_t;
07024
07025     #endif /* _ETH_HW_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/minix/fslib.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
07100     _PROTOTYPE( int bitmapsize, (bit_t _nr_bits)                          );
07101     _PROTOTYPE( unsigned conv2, (int _norm, int _w)                       );
07102     _PROTOTYPE( long conv4, (int _norm, long _x)                          );
07103     _PROTOTYPE( void conv_inode, (struct inode *_rip, d1_inode *_dip,
07104                                   d2_inode *_dip2, int _rw_flag, int _magic) );
07105     _PROTOTYPE( void old_icopy, (struct inode *_rip, d1_inode *_dip,
07106                                                 int _direction, int _norm));
07107     _PROTOTYPE( void new_icopy, (struct inode *_rip, d2_inode *_dip,
07108                                                 int _direction, int _norm));
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/minix/jmp_buf.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
07200     /* This file is intended for use by assembly language programs that
07201      * need to manipulate a jmp_buf.  It may only be used by those systems
07202      * for which a jmp_buf is identical to a struct sigcontext.
07203      */
07204
07205     #ifndef _JMP_BUF_H
07206     #define _JMP_BUF_H
07207
07208     #if !defined(CHIP)
07209     #include "error, configuration is not known"
07210     #endif
07211
07212     #if (CHIP == INTEL)
07213     #if _WORD_SIZE == 4
07214     #define JB_FLAGS         0
07215     #define JB_MASK          4
07216     #define JB_GS            8
07217     #define JB_FS           10
07218     #define JB_ES           12
07219     #define JB_DS           14
07220     #define JB_DI           16
07221     #define JB_SI           20
07222     #define JB_BP           24
07223     #define JB_ST           28
07224     #define JB_BX           32
07225     #define JB_DX           36
07226     #define JB_CX           40
07227     #define JB_AX           44
07228     #define JB_RETADR       48
07229     #define JB_IP           52
```

```
07230    #define JB_CS           56
07231    #define JB_PSW          60
07232    #define JB_SP           64
07233    #define JB_SS           68
07234    #else /* _WORD_SIZE == 2 */
07235    #define JB_FLAGS         0
07236    #define JB_MASK          2
07237    #define JB_ES            6
07238    #define JB_DS            8
07239    #define JB_DI           10
07240    #define JB_SI           12
07241    #define JB_BP           14
07242    #define JB_ST           16
07243    #define JB_BX           18
07244    #define JB_DX           20
07245    #define JB_CX           22
07246    #define JB_AX           24
07247    #define JB_RETADR       26
07248    #define JB_IP           28
07249    #define JB_CS           30
07250    #define JB_PSW          32
07251    #define JB_SP           34
07252    #define JB_SS           36
07253    #endif /* _WORD_SIZE == 2 */
07254    #else /* !(CHIP == INTEL) */
07255    #if (CHIP == M68000)
07256    #define JB_FLAGS         0
07257    #define JB_MASK          2
07258    #define JB_RETREG        6
07259    #define JB_D1           10
07260    #define JB_D2           14
07261    #define JB_D3           18
07262    #define JB_D4           22
07263    #define JB_D5           26
07264    #define JB_D6           20
07265    #define JB_D7           34
07266    #define JB_A0           38
07267    #define JB_A1           42
07268    #define JB_A2           46
07269    #define JB_A3           50
07270    #define JB_A4           54
07271    #define JB_A5           58
07272    #define JB_A6           62
07273    #define JB_SP           66
07274    #define JB_PC           70
07275    #define JB_PSW          74
07276    #else /* !(CHIP == INTEL) && !(CHIP == M68000) */
07277    #include "error, CHIP is not supported"
07278    #endif /* (CHIP == INTEL) */
07279
07280    /* Defines from C headers needed in assembly code.  The headers have too
07281     * much C stuff to used directly.
07282     */
07283    #define SIG_BLOCK       0       /* must agree with <signal.h> */
07284    #define SC_SIGCONTEXT   2       /* must agree with <sys/sigcontext.h> */
07285    #define SC_NOREGLOCALS  4       /* must agree with <sys/sigcontext.h> */
07286    #endif /* _JMP_BUF_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/minix/keymap.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

07300    /*      keymap.h - defines for keymapping               Author: Marcus Hampel
07301     */
07302    #ifndef _SYS__KEYMAP_H
07303    #define _SYS__KEYMAP_H
07304
07305    #define C(c)     ((c) & 0x1F)    /* Map to control code          */
07306    #define A(c)     ((c) | 0x80)    /* Set eight bit (ALT)          */
07307    #define CA(c)    A(C(c))         /* Control-Alt                  */
07308    #define L(c)     ((c) | HASCAPS) /* Add "Caps Lock has effect" attribute */
07309
07310    #define EXT      0x0100          /* Normal function keys         */
07311    #define CTRL     0x0200          /* Control key                  */
07312    #define SHIFT    0x0400          /* Shift key                    */
07313    #define ALT      0x0800          /* Alternate key                */
07314    #define EXTKEY   0x1000          /* extended keycode             */
07315    #define HASCAPS  0x8000          /* Caps Lock has effect         */
07316
07317    /* Numeric keypad */
07318    #define HOME     (0x01 + EXT)
07319    #define END      (0x02 + EXT)
07320    #define UP       (0x03 + EXT)
07321    #define DOWN     (0x04 + EXT)
07322    #define LEFT     (0x05 + EXT)
07323    #define RIGHT    (0x06 + EXT)
07324    #define PGUP     (0x07 + EXT)
07325    #define PGDN     (0x08 + EXT)
07326    #define MID      (0x09 + EXT)
07327    #define NMIN     (0x0A + EXT)
07328    #define PLUS     (0x0B + EXT)
07329    #define INSRT    (0x0C + EXT)
07330
07331    /* Alt + Numeric keypad */
07332    #define AHOME    (0x01 + ALT)
07333    #define AEND     (0x02 + ALT)
07334    #define AUP      (0x03 + ALT)
07335    #define ADOWN    (0x04 + ALT)
07336    #define ALEFT    (0x05 + ALT)
07337    #define ARIGHT   (0x06 + ALT)
07338    #define APGUP    (0x07 + ALT)
07339    #define APGDN    (0x08 + ALT)
07340    #define AMID     (0x09 + ALT)
07341    #define ANMIN    (0x0A + ALT)
07342    #define APLUS    (0x0B + ALT)
07343    #define AINSRT   (0x0C + ALT)
07344
07345    /* Ctrl + Numeric keypad */
07346    #define CHOME    (0x01 + CTRL)
07347    #define CEND     (0x02 + CTRL)
07348    #define CUP      (0x03 + CTRL)
07349    #define CDOWN    (0x04 + CTRL)
07350    #define CLEFT    (0x05 + CTRL)
07351    #define CRIGHT   (0x06 + CTRL)
07352    #define CPGUP    (0x07 + CTRL)
07353    #define CPGDN    (0x08 + CTRL)
07354    #define CMID     (0x09 + CTRL)
```

```
07355    #define CNMIN   (0x0A + CTRL)
07356    #define CPLUS   (0x0B + CTRL)
07357    #define CINSRT  (0x0C + CTRL)
07358
07359    /* Lock keys */
07360    #define CALOCK  (0x0D + EXT)    /* caps lock    */
07361    #define NLOCK   (0x0E + EXT)    /* number lock  */
07362    #define SLOCK   (0x0F + EXT)    /* scroll lock  */
07363
07364    /* Function keys */
07365    #define F1      (0x10 + EXT)
07366    #define F2      (0x11 + EXT)
07367    #define F3      (0x12 + EXT)
07368    #define F4      (0x13 + EXT)
07369    #define F5      (0x14 + EXT)
07370    #define F6      (0x15 + EXT)
07371    #define F7      (0x16 + EXT)
07372    #define F8      (0x17 + EXT)
07373    #define F9      (0x18 + EXT)
07374    #define F10     (0x19 + EXT)
07375    #define F11     (0x1A + EXT)
07376    #define F12     (0x1B + EXT)
07377
07378    /* Alt+Fn */
07379    #define AF1     (0x10 + ALT)
07380    #define AF2     (0x11 + ALT)
07381    #define AF3     (0x12 + ALT)
07382    #define AF4     (0x13 + ALT)
07383    #define AF5     (0x14 + ALT)
07384    #define AF6     (0x15 + ALT)
07385    #define AF7     (0x16 + ALT)
07386    #define AF8     (0x17 + ALT)
07387    #define AF9     (0x18 + ALT)
07388    #define AF10    (0x19 + ALT)
07389    #define AF11    (0x1A + ALT)
07390    #define AF12    (0x1B + ALT)
07391
07392    /* Ctrl+Fn */
07393    #define CF1     (0x10 + CTRL)
07394    #define CF2     (0x11 + CTRL)
07395    #define CF3     (0x12 + CTRL)
07396    #define CF4     (0x13 + CTRL)
07397    #define CF5     (0x14 + CTRL)
07398    #define CF6     (0x15 + CTRL)
07399    #define CF7     (0x16 + CTRL)
07400    #define CF8     (0x17 + CTRL)
07401    #define CF9     (0x18 + CTRL)
07402    #define CF10    (0x19 + CTRL)
07403    #define CF11    (0x1A + CTRL)
07404    #define CF12    (0x1B + CTRL)
07405
07406    /* Shift+Fn */
07407    #define SF1     (0x10 + SHIFT)
07408    #define SF2     (0x11 + SHIFT)
07409    #define SF3     (0x12 + SHIFT)
07410    #define SF4     (0x13 + SHIFT)
07411    #define SF5     (0x14 + SHIFT)
07412    #define SF6     (0x15 + SHIFT)
07413    #define SF7     (0x16 + SHIFT)
07414    #define SF8     (0x17 + SHIFT)
```

```
07415    #define SF9     (0x18 + SHIFT)
07416    #define SF10    (0x19 + SHIFT)
07417    #define SF11    (0x1A + SHIFT)
07418    #define SF12    (0x1B + SHIFT)
07419
07420    /* Alt+Shift+Fn */
07421    #define ASF1    (0x10 + ALT + SHIFT)
07422    #define ASF2    (0x11 + ALT + SHIFT)
07423    #define ASF3    (0x12 + ALT + SHIFT)
07424    #define ASF4    (0x13 + ALT + SHIFT)
07425    #define ASF5    (0x14 + ALT + SHIFT)
07426    #define ASF6    (0x15 + ALT + SHIFT)
07427    #define ASF7    (0x16 + ALT + SHIFT)
07428    #define ASF8    (0x17 + ALT + SHIFT)
07429    #define ASF9    (0x18 + ALT + SHIFT)
07430    #define ASF10   (0x19 + ALT + SHIFT)
07431    #define ASF11   (0x1A + ALT + SHIFT)
07432    #define ASF12   (0x1B + ALT + SHIFT)
07433
07434    #define MAP_COLS        6       /* Number of columns in keymap */
07435    #define NR_SCAN_CODES   0x80    /* Number of scan codes (rows in keymap) */
07436
07437    typedef unsigned short keymap_t[NR_SCAN_CODES * MAP_COLS];
07438
07439    #define KEY_MAGIC       "KMAZ"  /* Magic number of keymap file */
07440
07441    #endif /* _SYS__KEYMAP_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/minix/minlib.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

07500    #ifndef _MINLIB
07501    #define _MINLIB
07502
07503    #ifndef _ANSI_H
07504    #include <ansi.h>
07505    #endif
07506
07507    /* Miscellaneous BSD. */
07508    _PROTOTYPE(void swab, (char *_from, char *_to, int _count));
07509    _PROTOTYPE(char *itoa, (int _n));
07510    _PROTOTYPE(char *getpass, (const char *_prompt));
07511
07512    /* Miscellaneous MINIX. */
07513    _PROTOTYPE(void std_err, (char *_s));
07514    _PROTOTYPE(void prints, (const char *_s, ...));
07515    _PROTOTYPE(int fsversion, (char *_dev, char *_prog));
07516    _PROTOTYPE(int getprocessor, (void));
07517    _PROTOTYPE(int load_mtab, (char *_prog_name));
07518    _PROTOTYPE(int rewrite_mtab, (char *_prog_name));
07519    _PROTOTYPE(int get_mtab_entry, (char *_s1, char *_s2, char *_s3, char *_s4));
07520    _PROTOTYPE(int put_mtab_entry, (char *_s1, char *_s2, char *_s3, char *_s4));
07521
07522    #endif
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/minix/partition.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

07600    /*      minix/partition.h                               Author: Kees J. Bot
07601     *                                                              7 Dec 1995
07602     * Place of a partition on disk and the disk geometry,
07603     * for use with the DIOCGETP and DIOCSETP ioctl's.
07604     */
07605    #ifndef _MINIX__PARTITION_H
07606    #define _MINIX__PARTITION_H
07607
07608    struct partition {
07609      u32_t base;           /* byte offset to the partition start */
07610      u32_t size;           /* number of bytes in the partition */
07611      unsigned cylinders;   /* disk geometry */
07612      unsigned heads;
07613      unsigned sectors;
07614    };
07615    #endif /* _MINIX__PARTITION_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/minix/sound.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

07700    /* Definitions used by /dev/audio and /dev/mixer.
07701     *
07702     *  Feb 13 1995                 Author: Michel R. Prevenier
07703     */
07704
07705    #ifndef SOUND_H
07706    #define SOUND_H
07707
07708
07709    /* ------- Mixer stuff ------- */
07710
07711    /* Available devices */
07712    enum Device
07713    {
07714      Master,     /* Master volume */
07715      Dac,        /* DSP, digitized sound */
07716      Fm,         /* Fm synthesized sound */
07717      Cd,         /* Compact */
07718      Line,       /* Line in */
07719      Mic,        /* Microphone */
07720      Speaker,    /* Pc speaker */
07721      Treble,     /* Treble */
07722      Bass        /* Bass */
07723    };
07724
07725    enum InputState
07726    {
07727      ON, OFF
07728    };
07729
```

```
07730    /* Volume levels range from 0 to 31, bass & treble range from 0 to 15 */
07731    struct volume_level
07732    {
07733      enum Device   device;
07734      int           left;
07735      int           right;
07736    };
07737
07738    struct inout_ctrl
07739    {
07740      enum Device           device;
07741      enum InputState       left;
07742      enum InputState       right;
07743    };
07744
07745    #endif /* SOUND_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/minix/syslib.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
07800    /* Prototypes for system library functions. */
07801
07802    #ifndef _SYSLIB_H
07803    #define _SYSLIB_H
07804
07805    /* Hide names to avoid name space pollution. */
07806    #define sendrec         _sendrec
07807    #define receive         _receive
07808    #define send            _send
07809
07810    /* Minix user+system library. */
07811    _PROTOTYPE( void printk, (char *_fmt, ...)                              );
07812    _PROTOTYPE( int sendrec, (int _src_dest, message *_m_ptr)               );
07813    _PROTOTYPE( int _taskcall, (int _who, int _syscallnr, message *_msgptr) );
07814
07815    /* Minix system library. */
07816    _PROTOTYPE( int receive, (int _src, message *_m_ptr)                    );
07817    _PROTOTYPE( int send, (int _dest, message *_m_ptr)                      );
07818
07819    _PROTOTYPE( int sys_abort, (int _how, ...)                              );
07820    _PROTOTYPE( int sys_adjmap, (int _proc, struct mem_map *_ptr,
07821                    vir_clicks _data_clicks, vir_clicks _sp)                );
07822    _PROTOTYPE( int sys_copy, (int _src_proc, int _src_seg, phys_bytes _src_vir,
07823            int _dst_proc, int _dst_seg, phys_bytes _dst_vir, phys_bytes _bytes));
07824    _PROTOTYPE( int sys_exec, (int _proc, char *_ptr, int _traced,
07825                                    char *_aout, vir_bytes _initpc)         );
07826    _PROTOTYPE( int sys_execmap, (int _proc, struct mem_map *_ptr)          );
07827    _PROTOTYPE( int sys_fork, (int _parent, int _child, int _pid,
07828                                        phys_clicks _shadow)                );
07829    _PROTOTYPE( int sys_fresh, (int _proc, struct mem_map *_ptr,
07830            phys_clicks _dc, phys_clicks *_basep, phys_clicks *_sizep)      );
07831    _PROTOTYPE( int sys_getsp, (int _proc, vir_bytes *_newsp)               );
07832    _PROTOTYPE( int sys_newmap, (int _proc, struct mem_map *_ptr)           );
07833    _PROTOTYPE( int sys_getmap, (int _proc, struct mem_map *_ptr)           );
07834    _PROTOTYPE( int sys_sendsig, (int _proc, struct sigmsg *_ptr)           );
```

```
07835    _PROTOTYPE( int sys_oldsig, (int _proc, int _sig, sighandler_t _sighandler));
07836    _PROTOTYPE( int sys_endsig, (int _proc)                                  );
07837    _PROTOTYPE( int sys_sigreturn, (int _proc, vir_bytes _scp, int _flags)  );
07838    _PROTOTYPE( int sys_trace, (int _req, int _procnr, long _addr, long *_data_p));
07839    _PROTOTYPE( int sys_xit, (int _parent, int _proc, phys_clicks *_basep,
07840                                                 phys_clicks *_sizep));
07841    _PROTOTYPE( int sys_kill, (int _proc, int _sig)                          );
07842    _PROTOTYPE( int sys_times, (int _proc, clock_t _ptr[5])                  );
07843
07844    #endif /* _SYSLIB_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/minix/type.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
07900    #ifndef _TYPE_H
07901    #define _TYPE_H
07902    #ifndef _MINIX_TYPE_H
07903    #define _MINIX_TYPE_H
07904
07905    /* Type definitions. */
07906    typedef unsigned int vir_clicks; /* virtual  addresses and lengths in clicks */
07907    typedef unsigned long phys_bytes;/* physical addresses and lengths in bytes */
07908    typedef unsigned int phys_clicks;/* physical addresses and lengths in clicks */
07909
07910    #if (CHIP == INTEL)
07911    typedef unsigned int vir_bytes; /* virtual addresses and lengths in bytes */
07912    #endif
07913
07914    #if (CHIP == M68000)
07915    typedef unsigned long vir_bytes;/* virtual addresses and lengths in bytes */
07916    #endif
07917
07918    #if (CHIP == SPARC)
07919    typedef unsigned long vir_bytes;/* virtual addresses and lengths in bytes */
07920    #endif
07921
07922    /* Types relating to messages. */
07923    #define M1                 1
07924    #define M3                 3
07925    #define M4                 4
07926    #define M3_STRING         14
07927
07928    typedef struct {int m1i1, m1i2, m1i3; char *m1p1, *m1p2, *m1p3;} mess_1;
07929    typedef struct {int m2i1, m2i2, m2i3; long m2l1, m2l2; char *m2p1;} mess_2;
07930    typedef struct {int m3i1, m3i2; char *m3p1; char m3ca1[M3_STRING];} mess_3;
07931    typedef struct {long m4l1, m4l2, m4l3, m4l4, m4l5;} mess_4;
07932    typedef struct {char m5c1, m5c2; int m5i1, m5i2; long m5l1, m5l2, m5l3;}mess_5;
07933    typedef struct {int m6i1, m6i2, m6i3; long m6l1; sighandler_t m6f1;} mess_6;
07934
07935    typedef struct {
07936      int m_source;                /* who sent the message */
07937      int m_type;                  /* what kind of message is it */
07938      union {
07939            mess_1 m_m1;
```

```
07940           mess_2 m_m2;
07941           mess_3 m_m3;
07942           mess_4 m_m4;
07943           mess_5 m_m5;
07944           mess_6 m_m6;
07945     } m_u;
07946   } message;
07947
07948   /* The following defines provide names for useful members. */
07949   #define m1_i1  m_u.m_m1.m1i1
07950   #define m1_i2  m_u.m_m1.m1i2
07951   #define m1_i3  m_u.m_m1.m1i3
07952   #define m1_p1  m_u.m_m1.m1p1
07953   #define m1_p2  m_u.m_m1.m1p2
07954   #define m1_p3  m_u.m_m1.m1p3
07955
07956   #define m2_i1  m_u.m_m2.m2i1
07957   #define m2_i2  m_u.m_m2.m2i2
07958   #define m2_i3  m_u.m_m2.m2i3
07959   #define m2_l1  m_u.m_m2.m2l1
07960   #define m2_l2  m_u.m_m2.m2l2
07961   #define m2_p1  m_u.m_m2.m2p1
07962
07963   #define m3_i1  m_u.m_m3.m3i1
07964   #define m3_i2  m_u.m_m3.m3i2
07965   #define m3_p1  m_u.m_m3.m3p1
07966   #define m3_ca1 m_u.m_m3.m3ca1
07967
07968   #define m4_l1  m_u.m_m4.m4l1
07969   #define m4_l2  m_u.m_m4.m4l2
07970   #define m4_l3  m_u.m_m4.m4l3
07971   #define m4_l4  m_u.m_m4.m4l4
07972   #define m4_l5  m_u.m_m4.m4l5
07973
07974   #define m5_c1  m_u.m_m5.m5c1
07975   #define m5_c2  m_u.m_m5.m5c2
07976   #define m5_i1  m_u.m_m5.m5i1
07977   #define m5_i2  m_u.m_m5.m5i2
07978   #define m5_l1  m_u.m_m5.m5l1
07979   #define m5_l2  m_u.m_m5.m5l2
07980   #define m5_l3  m_u.m_m5.m5l3
07981
07982   #define m6_i1  m_u.m_m6.m6i1
07983   #define m6_i2  m_u.m_m6.m6i2
07984   #define m6_i3  m_u.m_m6.m6i3
07985   #define m6_l1  m_u.m_m6.m6l1
07986   #define m6_f1  m_u.m_m6.m6f1
07987
07988   struct mem_map {
07989     vir_clicks mem_vir;           /* virtual address */
07990     phys_clicks mem_phys;         /* physical address */
07991     vir_clicks mem_len;           /* length */
07992   };
07993
07994   struct iorequest_s {
07995     long io_position;             /* position in device file (really off_t) */
07996     char *io_buf;                 /* buffer in user space */
07997     int io_nbytes;                /* size of request */
07998     unsigned short io_request;    /* read, write (optionally) */
07999   };
```

```
08000    #endif /* _TYPE_H */
08001
08002    typedef struct {
08003      vir_bytes iov_addr;           /* address of an I/O buffer */
08004      vir_bytes iov_size;           /* sizeof an I/O buffer */
08005    } iovec_t;
08006
08007    typedef struct {
08008      vir_bytes cpv_src;            /* src address of data */
08009      vir_bytes cpv_dst;            /* dst address of data */
08010      vir_bytes cpv_size;           /* size of data */
08011    } cpvec_t;
08012
08013    /* MM passes the address of a structure of this type to KERNEL when
08014     * do_sendsig() is invoked as part of the signal catching mechanism.
08015     * The structure contain all the information that KERNEL needs to build
08016     * the signal stack.
08017     */
08018    struct sigmsg {
08019      int sm_signo;                 /* signal number being caught */
08020      unsigned long sm_mask;        /* mask to restore when handler returns */
08021      vir_bytes sm_sighandler;      /* address of handler */
08022      vir_bytes sm_sigreturn;       /* address of _sigreturn in C library */
08023      vir_bytes sm_stkptr;          /* user stack pointer */
08024    };
08025
08026    #define MESS_SIZE (sizeof(message))     /* might need usizeof from fs here */
08027    #define NIL_MESS ((message *) 0)
08028
08029    struct psinfo {         /* information for the ps(1) program */
08030      u16_t nr_tasks, nr_procs;     /* NR_TASKS and NR_PROCS constants. */
08031      vir_bytes proc, mproc, fproc; /* addresses of the main process tables. */
08032    };
08033
08034    #endif /* _MINIX_TYPE_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/ibm/diskparm.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

08100    /* PC (and AT) BIOS structure to hold disk parameters.  Under Minix, it is
08101     * used mainly for formatting.
08102     */
08103
08104    #ifndef _DISKPARM_H
08105    #define _DISKPARM_H
08106    struct disk_parameter_s {
08107      char spec1;
08108      char spec2;
08109      char motor_turnoff_sec;
08110      char sector_size_code;
08111      char sectors_per_cylinder;
08112      char gap_length;
08113      char dtl;
08114      char gap_length_for_format;
```

```
08115       char fill_byte_for_format;
08116       char head_settle_msec;
08117       char motor_start_eigth_sec;
08118     };
08119     #endif /* _DISKPARM_H */
```

++
include/ibm/partition.h
++

```
08200     /* Description of entry in partition table.  */
08201     #ifndef _PARTITION_H
08202     #define _PARTITION_H
08203
08204     struct part_entry {
08205       unsigned char bootind;        /* boot indicator 0/ACTIVE_FLAG  */
08206       unsigned char start_head;     /* head value for first sector   */
08207       unsigned char start_sec;      /* sector value + cyl bits for first sector */
08208       unsigned char start_cyl;      /* track value for first sector  */
08209       unsigned char sysind;         /* system indicator              */
08210       unsigned char last_head;      /* head value for last sector    */
08211       unsigned char last_sec;       /* sector value + cyl bits for last sector */
08212       unsigned char last_cyl;       /* track value for last sector   */
08213       unsigned long lowsec;         /* logical first sector          */
08214       unsigned long size;           /* size of partition in sectors  */
08215     };
08216
08217     #define ACTIVE_FLAG     0x80    /* value for active in bootind field (hd0) */
08218     #define NR_PARTITIONS   4       /* number of entries in partition table */
08219     #define PART_TABLE_OFF  0x1BE   /* offset of partition table in boot sector */
08220
08221     /* Partition types. */
08222     #define MINIX_PART      0x81    /* Minix partition type */
08223     #define NO_PART         0x00    /* unused entry */
08224     #define OLD_MINIX_PART  0x80    /* created before 1.4b, obsolete */
08225     #define EXT_PART        0x05    /* extended partition */
08226
08227     #endif /* _PARTITION_H */
```

++
src/kernel/assert.h
++

```
08300     /*
08301     assert.h
08302     */
08303     #ifndef ASSERT_H
08304     #define ASSERT_H
08305
08306     #if DEBUG
08307
08308     #define INIT_ASSERT     static char *assert_file= __FILE__;
08309
```

```
08310    void bad_assertion(char *file, int line, char *what);
08311    void bad_compare(char *file, int line, int lhs, char *what, int rhs);
08312
08313    #define assert(x)       (!(x) ? bad_assertion(assert_file, __LINE__, #x) \
08314                                                                    : (void) 0)
08315    #define compare(a,t,b)  (!((a) t (b)) ? bad_compare(assert_file, __LINE__, \
08316                                    (a), #a " " #t " " #b, (b)) : (void) 0)
08317    #else /* !DEBUG */
08318
08319    #define INIT_ASSERT     /* nothing */
08320
08321    #define assert(x)       (void)0
08322    #define compare(a,t,b)  (void)0
08323
08324    #endif /* !DEBUG */
08325
08326    #endif /* ASSERT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/const.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

08400    /* General constants used by the kernel. */
08401
08402    #if (CHIP == INTEL)
08403
08404    #define K_STACK_BYTES   1024    /* how many bytes for the kernel stack */
08405
08406    #define INIT_PSW      0x0200    /* initial psw */
08407    #define INIT_TASK_PSW 0x1200    /* initial psw for tasks (with IOPL 1) */
08408    #define TRACEBIT       0x100    /* OR this with psw in proc[] for tracing */
08409    #define SETPSW(rp, new) /* permits only certain bits to be set */ \
08410            ((rp)->p_reg.psw = (rp)->p_reg.psw & ~0xCD5 | (new) & 0xCD5)
08411
08412    /* Initial sp for mm, fs and init.
08413     *      2 bytes for short jump
08414     *      2 bytes unused
08415     *      3 words for init_org[] used by fs only
08416     *      3 words for real mode debugger trap (actually needs 1 more)
08417     *      3 words for save and restart temporaries
08418     *      3 words for interrupt
08419     * Leave no margin, to flush bugs early.
08420     */
08421    #define INIT_SP (2 + 2 + 3 * 2 + 3 * 2 + 3 * 2 + 3 * 2 + 3 * 2)
08422
08423    #define HCLICK_SHIFT       4    /* log2 of HCLICK_SIZE */
08424    #define HCLICK_SIZE       16    /* hardware segment conversion magic */
08425    #if CLICK_SIZE >= HCLICK_SIZE
08426    #define click_to_hclick(n) ((n) << (CLICK_SHIFT - HCLICK_SHIFT))
08427    #else
08428    #define click_to_hclick(n) ((n) >> (HCLICK_SHIFT - CLICK_SHIFT))
08429    #endif
08430    #define hclick_to_physb(n) ((phys_bytes) (n) << HCLICK_SHIFT)
08431    #define physb_to_hclick(n) ((n) >> HCLICK_SHIFT)
08432
08433    /* Interrupt vectors defined/reserved by processor. */
08434    #define DIVIDE_VECTOR      0    /* divide error */
```

```
08435    #define DEBUG_VECTOR       1    /* single step (trace) */
08436    #define NMI_VECTOR         2    /* non-maskable interrupt */
08437    #define BREAKPOINT_VECTOR  3    /* software breakpoint */
08438    #define OVERFLOW_VECTOR    4    /* from INTO */
08439
08440    /* Fixed system call vector. */
08441    #define SYS_VECTOR        32    /* system calls are made with int SYSVEC */
08442    #define SYS386_VECTOR     33    /* except 386 system calls use this */
08443    #define LEVEL0_VECTOR     34    /* for execution of a function at level 0 */
08444
08445    /* Suitable irq bases for hardware interrupts.  Reprogram the 8259(s) from
08446     * the PC BIOS defaults since the BIOS doesn't respect all the processor's
08447     * reserved vectors (0 to 31).
08448     */
08449    #define BIOS_IRQ0_VEC   0x08    /* base of IRQ0-7 vectors used by BIOS */
08450    #define BIOS_IRQ8_VEC   0x70    /* base of IRQ8-15 vectors used by BIOS */
08451    #define IRQ0_VECTOR     0x28    /* more or less arbitrary, but > SYS_VECTOR */
08452    #define IRQ8_VECTOR     0x30    /* together for simplicity */
08453
08454    /* Hardware interrupt numbers. */
08455    #define NR_IRQ_VECTORS    16
08456    #define CLOCK_IRQ          0
08457    #define KEYBOARD_IRQ       1
08458    #define CASCADE_IRQ        2    /* cascade enable for 2nd AT controller */
08459    #define ETHER_IRQ          3    /* default ethernet interrupt vector */
08460    #define SECONDARY_IRQ      3    /* RS232 interrupt vector for port 2 */
08461    #define RS232_IRQ          4    /* RS232 interrupt vector for port 1 */
08462    #define XT_WINI_IRQ        5    /* xt winchester */
08463    #define FLOPPY_IRQ         6    /* floppy disk */
08464    #define PRINTER_IRQ        7
08465    #define AT_WINI_IRQ       14    /* at winchester */
08466
08467    /* Interrupt number to hardware vector. */
08468    #define BIOS_VECTOR(irq)        \
08469            (((irq) < 8 ? BIOS_IRQ0_VEC : BIOS_IRQ8_VEC) + ((irq) & 0x07))
08470    #define VECTOR(irq)     \
08471            (((irq) < 8 ? IRQ0_VECTOR : IRQ8_VECTOR) + ((irq) & 0x07))
08472
08473    /* BIOS hard disk parameter vectors. */
08474    #define WINI_0_PARM_VEC 0x41
08475    #define WINI_1_PARM_VEC 0x46
08476
08477    /* 8259A interrupt controller ports. */
08478    #define INT_CTL         0x20    /* I/O port for interrupt controller */
08479    #define INT_CTLMASK     0x21    /* setting bits in this port disables ints */
08480    #define INT2_CTL        0xA0    /* I/O port for second interrupt controller */
08481    #define INT2_CTLMASK    0xA1    /* setting bits in this port disables ints */
08482
08483    /* Magic numbers for interrupt controller. */
08484    #define ENABLE          0x20    /* code used to re-enable after an interrupt */
08485
08486    /* Sizes of memory tables. */
08487    #define NR_MEMS            3    /* number of chunks of memory */
08488
08489    /* Miscellaneous ports. */
08490    #define PCR             0x65    /* Planar Control Register */
08491    #define PORT_B          0x61    /* I/O port for 8255 port B (kbd, beeper...) */
08492    #define TIMER0          0x40    /* I/O port for timer channel 0 */
08493    #define TIMER2          0x42    /* I/O port for timer channel 2 */
08494    #define TIMER_MODE      0x43    /* I/O port for timer mode control */
```

```
08495
08496   #endif /* (CHIP == INTEL) */
08497
08498   #if (CHIP == M68000)
08499
08500   #define K_STACK_BYTES   1024    /* how many bytes for the kernel stack */
08501
08502   /* Sizes of memory tables. */
08503   #define NR_MEMS            2    /* number of chunks of memory */
08504
08505   /* p_reg contains: d0-d7, a0-a6,   in that order. */
08506   #define NR_REGS           15    /* number of general regs in each proc slot */
08507
08508   #define TRACEBIT      0x8000    /* or this with psw in proc[] for tracing */
08509   #define SETPSW(rp, new)         /* permits only certain bits to be set */ \
08510           ((rp)->p_reg.psw = (rp)->p_reg.psw & ~0xFF | (new) & 0xFF)
08511
08512   #define MEM_BYTES  0xffffffff   /* memory size for /dev/mem */
08513
08514   #ifdef __ACK__
08515   #define FSTRUCOPY
08516   #endif
08517
08518   #endif /* (CHIP == M68000) */
08519
08520   /* The following items pertain to the scheduling queues. */
08521   #define TASK_Q             0    /* ready tasks are scheduled via queue 0 */
08522   #define SERVER_Q           1    /* ready servers are scheduled via queue 1 */
08523   #define USER_Q             2    /* ready users are scheduled via queue 2 */
08524
08525   #if (MACHINE == ATARI)
08526   #define SHADOW_Q           3    /* runnable, but shadowed processes */
08527   #define NQ                 4    /* # of scheduling queues */
08528   #else
08529   #define NQ                 3    /* # of scheduling queues */
08530   #endif
08531
08532   /* Env_parse() return values. */
08533   #define EP_UNSET        0       /* variable not set */
08534   #define EP_OFF          1       /* var = off */
08535   #define EP_ON           2       /* var = on (or field left blank) */
08536   #define EP_SET          3       /* var = 1:2:3 (nonblank field) */
08537
08538   /* To translate an address in kernel space to a physical address.  This is
08539    * the same as umap(proc_ptr, D, vir, sizeof(*vir)), but a lot less costly.
08540    */
08541   #define vir2phys(vir)   (data_base + (vir_bytes) (vir))
08542
08543   #define printf        printk    /* the kernel really uses printk, not printf */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/dp8390.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

08600    /*
08601    dp8390.h
08602
08603    Created:        before Dec 28, 1992 by Philip Homburg
08604    */
08605
08606    /* National Semiconductor DP8390 Network Interface Controller. */
08607
08608                                    /* Page 0, for reading ------------- */
08609    #define DP_CR           0x0     /* Read side of Command Register     */
08610    #define DP_CLDA0        0x1     /* Current Local Dma Address 0       */
08611    #define DP_CLDA1        0x2     /* Current Local Dma Address 1       */
08612    #define DP_BNRY         0x3     /* Boundary Pointer                  */
08613    #define DP_TSR          0x4     /* Transmit Status Register          */
08614    #define DP_NCR          0x5     /* Number of Collisions Register     */
08615    #define DP_FIFO         0x6     /* Fifo ??                           */
08616    #define DP_ISR          0x7     /* Interrupt Status Register         */
08617    #define DP_CRDA0        0x8     /* Current Remote Dma Address 0      */
08618    #define DP_CRDA1        0x9     /* Current Remote Dma Address 1      */
08619    #define DP_DUM1         0xA     /* unused                            */
08620    #define DP_DUM2         0xB     /* unused                            */
08621    #define DP_RSR          0xC     /* Receive Status Register           */
08622    #define DP_CNTR0        0xD     /* Tally Counter 0                   */
08623    #define DP_CNTR1        0xE     /* Tally Counter 1                   */
08624    #define DP_CNTR2        0xF     /* Tally Counter 2                   */
08625
08626                                    /* Page 0, for writing ------------- */
08627    #define DP_CR           0x0     /* Write side of Command Register    */
08628    #define DP_PSTART       0x1     /* Page Start Register               */
08629    #define DP_PSTOP        0x2     /* Page Stop Register                */
08630    #define DP_BNRY         0x3     /* Boundary Pointer                  */
08631    #define DP_TPSR         0x4     /* Transmit Page Start Register      */
08632    #define DP_TBCR0        0x5     /* Transmit Byte Count Register 0    */
08633    #define DP_TBCR1        0x6     /* Transmit Byte Count Register 1    */
08634    #define DP_ISR          0x7     /* Interrupt Status Register         */
08635    #define DP_RSAR0        0x8     /* Remote Start Address Register 0   */
08636    #define DP_RSAR1        0x9     /* Remote Start Address Register 1   */
08637    #define DP_RBCR0        0xA     /* Remote Byte Count Register 0      */
08638    #define DP_RBCR1        0xB     /* Remote Byte Count Register 1      */
08639    #define DP_RCR          0xC     /* Receive Configuration Register    */
08640    #define DP_TCR          0xD     /* Transmit Configuration Register   */
08641    #define DP_DCR          0xE     /* Data Configuration Register       */
08642    #define DP_IMR          0xF     /* Interrupt Mask Register           */
08643
08644                                    /* Page 1, read/write -------------- */
08645    #define DP_CR           0x0     /* Command Register                  */
08646    #define DP_PAR0         0x1     /* Physical Address Register 0       */
08647    #define DP_PAR1         0x2     /* Physical Address Register 1       */
08648    #define DP_PAR2         0x3     /* Physical Address Register 2       */
08649    #define DP_PAR3         0x4     /* Physical Address Register 3       */
08650    #define DP_PAR4         0x5     /* Physical Address Register 4       */
08651    #define DP_PAR5         0x6     /* Physical Address Register 5       */
08652    #define DP_CURR         0x7     /* Current Page Register             */
08653    #define DP_MAR0         0x8     /* Multicast Address Register 0      */
08654    #define DP_MAR1         0x9     /* Multicast Address Register 1      */
```

```
08655   #define DP_MAR2         0xA     /* Multicast Address Register 2      */
08656   #define DP_MAR3         0xB     /* Multicast Address Register 3      */
08657   #define DP_MAR4         0xC     /* Multicast Address Register 4      */
08658   #define DP_MAR5         0xD     /* Multicast Address Register 5      */
08659   #define DP_MAR6         0xE     /* Multicast Address Register 6      */
08660   #define DP_MAR7         0xF     /* Multicast Address Register 7      */
08661
08662   /* Bits in dp_cr */
08663   #define CR_STP          0x01    /* Stop: software reset              */
08664   #define CR_STA          0x02    /* Start: activate NIC               */
08665   #define CR_TXP          0x04    /* Transmit Packet                   */
08666   #define CR_DMA          0x38    /* Mask for DMA control              */
08667   #define CR_DM_NOP       0x00    /* DMA: No Operation                 */
08668   #define CR_DM_RR        0x08    /* DMA: Remote Read                  */
08669   #define CR_DM_RW        0x10    /* DMA: Remote Write                 */
08670   #define CR_DM_SP        0x18    /* DMA: Send Packet                  */
08671   #define CR_DM_ABORT     0x20    /* DMA: Abort Remote DMA Operation   */
08672   #define CR_PS           0xC0    /* Mask for Page Select              */
08673   #define CR_PS_P0        0x00    /* Register Page 0                   */
08674   #define CR_PS_P1        0x40    /* Register Page 1                   */
08675   #define CR_PS_T0        0x80    /* Test Mode Register Map ??         */
08676   #define CR_SP_T1        0xC0    /* Test Mode Register Map ??         */
08677
08678   /* Bits in dp_isr */
08679   #define ISR_MASK        0x3F
08680   #define ISR_PRX         0x01    /* Packet Received with no errors    */
08681   #define ISR_PTX         0x02    /* Packet Transmitted with no errors */
08682   #define ISR_RXE         0x04    /* Receive Error                     */
08683   #define ISR_TXE         0x08    /* Transmit Error                    */
08684   #define ISR_OVW         0x10    /* Overwrite Warning                 */
08685   #define ISR_CNT         0x20    /* Counter Overflow                  */
08686   #define ISR_RDC         0x40    /* Remote DMA Complete               */
08687   #define ISR_RST         0x80    /* Reset Status                      */
08688
08689   /* Bits in dp_imr */
08690   #define IMR_PRXE        0x01    /* Packet Received iEnable           */
08691   #define IMR_PTXE        0x02    /* Packet Transmitted iEnable        */
08692   #define IMR_RXEE        0x04    /* Receive Error iEnable             */
08693   #define IMR_TXEE        0x08    /* Transmit Error iEnable            */
08694   #define IMR_OVWE        0x10    /* Overwrite Warning iEnable         */
08695   #define IMR_CNTE        0x20    /* Counter Overflow iEnable          */
08696   #define IMR_RDCE        0x40    /* DMA Complete iEnable              */
08697
08698   /* Bits in dp_dcr */
08699   #define DCR_WTS         0x01    /* Word Transfer Select              */
08700   #define DCR_BYTEWIDE    0x00    /* WTS: byte wide transfers          */
08701   #define DCR_WORDWIDE    0x01    /* WTS: word wide transfers          */
08702   #define DCR_BOS         0x02    /* Byte Order Select                 */
08703   #define DCR_LTLENDIAN   0x00    /* BOS: Little Endian                */
08704   #define DCR_BIGENDIAN   0x02    /* BOS: Big Endian                   */
08705   #define DCR_LAS         0x04    /* Long Address Select               */
08706   #define DCR_BMS         0x08    /* Burst Mode Select                 */
08707   #define DCR_AR          0x10    /* Autoinitialize Remote             */
08708   #define DCR_FTS         0x60    /* Fifo Threshold Select             */
08709   #define DCR_2BYTES      0x00    /* 2 bytes                           */
08710   #define DCR_4BYTES      0x40    /* 4 bytes                           */
08711   #define DCR_8BYTES      0x20    /* 8 bytes                           */
08712   #define DCR_12BYTES     0x60    /* 12 bytes                          */
08713
08714   /* Bits in dp_tcr */
```

```
08715    #define TCR_CRC         0x01    /* Inhibit CRC                       */
08716    #define TCR_ELC         0x06    /* Encoded Loopback Control          */
08717    #define TCR_NORMAL      0x00    /* ELC: Normal Operation             */
08718    #define TCR_INTERNAL    0x02    /* ELC: Internal Loopback            */
08719    #define TCR_0EXTERNAL   0x04    /* ELC: External Loopback LPBK=0     */
08720    #define TCR_1EXTERNAL   0x06    /* ELC: External Loopback LPBK=1     */
08721    #define TCR_ATD         0x08    /* Auto Transmit                     */
08722    #define TCR_OFST        0x10    /* Collision Offset Enable (be nice) */
08723
08724    /* Bits in dp_tsr */
08725    #define TSR_PTX         0x01    /* Packet Transmitted (without error)*/
08726    #define TSR_DFR         0x02    /* Transmit Deferred                 */
08727    #define TSR_COL         0x04    /* Transmit Collided                 */
08728    #define TSR_ABT         0x08    /* Transmit Aborted                  */
08729    #define TSR_CRS         0x10    /* Carrier Sense Lost                */
08730    #define TSR_FU          0x20    /* FIFO Underrun                     */
08731    #define TSR_CDH         0x40    /* CD Heartbeat                      */
08732    #define TSR_OWC         0x80    /* Out of Window Collision           */
08733
08734    /* Bits in tp_rcr */
08735    #define RCR_SEP         0x01    /* Save Errored Packets              */
08736    #define RCR_AR          0x02    /* Accept Runt Packets               */
08737    #define RCR_AB          0x04    /* Accept Broadcast                  */
08738    #define RCR_AM          0x08    /* Accept Multicast                  */
08739    #define RCR_PRO         0x10    /* Physical Promiscuous              */
08740    #define RCR_MON         0x20    /* Monitor Mode                      */
08741
08742    /* Bits in dp_rsr */
08743    #define RSR_PRX         0x01    /* Packet Received Intact            */
08744    #define RSR_CRC         0x02    /* CRC Error                         */
08745    #define RSR_FAE         0x04    /* Frame Alignment Error             */
08746    #define RSR_FO          0x08    /* FIFO Overrun                      */
08747    #define RSR_MPA         0x10    /* Missed Packet                     */
08748    #define RSR_PHY         0x20    /* Multicast Address Match !!        */
08749    #define RSR_DIS         0x40    /* Receiver Disabled                 */
08750
08751
08752    typedef struct dp_rcvhdr
08753    {
08754            u8_t dr_status;                 /* Copy of rsr                       */
08755            u8_t dr_next;                   /* Pointer to next packet            */
08756            u8_t dr_rbcl;                   /* Receive Byte Count Low            */
08757            u8_t dr_rbch;                   /* Receive Byte Count High           */
08758    } dp_rcvhdr_t;
08759
08760    #define DP_PAGESIZE     256
08761
08762    /* Some macros to simplify accessing the dp8390 */
08763    #define inb_reg0(dep, reg)      (in_byte(dep->de_dp8390_port+reg))
08764    #define outb_reg0(dep, reg, data) (out_byte(dep->de_dp8390_port+reg, data))
08765    #define inb_reg1(dep, reg)      (in_byte (dep->de_dp8390_port+reg))
08766    #define outb_reg1(dep, reg, data) (out_byte(dep->de_dp8390_port+reg, data))
08767
08768    /* Software interface to the dp8390 driver */
08769
08770    struct dpeth;
08771    struct iovec_dat;
08772    _PROTOTYPE( typedef void (*dp_initf_t), (struct dpeth *dep)             );
08773    _PROTOTYPE( typedef void (*dp_stopf_t), (struct dpeth *dep)             );
08774    _PROTOTYPE( typedef void (*dp_user2nicf_t), (struct dpeth *dep,
```

```
08775                           struct iovec_dat *iovp, vir_bytes offset,
08776                           int nic_addr, vir_bytes count)                  );
08777   _PROTOTYPE( typedef void (*dp_nic2userf_t), (struct dpeth *dep,
08778                           int nic_addr, struct iovec_dat *iovp,
08779                           vir_bytes offset, vir_bytes count)              );
08780   #if 0
08781   _PROTOTYPE( typedef void (*dp_getheaderf_t), (struct dpeth *dep,
08782                           int page, struct dp_rcvhdr *h, u16_t *eth_type) );
08783   #endif
08784   _PROTOTYPE( typedef void (*dp_getblock_t), (struct dpeth *dep,
08785                   int page, size_t offset, size_t size, void *dst)        );
08786
08787   /* iovectors are handled IOVEC_NR entries at a time. */
08788   #define IOVEC_NR        16
08789
08790   typedef struct iovec_dat
08791   {
08792     iovec_t iod_iovec[IOVEC_NR];
08793     int iod_iovec_s;
08794     int iod_proc_nr;
08795     vir_bytes iod_iovec_addr;
08796   } iovec_dat_t;
08797
08798   #define SENDQ_NR        2       /* Maximum size of the send queue */
08799   #define SENDQ_PAGES     6       /* 6 * DP_PAGESIZE >= 1514 bytes */
08800
08801   typedef struct dpeth
08802   {
08803           /* The de_base_port field is the starting point of the probe.
08804            * The conf routine also fills de_linmem and de_irq. If the probe
08805            * routine knows the irq and/or memory address because they are
08806            * hardwired in the board, the probe should modify these fields.
08807            * Futhermore, the probe routine should also fill in de_initf and
08808            * de_stopf fields with the appropriate function pointers and set
08809            * de_prog_IO iff programmed I/O is to be used.
08810            */
08811           port_t de_base_port;
08812           phys_bytes de_linmem;
08813           int de_irq;
08814           dp_initf_t de_initf;
08815           dp_stopf_t de_stopf;
08816           int de_prog_IO;
08817
08818           /* The initf function fills the following fields. Only cards that do
08819            * programmed I/O fill in the de_pata_port field.
08820            * In addition, the init routine has to fill in the sendq data
08821            * structures.
08822            */
08823           ether_addr_t de_address;
08824           port_t de_dp8390_port;
08825           port_t de_data_port;
08826           int de_16bit;
08827           int de_ramsize;
08828           int de_offset_page;
08829           int de_startpage;
08830           int de_stoppage;
08831
08832           /* Do it yourself send queue */
08833           struct sendq
08834           {
```

```
08835                   int sq_filled;          /* this buffer contains a packet */
08836                   int sq_size;            /* with this size */
08837                   int sq_sendpage;        /* starting page of the buffer */
08838           } de_sendq[SENDQ_NR];
08839           int de_sendq_nr;
08840           int de_sendq_head;              /* Enqueue at the head */
08841           int de_sendq_tail;              /* Dequeue at the tail */
08842
08843           /* The de_memsegm field is calculated based on the de_linmem field */
08844           segm_t de_memsegm;
08845
08846           /* Fields for internal use by the dp8390 driver. */
08847           int de_flags;
08848           int de_mode;
08849           eth_stat_t de_stat;
08850           iovec_dat_t de_read_iovec;
08851           iovec_dat_t de_write_iovec;
08852           iovec_dat_t de_tmp_iovec;
08853           vir_bytes de_read_s;
08854           int de_client;
08855           message de_sendmsg;
08856           dp_user2nicf_t de_user2nicf;
08857           dp_nic2userf_t de_nic2userf;
08858           dp_getblock_t de_getblockf;
08859   } dpeth_t;
08860
08861   #define DEI_DEFAULT     0x8000
08862
08863   #define DEF_EMPTY       0x000
08864   #define DEF_PACK_SEND   0x001
08865   #define DEF_PACK_RECV   0x002
08866   #define DEF_SEND_AVAIL  0x004
08867   #define DEF_READING     0x010
08868   #define DEF_PROMISC     0x040
08869   #define DEF_MULTI       0x080
08870   #define DEF_BROAD       0x100
08871   #define DEF_ENABLED     0x200
08872   #define DEF_STOPPED     0x400
08873
08874   #define DEM_DISABLED    0x0
08875   #define DEM_SINK        0x1
08876   #define DEM_ENABLED     0x2
08877
08878
08879   /*
08880    * $PchHeader: /mount/hd2/minix/sys/kernel/ibm/RCS/dp8390.h,v 1.4 1995/06/12 18:21:01 philip Exp $
08881    */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/driver.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

08900    /* Types and constants shared between the generic and device dependent
08901     * device driver code.
08902     */
08903
08904    #include <minix/callnr.h>
08905    #include <minix/com.h>
08906    #include "proc.h"
08907    #include <minix/partition.h>
08908
08909    /* Info about and entry points into the device dependent code. */
08910    struct driver {
08911      _PROTOTYPE( char *(*dr_name), (void) );
08912      _PROTOTYPE( int (*dr_open), (struct driver *dp, message *m_ptr) );
08913      _PROTOTYPE( int (*dr_close), (struct driver *dp, message *m_ptr) );
08914      _PROTOTYPE( int (*dr_ioctl), (struct driver *dp, message *m_ptr) );
08915      _PROTOTYPE( struct device *(*dr_prepare), (int device) );
08916      _PROTOTYPE( int (*dr_schedule), (int proc_nr, struct iorequest_s *request) );
08917      _PROTOTYPE( int (*dr_finish), (void) );
08918      _PROTOTYPE( void (*dr_cleanup), (void) );
08919      _PROTOTYPE( void (*dr_geometry), (struct partition *entry) );
08920    };
08921
08922    #if (CHIP == INTEL)
08923
08924    /* Number of bytes you can DMA before hitting a 64K boundary: */
08925    #define dma_bytes_left(phys)    \
08926       ((unsigned) (sizeof(int) == 2 ? 0 : 0x10000) - (unsigned) ((phys) & 0xFFFF))
08927
08928    #endif /* CHIP == INTEL */
08929
08930    /* Base and size of a partition in bytes. */
08931    struct device {
08932      unsigned long dv_base;
08933      unsigned long dv_size;
08934    };
08935
08936    #define NIL_DEV         ((struct device *) 0)
08937
08938    /* Functions defined by driver.c: */
08939    _PROTOTYPE( void driver_task, (struct driver *dr) );
08940    _PROTOTYPE( int do_rdwt, (struct driver *dr, message *m_ptr) );
08941    _PROTOTYPE( int do_vrdwt, (struct driver *dr, message *m_ptr) );
08942    _PROTOTYPE( char *no_name, (void) );
08943    _PROTOTYPE( int do_nop, (struct driver *dp, message *m_ptr) );
08944    _PROTOTYPE( int nop_finish, (void) );
08945    _PROTOTYPE( void nop_cleanup, (void) );
08946    _PROTOTYPE( void clock_mess, (int ticks, watchdog_t func) );
08947    _PROTOTYPE( int do_diocntl, (struct driver *dr, message *m_ptr) );
08948
08949    /* Parameters for the disk drive. */
08950    #define SECTOR_SIZE      512    /* physical sector size in bytes */
08951    #define SECTOR_SHIFT       9    /* for division */
08952    #define SECTOR_MASK      511    /* and remainder */
08953
08954    /* Size of the DMA buffer buffer in bytes. */
```

```
08955    #define DMA_BUF_SIZE    (DMA_SECTORS * SECTOR_SIZE)
08956
08957    #if (CHIP == INTEL)
08958    extern u8_t *tmp_buf;                  /* the DMA buffer */
08959    #else
08960    extern u8_t tmp_buf[];                 /* the DMA buffer */
08961    #endif
08962    extern phys_bytes tmp_phys;            /* phys address of DMA buffer */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/drvlib.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
09000    /* IBM device driver definitions                        Author: Kees J. Bot
09001     *                                                             7 Dec 1995
09002     */
09003
09004    #include <ibm/partition.h>
09005
09006    _PROTOTYPE( void partition, (struct driver *dr, int device, int style) );
09007
09008    /* BIOS parameter table layout. */
09009    #define bp_cylinders(t)         (* (u16_t *) (&(t)[0]))
09010    #define bp_heads(t)             (* (u8_t *)  (&(t)[2]))
09011    #define bp_reduced_wr(t)        (* (u16_t *) (&(t)[3]))
09012    #define bp_precomp(t)           (* (u16_t *) (&(t)[5]))
09013    #define bp_max_ecc(t)           (* (u8_t *)  (&(t)[7]))
09014    #define bp_ctlbyte(t)           (* (u8_t *)  (&(t)[8]))
09015    #define bp_landingzone(t)       (* (u16_t *) (&(t)[12]))
09016    #define bp_sectors(t)           (* (u8_t *)  (&(t)[14]))
09017
09018    /* Miscellaneous. */
09019    #define DEV_PER_DRIVE   (1 + NR_PARTITIONS)
09020    #define MINOR_hd1a      128
09021    #define MINOR_fd0a      (28<<2)
09022    #define P_FLOPPY        0
09023    #define P_PRIMARY       1
09024    #define P_SUB           2
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/glo.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
09100    /* Global variables used in the kernel. */
09101
09102    /* EXTERN is defined as extern except in table.c. */
09103    #ifdef _TABLE
09104    #undef EXTERN
09105    #define EXTERN
09106    #endif
09107
09108    /* Kernel memory. */
09109    EXTERN phys_bytes code_base;    /* base of kernel code */
```

```
09110    EXTERN phys_bytes data_base;    /* base of kernel data */
09111
09112    /* Low level interrupt communications. */
09113    EXTERN struct proc *held_head;  /* head of queue of held-up interrupts */
09114    EXTERN struct proc *held_tail;  /* tail of queue of held-up interrupts */
09115    EXTERN unsigned char k_reenter; /* kernel reentry count (entry count less 1)*/
09116
09117    /* Process table.  Here to stop too many things having to include proc.h. */
09118    EXTERN struct proc *proc_ptr;   /* pointer to currently running process */
09119
09120    /* Signals. */
09121    EXTERN int sig_procs;           /* number of procs with p_pending != 0 */
09122
09123    /* Memory sizes. */
09124    EXTERN struct memory mem[NR_MEMS];      /* base and size of chunks of memory */
09125    EXTERN phys_clicks tot_mem_size;        /* total system memory size */
09126
09127    /* Miscellaneous. */
09128    extern u16_t sizes[];           /* table filled in by boot monitor */
09129    extern struct tasktab tasktab[];/* initialized in table.c, so extern here */
09130    extern char *t_stack[];         /* initialized in table.c, so extern here */
09131    EXTERN unsigned lost_ticks;     /* clock ticks counted outside the clock task */
09132    EXTERN clock_t tty_timeout;     /* time to wake up the TTY task */
09133    EXTERN int current;             /* currently visible console */
09134
09135    #if (CHIP == INTEL)
09136
09137    /* Machine type. */
09138    EXTERN int pc_at;               /* PC-AT compatible hardware interface */
09139    EXTERN int ps_mca;              /* PS/2 with Micro Channel */
09140    EXTERN unsigned int processor;  /* 86, 186, 286, 386, ... */
09141    #if _WORD_SIZE == 2
09142    EXTERN int protected_mode;      /* nonzero if running in Intel protected mode*/
09143    #else
09144    #define protected_mode  1       /* 386 mode implies protected mode */
09145    #endif
09146
09147    /* Video card types. */
09148    EXTERN int ega;                 /* nonzero if console is EGA */
09149    EXTERN int vga;                 /* nonzero if console is VGA */
09150
09151    /* Memory sizes. */
09152    EXTERN unsigned ext_memsize;    /* initialized by assembler startup code */
09153    EXTERN unsigned low_memsize;
09154
09155    /* Miscellaneous. */
09156    EXTERN irq_handler_t irq_table[NR_IRQ_VECTORS];
09157    EXTERN int irq_use;             /* bit map of all in-use irq's */
09158    EXTERN reg_t mon_ss, mon_sp;    /* monitor stack */
09159    EXTERN int mon_return;          /* true if return to the monitor possible */
09160    EXTERN phys_bytes reboot_code;  /* program for the boot monitor */
09161
09162    /* Variables that are initialized elsewhere are just extern here. */
09163    extern struct segdesc_s gdt[];  /* global descriptor table for protected mode*/
09164
09165    EXTERN _PROTOTYPE( void (*level0_func), (void) );
09166    #endif /* (CHIP == INTEL) */
09167
09168    #if (CHIP == M68000)
09169    /* Variables that are initialized elsewhere are just extern here. */
```

```
09170    extern int keypad;              /* Flag for keypad mode */
09171    extern int app_mode;            /* Flag for arrow key application mode */
09172    extern int STdebKey;            /* nonzero if ctl-alt-Fx detected */
09173    extern struct tty *cur_cons;    /* virtual cons currently displayed */
09174    extern unsigned char font8[];   /* 8 pixel wide font table (initialized) */
09175    extern unsigned char font12[];  /* 12 pixel wide font table (initialized) */
09176    extern unsigned char font16[];  /* 16 pixel wide font table (initialized) */
09177    extern unsigned short resolution; /* screen res; ST_RES_LOW..TT_RES_HIGH */
09178    #endif
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/kernel.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
09200    /* This is the master header for the kernel.  It includes some other files
09201     * and defines the principal constants.
09202     */
09203    #define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
09204    #define _MINIX             1    /* tell headers to include MINIX stuff */
09205    #define _SYSTEM            1    /* tell headers that this is the kernel */
09206
09207    /* The following are so basic, all the *.c files get them automatically. */
09208    #include <minix/config.h>       /* MUST be first */
09209    #include <ansi.h>               /* MUST be second */
09210    #include <sys/types.h>
09211    #include <minix/const.h>
09212    #include <minix/type.h>
09213    #include <minix/syslib.h>
09214
09215    #include <string.h>
09216    #include <limits.h>
09217    #include <errno.h>
09218
09219    #include "const.h"
09220    #include "type.h"
09221    #include "proto.h"
09222    #include "glo.h"
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/ne2000.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
09300    /*
09301    ne2000.h
09302
09303    Created:        March 15, 1994 by Philip Homburg <philip@cs.vu.nl>
09304    */
09305
09306    #ifndef NE2000_H
09307    #define NE2000_H
09308
09309    #define NE_DP8390       0x00
09310    #define NE_DATA         0x10
09311    #define NE_RESET        0x1F
09312
09313    #define NE1000_START    0x2000
09314    #define NE1000_SIZE     0x2000
```

```
09315    #define NE2000_START    0x4000
09316    #define NE2000_SIZE     0x4000
09317
09318    #define inb_ne(dep, reg) (in_byte(dep->de_base_port+reg))
09319    #define outb_ne(dep, reg, data) (out_byte(dep->de_base_port+reg, data))
09320    #define inw_ne(dep, reg) (in_word(dep->de_base_port+reg))
09321    #define outw_ne(dep, reg, data) (out_word(dep->de_base_port+reg, data))
09322
09323    #endif /* NE2000_H */
09324
09325    /*
09326     * $PchHeader: /mount/hd2/minix/sys/kernel/ibm/RCS/ne2000.h,v 1.1 1994/04/07 22:58:48 philip Exp $
09327     */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/proc.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

09400    #ifndef PROC_H
09401    #define PROC_H
09402
09403    /* Here is the declaration of the process table.  It contains the process'
09404     * registers, memory map, accounting, and message send/receive information.
09405     * Many assembly code routines reference fields in it.  The offsets to these
09406     * fields are defined in the assembler include file sconst.h.  When changing
09407     * 'proc', be sure to change sconst.h to match.
09408     */
09409
09410    struct proc {
09411      struct stackframe_s p_reg;    /* process' registers saved in stack frame */
09412
09413    #if (CHIP == INTEL)
09414      reg_t p_ldt_sel;              /* selector in gdt giving ldt base and limit*/
09415      struct segdesc_s p_ldt[2];    /* local descriptors for code and data */
09416                                    /* 2 is LDT_SIZE - avoid include protect.h */
09417    #endif /* (CHIP == INTEL) */
09418
09419    #if (CHIP == M68000)
09420      reg_t p_splow;                /* lowest observed stack value */
09421      int p_trap;                   /* trap type (only low byte) */
09422    #if (SHADOWING == 0)
09423      char *p_crp;                  /* mmu table pointer (really struct _rpr *) */
09424    #else
09425      phys_clicks p_shadow;         /* set if shadowed process image */
09426      int align;                    /* make the struct size a multiple of 4 */
09427    #endif
09428      int p_nflips;                 /* statistics */
09429      char p_physio;                /* cannot be (un)shadowed now if set */
09430    #if defined(FPP)
09431      struct fsave p_fsave;         /* FPP state frame and registers */
09432      int align2;                   /* make the struct size a multiple of 4 */
09433    #endif
09434    #endif /* (CHIP == M68000) */
09435
09436      reg_t *p_stguard;             /* stack guard word */
09437
09438      int p_nr;                     /* number of this process (for fast access) */
09439
```

```
09440       int p_int_blocked;            /* nonzero if int msg blocked by busy task */
09441       int p_int_held;               /* nonzero if int msg held by busy syscall */
09442       struct proc *p_nextheld;      /* next in chain of held-up int processes */
09443
09444       int p_flags;                  /* P_SLOT_FREE, SENDING, RECEIVING, etc. */
09445       struct mem_map p_map[NR_SEGS];/* memory map */
09446       pid_t p_pid;                  /* process id passed in from MM */
09447
09448       clock_t user_time;            /* user time in ticks */
09449       clock_t sys_time;             /* sys time in ticks */
09450       clock_t child_utime;          /* cumulative user time of children */
09451       clock_t child_stime;          /* cumulative sys time of children */
09452       clock_t p_alarm;              /* time of next alarm in ticks, or 0 */
09453
09454       struct proc *p_callerq;       /* head of list of procs wishing to send */
09455       struct proc *p_sendlink;      /* link to next proc wishing to send */
09456       message *p_messbuf;           /* pointer to message buffer */
09457       int p_getfrom;                /* from whom does process want to receive? */
09458       int p_sendto;
09459
09460       struct proc *p_nextready;     /* pointer to next ready process */
09461       sigset_t p_pending;           /* bit map for pending signals */
09462       unsigned p_pendcount;         /* count of pending and unfinished signals */
09463
09464       char p_name[16];              /* name of the process */
09465   };
09466
09467   /* Guard word for task stacks. */
09468   #define STACK_GUARD     ((reg_t) (sizeof(reg_t) == 2 ? 0xBEEF : 0xDEADBEEF))
09469
09470   /* Bits for p_flags in proc[].  A process is runnable iff p_flags == 0. */
09471   #define P_SLOT_FREE      001    /* set when slot is not in use */
09472   #define NO_MAP           002    /* keeps unmapped forked child from running */
09473   #define SENDING          004    /* set when process blocked trying to send */
09474   #define RECEIVING        010    /* set when process blocked trying to recv */
09475   #define PENDING          020    /* set when inform() of signal pending */
09476   #define SIG_PENDING      040    /* keeps to-be-signalled proc from running */
09477   #define P_STOP          0100    /* set when process is being traced */
09478
09479   /* Magic process table addresses. */
09480   #define BEG_PROC_ADDR (&proc[0])
09481   #define END_PROC_ADDR (&proc[NR_TASKS + NR_PROCS])
09482   #define END_TASK_ADDR (&proc[NR_TASKS])
09483   #define BEG_SERV_ADDR (&proc[NR_TASKS])
09484   #define BEG_USER_ADDR (&proc[NR_TASKS + LOW_USER])
09485
09486   #define NIL_PROC          ((struct proc *) 0)
09487   #define isidlehardware(n) ((n) == IDLE || (n) == HARDWARE)
09488   #define isokprocn(n)      ((unsigned) ((n) + NR_TASKS) < NR_PROCS + NR_TASKS)
09489   #define isoksrc_dest(n)   (isokprocn(n) || (n) == ANY)
09490   #define isoksusern(n)     ((unsigned) (n) < NR_PROCS)
09491   #define isokusern(n)      ((unsigned) ((n) - LOW_USER) < NR_PROCS - LOW_USER)
09492   #define isrxhardware(n)   ((n) == ANY || (n) == HARDWARE)
09493   #define issysentn(n)      ((n) == FS_PROC_NR || (n) == MM_PROC_NR)
09494   #define istaskp(p)        ((p) < END_TASK_ADDR && (p) != proc_addr(IDLE))
09495   #define isuserp(p)        ((p) >= BEG_USER_ADDR)
09496   #define proc_addr(n)      (pproc_addr + NR_TASKS)[(n)]
09497   #define cproc_addr(n)     (&(proc + NR_TASKS)[(n)])
09498   #define proc_number(p)    ((p)->p_nr)
09499   #define proc_vir2phys(p, vir) \
```

```
09500                           (((phys_bytes)(p)->p_map[D].mem_phys << CLICK_SHIFT) \
09501                                                   + (vir_bytes) (vir))
09502   #if (SHADOWING == 1)
09503   #define isshadowp(p)      ((p)->p_shadow != 0)
09504   #endif
09505
09506   EXTERN struct proc proc[NR_TASKS + NR_PROCS];   /* process table */
09507   EXTERN struct proc *pproc_addr[NR_TASKS + NR_PROCS];
09508   /* ptrs to process table slots; fast because now a process entry can be found
09509      by indexing the pproc_addr array, while accessing an element i requires
09510      a multiplication with sizeof(struct proc) to determine the address */
09511   EXTERN struct proc *bill_ptr;   /* ptr to process to bill for clock ticks */
09512   EXTERN struct proc *rdy_head[NQ];       /* pointers to ready list headers */
09513   EXTERN struct proc *rdy_tail[NQ];       /* pointers to ready list tails */
09514
09515   #endif /* PROC_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/protect.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

09600   /* Constants for protected mode. */
09601
09602   /* Table sizes. */
09603   #define GDT_SIZE (FIRST_LDT_INDEX + NR_TASKS + NR_PROCS) /* spec. and LDT's */
09604   #define IDT_SIZE (IRQ8_VECTOR + 8)      /* only up to the highest vector */
09605   #define LDT_SIZE         2      /* contains CS and DS only */
09606
09607   /* Fixed global descriptors.  1 to 7 are prescribed by the BIOS. */
09608   #define GDT_INDEX        1      /* GDT descriptor */
09609   #define IDT_INDEX        2      /* IDT descriptor */
09610   #define DS_INDEX         3      /* kernel DS */
09611   #define ES_INDEX         4      /* kernel ES (386: flag 4 Gb at startup) */
09612   #define SS_INDEX         5      /* kernel SS (386: monitor SS at startup) */
09613   #define CS_INDEX         6      /* kernel CS */
09614   #define MON_CS_INDEX     7      /* temp for BIOS (386: monitor CS at startup) */
09615   #define TSS_INDEX        8      /* kernel TSS */
09616   #define DS_286_INDEX     9      /* scratch 16-bit source segment */
09617   #define ES_286_INDEX    10      /* scratch 16-bit destination segment */
09618   #define VIDEO_INDEX     11      /* video memory segment */
09619   #define DP_ETH0_INDEX   12      /* Western Digital Etherplus buffer */
09620   #define DP_ETH1_INDEX   13      /* Western Digital Etherplus buffer */
09621   #define FIRST_LDT_INDEX 14      /* rest of descriptors are LDT's */
09622
09623   #define GDT_SELECTOR      0x08  /* (GDT_INDEX * DESC_SIZE) bad for asld */
09624   #define IDT_SELECTOR      0x10  /* (IDT_INDEX * DESC_SIZE) */
09625   #define DS_SELECTOR       0x18  /* (DS_INDEX * DESC_SIZE) */
09626   #define ES_SELECTOR       0x20  /* (ES_INDEX * DESC_SIZE) */
09627   #define FLAT_DS_SELECTOR  0x21  /* less privileged ES */
09628   #define SS_SELECTOR       0x28  /* (SS_INDEX * DESC_SIZE) */
09629   #define CS_SELECTOR       0x30  /* (CS_INDEX * DESC_SIZE) */
09630   #define MON_CS_SELECTOR   0x38  /* (MON_CS_INDEX * DESC_SIZE) */
09631   #define TSS_SELECTOR      0x40  /* (TSS_INDEX * DESC_SIZE) */
09632   #define DS_286_SELECTOR   0x49  /* (DS_286_INDEX * DESC_SIZE + 1) */
09633   #define ES_286_SELECTOR   0x51  /* (ES_286_INDEX * DESC_SIZE + 1) */
09634   #define VIDEO_SELECTOR    0x59  /* (VIDEO_INDEX * DESC_SIZE + 1) */
```

```
09635    #define DP_ETH0_SELECTOR  0x61  /* (DP_ETH0_INDEX * DESC_SIZE) */
09636    #define DP_ETH1_SELECTOR  0x69  /* (DP_ETH1_INDEX * DESC_SIZE) */
09637
09638    /* Fixed local descriptors. */
09639    #define CS_LDT_INDEX     0      /* process CS */
09640    #define DS_LDT_INDEX     1      /* process DS=ES=FS=GS=SS */
09641
09642    /* Privileges. */
09643    #define INTR_PRIVILEGE   0      /* kernel and interrupt handlers */
09644    #define TASK_PRIVILEGE   1
09645    #define USER_PRIVILEGE   3
09646
09647    /* 286 hardware constants. */
09648
09649    /* Exception vector numbers. */
09650    #define BOUNDS_VECTOR       5   /* bounds check failed */
09651    #define INVAL_OP_VECTOR     6   /* invalid opcode */
09652    #define COPROC_NOT_VECTOR   7   /* coprocessor not available */
09653    #define DOUBLE_FAULT_VECTOR 8
09654    #define COPROC_SEG_VECTOR   9   /* coprocessor segment overrun */
09655    #define INVAL_TSS_VECTOR   10   /* invalid TSS */
09656    #define SEG_NOT_VECTOR     11   /* segment not present */
09657    #define STACK_FAULT_VECTOR 12   /* stack exception */
09658    #define PROTECTION_VECTOR  13   /* general protection */
09659
09660    /* Selector bits. */
09661    #define TI            0x04      /* table indicator */
09662    #define RPL           0x03      /* requester privilege level */
09663
09664    /* Descriptor structure offsets. */
09665    #define DESC_BASE        2      /* to base_low */
09666    #define DESC_BASE_MIDDLE 4      /* to base_middle */
09667    #define DESC_ACCESS      5      /* to access byte */
09668    #define DESC_SIZE        8      /* sizeof (struct segdesc_s) */
09669
09670    /* Segment sizes. */
09671    #define MAX_286_SEG_SIZE 0x10000L
09672
09673    /* Base and limit sizes and shifts. */
09674    #define BASE_MIDDLE_SHIFT   16  /* shift for base --> base_middle */
09675
09676    /* Access-byte and type-byte bits. */
09677    #define PRESENT       0x80      /* set for descriptor present */
09678    #define DPL           0x60      /* descriptor privilege level mask */
09679    #define DPL_SHIFT        5
09680    #define SEGMENT       0x10      /* set for segment-type descriptors */
09681
09682    /* Access-byte bits. */
09683    #define EXECUTABLE    0x08      /* set for executable segment */
09684    #define CONFORMING    0x04      /* set for conforming segment if executable */
09685    #define EXPAND_DOWN   0x04      /* set for expand-down segment if !executable*/
09686    #define READABLE      0x02      /* set for readable segment if executable */
09687    #define WRITEABLE     0x02      /* set for writeable segment if !executable */
09688    #define TSS_BUSY      0x02      /* set if TSS descriptor is busy */
09689    #define ACCESSED      0x01      /* set if segment accessed */
09690
09691    /* Special descriptor types. */
09692    #define AVL_286_TSS      1      /* available 286 TSS */
09693    #define LDT              2      /* local descriptor table */
09694    #define BUSY_286_TSS     3      /* set transparently to the software */
```

```
09695    #define CALL_286_GATE    4      /* not used */
09696    #define TASK_GATE        5      /* only used by debugger */
09697    #define INT_286_GATE     6      /* interrupt gate, used for all vectors */
09698    #define TRAP_286_GATE    7      /* not used */
09699
09700    /* Extra 386 hardware constants. */
09701
09702    /* Exception vector numbers. */
09703    #define PAGE_FAULT_VECTOR   14
09704    #define COPROC_ERR_VECTOR   16  /* coprocessor error */
09705
09706    /* Descriptor structure offsets. */
09707    #define DESC_GRANULARITY     6  /* to granularity byte */
09708    #define DESC_BASE_HIGH       7  /* to base_high */
09709
09710    /* Base and limit sizes and shifts. */
09711    #define BASE_HIGH_SHIFT     24  /* shift for base --> base_high */
09712    #define BYTE_GRAN_MAX   0xFFFFFL   /* maximum size for byte granular segment */
09713    #define GRANULARITY_SHIFT   16  /* shift for limit --> granularity */
09714    #define OFFSET_HIGH_SHIFT   16  /* shift for (gate) offset --> offset_high */
09715    #define PAGE_GRAN_SHIFT     12  /* extra shift for page granular limits */
09716
09717    /* Type-byte bits. */
09718    #define DESC_386_BIT      0x08  /* 386 types are obtained by ORing with this */
09719                                    /* LDT's and TASK_GATE's don't need it */
09720
09721    /* Granularity byte. */
09722    #define GRANULAR          0x80  /* set for 4K granularilty */
09723    #define DEFAULT           0x40  /* set for 32-bit defaults (executable seg) */
09724    #define BIG               0x40  /* set for "BIG" (expand-down seg) */
09725    #define AVL               0x10  /* 0 for available */
09726    #define LIMIT_HIGH        0x0F  /* mask for high bits of limit */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/proto.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
09800    /* Function prototypes. */
09801
09802    #ifndef PROTO_H
09803    #define PROTO_H
09804
09805    /* Struct declarations. */
09806    struct dpeth;
09807    struct proc;
09808    struct tty;
09809
09810    /* at_wini.c, bios_wini.c, esdi_wini.c, ps_wini.c, xt_wini.c, wini.c */
09811    _PROTOTYPE( void winchester_task, (void)                                );
09812    _PROTOTYPE( void at_winchester_task, (void)                             );
09813    _PROTOTYPE( void bios_winchester_task, (void)                           );
09814    _PROTOTYPE( void esdi_winchester_task, (void)                           );
09815    _PROTOTYPE( void ps_winchester_task, (void)                             );
09816    _PROTOTYPE( void xt_winchester_task, (void)                             );
09817
09818    /* aha_scsi.c */
09819    _PROTOTYPE( void aha_scsi_task, (void)                                  );
```

```
09820
09821   /* clock.c */
09822   _PROTOTYPE( void clock_task, (void)                                     );
09823   _PROTOTYPE( void clock_stop, (void)                                     );
09824   _PROTOTYPE( clock_t get_uptime, (void)                                  );
09825   _PROTOTYPE( void syn_alrm_task, (void)                                  );
09826
09827   /* dmp.c */
09828   _PROTOTYPE( void map_dmp, (void)                                        );
09829   _PROTOTYPE( void p_dmp, (void)                                          );
09830   _PROTOTYPE( void reg_dmp, (struct proc *rp)                             );
09831
09832   /* dp8390.c */
09833   _PROTOTYPE( void dp8390_task, (void)                                    );
09834   _PROTOTYPE( void dp_dump, (void)                                        );
09835   _PROTOTYPE( void dp8390_stop, (void)                                    );
09836
09837   /* floppy.c, stfloppy.c */
09838   _PROTOTYPE( void floppy_task, (void)                                    );
09839   _PROTOTYPE( void floppy_stop, (void)                                    );
09840
09841   /* main.c, stmain.c */
09842   _PROTOTYPE( void main, (void)                                           );
09843   _PROTOTYPE( void panic, (const char *s, int n)                          );
09844
09845   /* mcd.c */
09846   _PROTOTYPE( void mcd_task, (void)                                       );
09847
09848   /* memory.c */
09849   _PROTOTYPE( void mem_task, (void)                                       );
09850
09851   /* misc.c */
09852   _PROTOTYPE( int env_parse, (char *env, char *fmt, int field,
09853                           long *param, long min, long max)                );
09854
09855   /* printer.c, stprint.c */
09856   _PROTOTYPE( void printer_task, (void)                                   );
09857
09858   /* proc.c */
09859   _PROTOTYPE( void interrupt, (int task)                                  );
09860   _PROTOTYPE( int lock_mini_send, (struct proc *caller_ptr,
09861                       int dest, message *m_ptr)                           );
09862   _PROTOTYPE( void lock_pick_proc, (void)                                 );
09863   _PROTOTYPE( void lock_ready, (struct proc *rp)                          );
09864   _PROTOTYPE( void lock_sched, (void)                                     );
09865   _PROTOTYPE( void lock_unready, (struct proc *rp)                        );
09866   _PROTOTYPE( int sys_call, (int function, int src_dest, message *m_ptr)  );
09867   _PROTOTYPE( void unhold, (void)                                         );
09868
09869   /* rs232.c */
09870   _PROTOTYPE( void rs_init, (struct tty *tp)                              );
09871
09872   /* sb16_dsp.c */
09873   _PROTOTYPE( void dsp_task, (void)                                       );
09874
09875   /* sb16_mixer.c */
09876   _PROTOTYPE( void mixer_task, (void)                                     );
09877
09878   /* system.c */
09879   _PROTOTYPE( void cause_sig, (int proc_nr, int sig_nr)                   );
```

```
09880   _PROTOTYPE( void inform, (void)                                         );
09881   _PROTOTYPE( phys_bytes numap, (int proc_nr, vir_bytes vir_addr,
09882                   vir_bytes bytes)                                        );
09883   _PROTOTYPE( void sys_task, (void)                                       );
09884   _PROTOTYPE( phys_bytes umap, (struct proc *rp, int seg, vir_bytes vir_addr,
09885                   vir_bytes bytes)                                        );
09886
09887   /* tty.c */
09888   _PROTOTYPE( void handle_events, (struct tty *tp)                        );
09889   _PROTOTYPE( void sigchar, (struct tty *tp, int sig)                     );
09890   _PROTOTYPE( void tty_task, (void)                                       );
09891   _PROTOTYPE( int in_process, (struct tty *tp, char *buf, int count)      );
09892   _PROTOTYPE( void out_process, (struct tty *tp, char *bstart, char *bpos,
09893                                   char *bend, int *icount, int *ocount)   );
09894   _PROTOTYPE( void tty_wakeup, (clock_t now)                              );
09895   _PROTOTYPE( void tty_reply, (int code, int replyee, int proc_nr,
09896                                                   int status)             );
09897   _PROTOTYPE( void tty_devnop, (struct tty *tp)                           );
09898
09899   /* library */
09900   _PROTOTYPE( void *memcpy, (void *_s1, const void *_s2, size_t _n)       );
09901
09902   #if (CHIP == INTEL)
09903
09904   /* clock.c */
09905   _PROTOTYPE( void milli_start, (struct milli_state *msp)                 );
09906   _PROTOTYPE( unsigned milli_elapsed, (struct milli_state *msp)           );
09907   _PROTOTYPE( void milli_delay, (unsigned millisec)                       );
09908
09909   /* console.c */
09910   _PROTOTYPE( void cons_stop, (void)                                      );
09911   _PROTOTYPE( void putk, (int c)                                          );
09912   _PROTOTYPE( void scr_init, (struct tty *tp)                             );
09913   _PROTOTYPE( void toggle_scroll, (void)                                  );
09914   _PROTOTYPE( int con_loadfont, (phys_bytes user_phys)                    );
09915   _PROTOTYPE( void select_console, (int cons_line)                        );
09916
09917   /* cstart.c */
09918   _PROTOTYPE( void cstart, (U16_t cs, U16_t ds, U16_t mcs, U16_t mds,
09919                                   U16_t parmoff, U16_t parmsize)          );
09920   _PROTOTYPE( char *k_getenv, (char *name)                                );
09921
09922   /* exception.c */
09923   _PROTOTYPE( void exception, (unsigned vec_nr)                           );
09924
09925   /* i8259.c */
09926   _PROTOTYPE( irq_handler_t get_irq_handler, (int irq)                    );
09927   _PROTOTYPE( void put_irq_handler, (int irq, irq_handler_t handler)      );
09928   _PROTOTYPE( void intr_init, (int mine)                                  );
09929
09930   /* keyboard.c */
09931   _PROTOTYPE( void kb_init, (struct tty *tp)                              );
09932   _PROTOTYPE( int kbd_loadmap, (phys_bytes user_phys)                     );
09933   _PROTOTYPE( void wreboot, (int how)                                     );
09934
09935   /* klib*.s */
09936   _PROTOTYPE( void bios13, (void)                                         );
09937   _PROTOTYPE( phys_bytes check_mem, (phys_bytes base, phys_bytes size)    );
09938   _PROTOTYPE( void cp_mess, (int src,phys_clicks src_clicks,vir_bytes src_offset,
09939                   phys_clicks dst_clicks, vir_bytes dst_offset)           );
```

```
09940    _PROTOTYPE( int in_byte, (port_t port)                                  );
09941    _PROTOTYPE( int in_word, (port_t port)                                  );
09942    _PROTOTYPE( void lock, (void)                                           );
09943    _PROTOTYPE( void unlock, (void)                                         );
09944    _PROTOTYPE( void enable_irq, (unsigned irq)                             );
09945    _PROTOTYPE( int disable_irq, (unsigned irq)                             );
09946    _PROTOTYPE( u16_t mem_rdw, (segm_t segm, vir_bytes offset)              );
09947    _PROTOTYPE( void out_byte, (port_t port, int value)                     );
09948    _PROTOTYPE( void out_word, (port_t port, int value)                     );
09949    _PROTOTYPE( void phys_copy, (phys_bytes source, phys_bytes dest,
09950                    phys_bytes count)                                       );
09951    _PROTOTYPE( void port_read, (unsigned port, phys_bytes destination,
09952                    unsigned bytcount)                                      );
09953    _PROTOTYPE( void port_read_byte, (unsigned port, phys_bytes destination,
09954                    unsigned bytcount)                                      );
09955    _PROTOTYPE( void port_write, (unsigned port, phys_bytes source,
09956                    unsigned bytcount)                                      );
09957    _PROTOTYPE( void port_write_byte, (unsigned port, phys_bytes source,
09958                    unsigned bytcount)                                      );
09959    _PROTOTYPE( void reset, (void)                                          );
09960    _PROTOTYPE( void vid_vid_copy, (unsigned src, unsigned dst, unsigned count));
09961    _PROTOTYPE( void mem_vid_copy, (u16_t *src, unsigned dst, unsigned count));
09962    _PROTOTYPE( void level0, (void (*func)(void))                           );
09963    _PROTOTYPE( void monitor, (void)                                        );
09964
09965    /* misc.c */
09966    _PROTOTYPE( void mem_init, (void)                                       );
09967
09968    /* mpx*.s */
09969    _PROTOTYPE( void idle_task, (void)                                      );
09970    _PROTOTYPE( void restart, (void)                                        );
09971
09972    /* The following are never called from C (pure asm procs). */
09973
09974    /* Exception handlers (real or protected mode), in numerical order. */
09975    void _PROTOTYPE( int00, (void) ), _PROTOTYPE( divide_error, (void) );
09976    void _PROTOTYPE( int01, (void) ), _PROTOTYPE( single_step_exception, (void) );
09977    void _PROTOTYPE( int02, (void) ), _PROTOTYPE( nmi, (void) );
09978    void _PROTOTYPE( int03, (void) ), _PROTOTYPE( breakpoint_exception, (void) );
09979    void _PROTOTYPE( int04, (void) ), _PROTOTYPE( overflow, (void) );
09980    void _PROTOTYPE( int05, (void) ), _PROTOTYPE( bounds_check, (void) );
09981    void _PROTOTYPE( int06, (void) ), _PROTOTYPE( inval_opcode, (void) );
09982    void _PROTOTYPE( int07, (void) ), _PROTOTYPE( copr_not_available, (void) );
09983    void                              _PROTOTYPE( double_fault, (void) );
09984    void                              _PROTOTYPE( copr_seg_overrun, (void) );
09985    void                              _PROTOTYPE( inval_tss, (void) );
09986    void                              _PROTOTYPE( segment_not_present, (void) );
09987    void                              _PROTOTYPE( stack_exception, (void) );
09988    void                              _PROTOTYPE( general_protection, (void) );
09989    void                              _PROTOTYPE( page_fault, (void) );
09990    void                              _PROTOTYPE( copr_error, (void) );
09991
09992    /* Hardware interrupt handlers. */
09993    _PROTOTYPE( void hwint00, (void) );
09994    _PROTOTYPE( void hwint01, (void) );
09995    _PROTOTYPE( void hwint02, (void) );
09996    _PROTOTYPE( void hwint03, (void) );
09997    _PROTOTYPE( void hwint04, (void) );
09998    _PROTOTYPE( void hwint05, (void) );
09999    _PROTOTYPE( void hwint06, (void) );
```

```
10000   _PROTOTYPE( void hwint07, (void) );
10001   _PROTOTYPE( void hwint08, (void) );
10002   _PROTOTYPE( void hwint09, (void) );
10003   _PROTOTYPE( void hwint10, (void) );
10004   _PROTOTYPE( void hwint11, (void) );
10005   _PROTOTYPE( void hwint12, (void) );
10006   _PROTOTYPE( void hwint13, (void) );
10007   _PROTOTYPE( void hwint14, (void) );
10008   _PROTOTYPE( void hwint15, (void) );
10009
10010   /* Software interrupt handlers, in numerical order. */
10011   _PROTOTYPE( void trp, (void) );
10012   _PROTOTYPE( void s_call, (void) ), _PROTOTYPE( p_s_call, (void) );
10013   _PROTOTYPE( void level0_call, (void) );
10014
10015   /* ne2000.c */
10016   _PROTOTYPE( int ne_probe, (struct dpeth *dep)                          );
10017
10018   /* printer.c */
10019   _PROTOTYPE( void pr_restart, (void)                                    );
10020
10021   /* protect.c */
10022   _PROTOTYPE( void prot_init, (void)                                     );
10023   _PROTOTYPE( void init_codeseg, (struct segdesc_s *segdp, phys_bytes base,
10024                   phys_bytes size, int privilege)                        );
10025   _PROTOTYPE( void init_dataseg, (struct segdesc_s *segdp, phys_bytes base,
10026                   phys_bytes size, int privilege)                        );
10027   _PROTOTYPE( phys_bytes seg2phys, (U16_t seg)                           );
10028   _PROTOTYPE( void enable_iop, (struct proc *pp)                         );
10029
10030   /* pty.c */
10031   _PROTOTYPE( void do_pty, (struct tty *tp, message *m_ptr)              );
10032   _PROTOTYPE( void pty_init, (struct tty *tp)                            );
10033
10034   /* system.c */
10035   _PROTOTYPE( void alloc_segments, (struct proc *rp)                     );
10036
10037   /* wdeth.c */
10038   _PROTOTYPE( int wdeth_probe, (struct dpeth *dep)                       );
10039
10040   #endif /* (CHIP == INTEL) */
10041
10042   #if (CHIP == M68000)
10043
10044   /* cstart.c */
10045   _PROTOTYPE( void cstart, (char *parmoff, size_t parmsize)              );
10046
10047   /* stfloppy.c */
10048   _PROTOTYPE( void fd_timer, (void)                                      );
10049
10050   /* stmain.c */
10051   _PROTOTYPE( void none, (void)                                          );
10052   _PROTOTYPE( void rupt, (void)                                          );
10053   _PROTOTYPE( void trap, (void)                                          );
10054   _PROTOTYPE( void checksp, (void)                                       );
10055   _PROTOTYPE( void aciaint, (void)                                       );
10056   _PROTOTYPE( void fake_int, (const char *s, int t)                      );
10057   _PROTOTYPE( void timint, (int t)                                       );
10058   _PROTOTYPE( void mdiint, (void)                                        );
10059   _PROTOTYPE( void iob, (int t)                                          );
```

```
10060   _PROTOTYPE( void idle_task, (void)                                      );
10061
10062   /* rs232.c */
10063   _PROTOTYPE( void siaint, (int type)                                     );
10064
10065   /* stcon.c */
10066   _PROTOTYPE( void func_key, (void)                                       );
10067   _PROTOTYPE( void dump, (void)                                           );
10068   _PROTOTYPE( void putk, (int c)                                          );
10069
10070   /* stdma.c */
10071   _PROTOTYPE( void dmagrab, (int p, dmaint_t func)                        );
10072   _PROTOTYPE( void dmafree, (int p)                                       );
10073   _PROTOTYPE( void dmaint, (void)                                         );
10074   _PROTOTYPE( void dmaaddr, (phys_bytes ad)                               );
10075   _PROTOTYPE( int dmardat, (int mode, int delay)                          );
10076   _PROTOTYPE( void dmawdat, (int mode, int data, int delay)               );
10077   _PROTOTYPE( void dmawcmd, (int data, unsigned mode)                     );
10078   _PROTOTYPE( void dmacomm, (int mode, int data, int delay)               );
10079   _PROTOTYPE( int dmastat, (int mode, int delay)                          );
10080
10081   /* stdskclk.c */
10082   _PROTOTYPE( int do_xbms, (phys_bytes address, int count, int rw, int minor) );
10083
10084   /* stkbd.c */
10085   _PROTOTYPE( void kbdint, (void)                                         );
10086   _PROTOTYPE( void kb_timer, (void)                                       );
10087   _PROTOTYPE( int kb_read, (int minor, char **bufindirect)                );
10088   _PROTOTYPE( void kb_init, (int minor)                                   );
10089
10090   /* stshadow.c */
10091   _PROTOTYPE( void mkshadow, (struct proc *p, phys_clicks c2)             );
10092   _PROTOTYPE( void rmshadow, (struct proc *p, phys_clicks *basep,
10093                   phys_clicks *sizep)                                     );
10094   _PROTOTYPE( void unshadow, (struct proc *p)                             );
10095
10096   /* stvdu.c */
10097   _PROTOTYPE( void flush, (struct tty *tp)                                );
10098   _PROTOTYPE( void console, (struct tty *tp)                              );
10099   _PROTOTYPE( void out_char, (struct tty *tp, int c)                      );
10100   _PROTOTYPE( void scr_init, (int minor)                                  );
10101   _PROTOTYPE( void vduswitch, (struct tty *tp)                            );
10102   _PROTOTYPE( void vdusetup, (unsigned int vres, char *vram,
10103                               unsigned short *vrgb)                       );
10104   _PROTOTYPE( void vbl, (void)                                            );
10105   _PROTOTYPE( int vdu_loadfont, (message *m_ptr)                          );
10106
10107   /* stwini.c */
10108   _PROTOTYPE( int wini_open, (message *mp)                                );
10109   _PROTOTYPE( int wini_rdwt, (message *mp)                                );
10110   _PROTOTYPE( int wini_hvrdwt, (message *mp)                              );
10111   _PROTOTYPE( int wini_transfer, (int rw, int pnr, int minor,
10112                   long pos, int count, vir_bytes vadr)                    );
10113   _PROTOTYPE( int wini_ioctl, (message *mp)                               );
10114   _PROTOTYPE( int wini_close, (message *mp)                               );
10115
10116   /* stacsi.c */
10117   _PROTOTYPE( int acsi_cmd, (int drive,  unsigned char *cmd, int cmdlen,
10118                   phys_bytes address, phys_bytes data_len,  int rw)       );
10119
```

```
10120   /* stscsi.c */
10121   _PROTOTYPE( void scsi_task, (void)                                      );
10122   _PROTOTYPE( void scsidmaint, (void)                                     );
10123   _PROTOTYPE( void scsiint, (void)                                        );
10124   _PROTOTYPE( int scsi_cmd, (int drive,  unsigned char *cmd, int cmdlen,
10125                   phys_bytes address, phys_bytes data_len,  int rw)       );
10126
10127   /* klib68k.s */
10128   _PROTOTYPE( void flipclicks, (phys_clicks c1, phys_clicks c2, phys_clicks n) );
10129   _PROTOTYPE( void copyclicks, (phys_clicks src, phys_clicks dest,
10130                   phys_clicks nclicks)                                    );
10131   _PROTOTYPE( void zeroclicks, (phys_clicks dest, phys_clicks nclicks)    );
10132   _PROTOTYPE( void phys_copy, (phys_bytes src, phys_bytes dest, phys_bytes n) );
10133
10134   /* stdskclks.s */
10135   _PROTOTYPE( int rd1byte, (void)                                         );
10136   _PROTOTYPE( int wr1byte, (int)                                          );
10137   _PROTOTYPE( long getsupra, (void)                                       );
10138   _PROTOTYPE( long geticd, (void)                                         );
10139
10140   /* mpx.s */
10141   _PROTOTYPE( int lock, (void)                                            );
10142   _PROTOTYPE( void unlock, (void)                                         );
10143   _PROTOTYPE( void restore, (int oldsr)                                   );
10144   _PROTOTYPE( void reboot, (void)                                         );
10145   _PROTOTYPE( int test_and_set, (char *flag)                              );
10146   _PROTOTYPE( unsigned long get_mem_size, (char *start_addr)              );
10147
10148   /* stprint.c */
10149   #ifdef DEBOUT
10150   _PROTOTYPE( void prtc, (int c)                                          );
10151   #endif
10152
10153   #ifdef FPP
10154   /* fpp.c */
10155   _PROTOTYPE( void fppinit, (void)                                        );
10156   _PROTOTYPE( void fpp_new_state, (struct proc *rp)                       );
10157   _PROTOTYPE( void fpp_save, (struct proc *rp, struct cpu_state *p)       );
10158   _PROTOTYPE( struct cpu_state  *fpp_restore, (struct proc *rp)           );
10159
10160   /* fpps.s */
10161   _PROTOTYPE( void _fppsave, (struct state_frame *p)                      );
10162   _PROTOTYPE( void _fppsavereg, (struct fpp_model *p)                     );
10163   _PROTOTYPE( void _fpprestore, (struct state_frame *p)                   );
10164   _PROTOTYPE( void _fpprestreg, (struct fpp_model *p)                     );
10165   #endif
10166
10167   #if (SHADOWING == 0)
10168   /* pmmu.c */
10169   _PROTOTYPE(void pmmuinit , (void)                                       );
10170   _PROTOTYPE(void pmmu_init_proc , (struct proc *rp )                     );
10171   _PROTOTYPE(void pmmu_restore , (struct proc *rp )                       );
10172   _PROTOTYPE(void pmmu_delete , (struct proc *rp )                        );
10173   _PROTOTYPE(void pmmu_flush , (struct proc *rp )                         );
10174   #endif
10175
10176   #endif /* (CHIP == M68000) */
10177
10178   #endif /* PROTO_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/sb16.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

10200    #ifndef SB16_H
10201    #define SB16_H
10202
10203    #define SB_DEBUG                0       /* 1 = print debug info */
10204    #define SB_DEBUG_2              0       /* 1 = print more debug info */
10205
10206    #define SB_TIMEOUT              32000   /* timeout count */
10207
10208    /* IRQ, base address and DMA channels */
10209    #define SB_IRQ          5
10210    #define SB_BASE_ADDR    0x220           /* 0x210, 0x220, 0x230, 0x240,
10211                                             * 0x250, 0x260, 0x280
10212                                             */
10213    #define SB_DMA_8        1               /* 0, 1, 3 */
10214    #define SB_DMA_16       5               /* 5, 6, 7 */
10215    #if _WORD_SIZE == 2
10216    #define DMA_SIZE        8192            /* Dma buffer MUST BE MULTIPLE OF 2 */
10217    #else
10218    #define DMA_SIZE        32768           /* Dma buffer MUST BE MULTIPLE OF 2 */
10219    #endif
10220
10221    /* Some defaults for the DSP */
10222    #define DEFAULT_SPEED           22050      /* Sample rate */
10223    #define DEFAULT_BITS            8          /* Nr. of bits */
10224    #define DEFAULT_SIGN            0          /* 0 = unsigned, 1 = signed */
10225    #define DEFAULT_STEREO          0          /* 0 = mono, 1 = stereo */
10226
10227    /* DMA port addresses */
10228    #define DMA8_ADDR       ((SB_DMA_8 & 3) << 1) + 0x00
10229    #define DMA8_COUNT      ((SB_DMA_8 & 3) << 1) + 0x01
10230    #define DMA8_MASK       0x0A
10231    #define DMA8_MODE       0x0B
10232    #define DMA8_CLEAR      0x0C
10233
10234
10235    /* If after this preprocessing stuff DMA8_PAGE is not defined
10236     * the 8-bit DMA channel specified is not valid
10237     */
10238    #if SB_DMA_8 == 0
10239    #  define DMA8_PAGE     0x87
10240    #else
10241    #  if SB_DMA_8 == 1
10242    #    define DMA8_PAGE   0x83
10243    #  else
10244    #    if SB_DMA_8 == 3
10245    #      define DMA8_PAGE 0x82
10246    #    endif
10247    #  endif
10248    #endif
10249
10250
10251    #define DMA16_ADDR      ((SB_DMA_16 & 3) << 2) + 0xC0
10252    #define DMA16_COUNT     ((SB_DMA_16 & 3) << 2) + 0xC2
10253    #define DMA16_MASK      0xD4
10254    #define DMA16_MODE      0xD6
```

```
10255   #define DMA16_CLEAR     0xD8
10256
10257
10258   /* If after this preprocessing stuff DMA16_PAGE is not defined
10259    * the 16-bit DMA channel specified is not valid
10260    */
10261   #if SB_DMA_16 == 5
10262   #  define DMA16_PAGE      0x8B
10263   #else
10264   #  if SB_DMA_16 == 6
10265   #    define DMA16_PAGE    0x89
10266   #  else
10267   #    if SB_DMA_16 == 7
10268   #      define DMA16_PAGE  0x8A
10269   #    endif
10270   #  endif
10271   #endif
10272
10273
10274   /* DMA modes */
10275   #define DMA16_AUTO_PLAY         0x58 + (SB_DMA_16 & 3)
10276   #define DMA16_AUTO_REC          0x54 + (SB_DMA_16 & 3)
10277   #define DMA8_AUTO_PLAY          0x58 + SB_DMA_8
10278   #define DMA8_AUTO_REC           0x54 + SB_DMA_8
10279
10280
10281   /* IO ports for soundblaster */
10282   #define DSP_RESET       0x6 + SB_BASE_ADDR
10283   #define DSP_READ        0xA + SB_BASE_ADDR
10284   #define DSP_WRITE       0xC + SB_BASE_ADDR
10285   #define DSP_COMMAND     0xC + SB_BASE_ADDR
10286   #define DSP_STATUS      0xC + SB_BASE_ADDR
10287   #define DSP_DATA_AVL    0xE + SB_BASE_ADDR
10288   #define DSP_DATA16_AVL  0xF + SB_BASE_ADDR
10289   #define MIXER_REG       0x4 + SB_BASE_ADDR
10290   #define MIXER_DATA      0x5 + SB_BASE_ADDR
10291   #define OPL3_LEFT       0x0 + SB_BASE_ADDR
10292   #define OPL3_RIGHT      0x2 + SB_BASE_ADDR
10293   #define OPL3_BOTH       0x8 + SB_BASE_ADDR
10294
10295
10296   /* DSP Commands */
10297   #define DSP_INPUT_RATE          0x42  /* set input sample rate */
10298   #define DSP_OUTPUT_RATE         0x41  /* set output sample rate */
10299   #define DSP_CMD_SPKON           0xD1  /* set speaker on */
10300   #define DSP_CMD_SPKOFF          0xD3  /* set speaker off */
10301   #define DSP_CMD_DMA8HALT        0xD0  /* halt DMA 8-bit operation */
10302   #define DSP_CMD_DMA8CONT        0xD4  /* continue DMA 8-bit operation */
10303   #define DSP_CMD_DMA16HALT       0xD5  /* halt DMA 16-bit operation */
10304   #define DSP_CMD_DMA16CONT       0xD6  /* continue DMA 16-bit operation */
10305   #define DSP_GET_VERSION         0xE1  /* get version number of DSP */
10306   #define DSP_CMD_8BITAUTO_IN     0xCE  /* 8 bit auto-initialized input */
10307   #define DSP_CMD_8BITAUTO_OUT    0xC6  /* 8 bit auto-initialized output */
10308   #define DSP_CMD_16BITAUTO_IN    0xBE  /* 16 bit auto-initialized input */
10309   #define DSP_CMD_16BITAUTO_OUT   0xB6  /* 16 bit auto-initialized output */
10310   #define DSP_CMD_IRQREQ8         0xF2  /* Interrupt request 8 bit        */
10311   #define DSP_CMD_IRQREQ16        0xF3  /* Interrupt request 16 bit        */
10312
10313
10314   /* DSP Modes */
```

```
10315    #define DSP_MODE_MONO_US        0x00  /* Mono unsigned */
10316    #define DSP_MODE_MONO_S         0x10  /* Mono signed */
10317    #define DSP_MODE_STEREO_US      0x20  /* Stereo unsigned */
10318    #define DSP_MODE_STEREO_S       0x30  /* Stereo signed */
10319
10320
10321    /* MIXER commands */
10322    #define MIXER_RESET             0x00  /* Reset */
10323    #define MIXER_DAC_LEVEL         0x04  /* Used for detection only */
10324    #define MIXER_MASTER_LEFT       0x30  /* Master volume left */
10325    #define MIXER_MASTER_RIGHT      0x31  /* Master volume right */
10326    #define MIXER_DAC_LEFT          0x32  /* Dac level left */
10327    #define MIXER_DAC_RIGHT         0x33  /* Dac level right */
10328    #define MIXER_FM_LEFT           0x34  /* Fm level left */
10329    #define MIXER_FM_RIGHT          0x35  /* Fm level right */
10330    #define MIXER_CD_LEFT           0x36  /* Cd audio level left */
10331    #define MIXER_CD_RIGHT          0x37  /* Cd audio level right */
10332    #define MIXER_LINE_LEFT         0x38  /* Line in level left */
10333    #define MIXER_LINE_RIGHT        0x39  /* Line in level right */
10334    #define MIXER_MIC_LEVEL         0x3A  /* Microphone level */
10335    #define MIXER_PC_LEVEL          0x3B  /* Pc speaker level */
10336    #define MIXER_OUTPUT_CTRL       0x3C  /* Output control */
10337    #define MIXER_IN_LEFT           0x3D  /* Input control left */
10338    #define MIXER_IN_RIGHT          0x3E  /* Input control right */
10339    #define MIXER_GAIN_IN_LEFT      0x3F  /* Input gain control left */
10340    #define MIXER_GAIN_IN_RIGHT     0x40  /* Input gain control right */
10341    #define MIXER_GAIN_OUT_LEFT     0x41  /* Output gain control left */
10342    #define MIXER_GAIN_OUT_RIGHT    0x42  /* Output gain control rigth */
10343    #define MIXER_AGC               0x43  /* Automatic gain control */
10344    #define MIXER_TREBLE_LEFT       0x44  /* Treble left */
10345    #define MIXER_TREBLE_RIGHT      0x45  /* Treble right */
10346    #define MIXER_BASS_LEFT         0x46  /* Bass left */
10347    #define MIXER_BASS_RIGHT        0x47  /* Bass right */
10348    #define MIXER_SET_IRQ           0x80  /* Set irq number */
10349    #define MIXER_SET_DMA           0x81  /* Set DMA channels */
10350    #define MIXER_IRQ_STATUS        0x82  /* Irq status */
10351
10352    /* Mixer constants */
10353    #define MIC                     0x01  /* Microphone */
10354    #define CD_RIGHT                0x02
10355    #define CD_LEFT                 0x04
10356    #define LINE_RIGHT              0x08
10357    #define LINE_LEFT               0x10
10358    #define FM_RIGHT                0x20
10359    #define FM_LEFT                 0x40
10360
10361    /* DSP constants */
10362    #define DSP_MAX_SPEED           44100      /* Max sample speed in KHz */
10363    #define DSP_MIN_SPEED           4000       /* Min sample speed in KHz */
10364    #define DSP_MAX_FRAGMENT_SIZE   DMA_SIZE   /* Maximum fragment size */
10365    #define DSP_MIN_FRAGMENT_SIZE   1024       /* Minimum fragment size */
10366
10367
10368    /* Number of bytes you can DMA before hitting a 64K boundary: */
10369    #define dma_bytes_left(phys)    \
10370       ((unsigned) (sizeof(int) == 2 ? 0 : 0x10000) - (unsigned) ((phys) & 0xFFFF))
10371
10372
10373    /* Function prototypes used by mixer and dsp */
10374    _PROTOTYPE(int mixer_set, (int reg, int data));
```

```
10375
10376    #endif /* SB16_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/sconst.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

10400    ! Miscellaneous constants used in assembler code.
10401    W               =       _WORD_SIZE      ! Machine word size.
10402
10403    ! Offsets in struct proc. They MUST match proc.h.
10404    P_STACKBASE     =       0
10405    #if _WORD_SIZE == 2
10406    ESREG           =       P_STACKBASE
10407    #else
10408    GSREG           =       P_STACKBASE
10409    FSREG           =       GSREG + 2       ! 386 introduces FS and GS segments
10410    ESREG           =       FSREG + 2
10411    #endif
10412    DSREG           =       ESREG + 2
10413    DIREG           =       DSREG + 2
10414    SIREG           =       DIREG + W
10415    BPREG           =       SIREG + W
10416    STREG           =       BPREG + W       ! hole for another SP
10417    BXREG           =       STREG + W
10418    DXREG           =       BXREG + W
10419    CXREG           =       DXREG + W
10420    AXREG           =       CXREG + W
10421    RETADR          =       AXREG + W       ! return address for save() call
10422    PCREG           =       RETADR + W
10423    CSREG           =       PCREG + W
10424    PSWREG          =       CSREG + W
10425    SPREG           =       PSWREG + W
10426    SSREG           =       SPREG + W
10427    P_STACKTOP      =       SSREG + W
10428    P_LDT_SEL       =       P_STACKTOP
10429    P_LDT           =       P_LDT_SEL + W
10430
10431    #if _WORD_SIZE == 2
10432    Msize           =       12              ! size of a message in 16-bit words
10433    #else
10434    Msize           =       9               ! size of a message in 32-bit words
10435    #endif
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/tty.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

10500   /*      tty.h - Terminals       */
10501
10502   #define TTY_IN_BYTES     256    /* tty input queue size */
10503   #define TAB_SIZE           8    /* distance between tab stops */
10504   #define TAB_MASK           7    /* mask to compute a tab stop position */
10505
10506   #define ESC             '\33'   /* escape */
10507
10508   #define O_NOCTTY       00400    /* from <fcntl.h>, or cc will choke */
10509   #define O_NONBLOCK     04000
10510
10511   typedef _PROTOTYPE( void (*devfun_t), (struct tty *tp) );
10512   typedef _PROTOTYPE( void (*devfunarg_t), (struct tty *tp, int c) );
10513
10514   typedef struct tty {
10515     int tty_events;               /* set when TTY should inspect this line */
10516
10517     /* Input queue.  Typed characters are stored here until read by a program. */
10518     u16_t *tty_inhead;            /* pointer to place where next char goes */
10519     u16_t *tty_intail;            /* pointer to next char to be given to prog */
10520     int tty_incount;              /* # chars in the input queue */
10521     int tty_eotct;                /* number of "line breaks" in input queue */
10522     devfun_t tty_devread;         /* routine to read from low level buffers */
10523     devfun_t tty_icancel;         /* cancel any device input */
10524     int tty_min;                  /* minimum requested #chars in input queue */
10525     clock_t tty_time;             /* time when the input is available */
10526     struct tty *tty_timenext;     /* for a list of ttys with active timers */
10527
10528     /* Output section. */
10529     devfun_t tty_devwrite;        /* routine to start actual device output */
10530     devfunarg_t tty_echo;         /* routine to echo characters input */
10531     devfun_t tty_ocancel;         /* cancel any ongoing device output */
10532     devfun_t tty_break;           /* let the device send a break */
10533
10534     /* Terminal parameters and status. */
10535     int tty_position;             /* current position on the screen for echoing */
10536     char tty_reprint;             /* 1 when echoed input messed up, else 0 */
10537     char tty_escaped;             /* 1 when LNEXT (^V) just seen, else 0 */
10538     char tty_inhibited;           /* 1 when STOP (^S) just seen (stops output) */
10539     char tty_pgrp;                /* slot number of controlling process */
10540     char tty_openct;              /* count of number of opens of this tty */
10541
10542     /* Information about incomplete I/O requests is stored here. */
10543     char tty_inrepcode;           /* reply code, TASK_REPLY or REVIVE */
10544     char tty_incaller;            /* process that made the call (usually FS) */
10545     char tty_inproc;              /* process that wants to read from tty */
10546     vir_bytes tty_in_vir;         /* virtual address where data is to go */
10547     int tty_inleft;               /* how many chars are still needed */
10548     int tty_incum;                /* # chars input so far */
10549     char tty_outrepcode;          /* reply code, TASK_REPLY or REVIVE */
10550     char tty_outcaller;           /* process that made the call (usually FS) */
10551     char tty_outproc;             /* process that wants to write to tty */
10552     vir_bytes tty_out_vir;        /* virtual address where data comes from */
10553     int tty_outleft;              /* # chars yet to be output */
10554     int tty_outcum;               /* # chars output so far */
```

```
10555       char tty_iocaller;            /* process that made the call (usually FS) */
10556       char tty_ioproc;              /* process that wants to do an ioctl */
10557       int tty_ioreq;                /* ioctl request code */
10558       vir_bytes tty_iovir;          /* virtual address of ioctl buffer */
10559
10560       /* Miscellaneous. */
10561       devfun_t tty_ioctl;           /* set line speed, etc. at the device level */
10562       devfun_t tty_close;           /* tell the device that the tty is closed */
10563       void *tty_priv;               /* pointer to per device private data */
10564       struct termios tty_termios;   /* terminal attributes */
10565       struct winsize tty_winsize;   /* window size (#lines and #columns) */
10566
10567       u16_t tty_inbuf[TTY_IN_BYTES];/* tty input buffer */
10568     } tty_t;
10569
10570     EXTERN tty_t tty_table[NR_CONS+NR_RS_LINES+NR_PTYS];
10571
10572     /* Values for the fields. */
10573     #define NOT_ESCAPED        0    /* previous character is not LNEXT (^V) */
10574     #define ESCAPED            1    /* previous character was LNEXT (^V) */
10575     #define RUNNING            0    /* no STOP (^S) has been typed to stop output */
10576     #define STOPPED            1    /* STOP (^S) has been typed to stop output */
10577
10578     /* Fields and flags on characters in the input queue. */
10579     #define IN_CHAR       0x00FF    /* low 8 bits are the character itself */
10580     #define IN_LEN        0x0F00    /* length of char if it has been echoed */
10581     #define IN_LSHIFT          8    /* length = (c & IN_LEN) >> IN_LSHIFT */
10582     #define IN_EOT        0x1000    /* char is a line break (^D, LF) */
10583     #define IN_EOF        0x2000    /* char is EOF (^D), do not return to user */
10584     #define IN_ESC        0x4000    /* escaped by LNEXT (^V), no interpretation */
10585
10586     /* Times and timeouts. */
10587     #define TIME_NEVER      ((clock_t) -1 < 0 ? (clock_t) LONG_MAX : (clock_t) -1)
10588     #define force_timeout() ((void) (tty_timeout = 0))
10589
10590     EXTERN tty_t *tty_timelist;     /* list of ttys with active timers */
10591
10592     /* Number of elements and limit of a buffer. */
10593     #define buflen(buf)     (sizeof(buf) / sizeof((buf)[0]))
10594     #define bufend(buf)     ((buf) + buflen(buf))
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/type.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

10600     #ifndef TYPE_H
10601     #define TYPE_H
10602
10603     typedef _PROTOTYPE( void task_t, (void) );
10604     typedef _PROTOTYPE( int (*rdwt_t), (message *m_ptr) );
10605     typedef _PROTOTYPE( void (*watchdog_t), (void) );
10606
10607     struct tasktab {
10608       task_t *initial_pc;
10609       int stksize;
```

```
10610     char name[8];
10611   };
10612
10613   struct memory {
10614     phys_clicks base;
10615     phys_clicks size;
10616   };
10617
10618   /* Administration for clock polling. */
10619   struct milli_state {
10620     unsigned long accum_count;    /* accumulated clock ticks */
10621     unsigned prev_count;          /* previous clock value */
10622   };
10623
10624   #if (CHIP == INTEL)
10625   typedef unsigned port_t;
10626   typedef unsigned segm_t;
10627   typedef unsigned reg_t;         /* machine register */
10628
10629   /* The stack frame layout is determined by the software, but for efficiency
10630    * it is laid out so the assembly code to use it is as simple as possible.
10631    * 80286 protected mode and all real modes use the same frame, built with
10632    * 16-bit registers.  Real mode lacks an automatic stack switch, so little
10633    * is lost by using the 286 frame for it.  The 386 frame differs only in
10634    * having 32-bit registers and more segment registers.  The same names are
10635    * used for the larger registers to avoid differences in the code.
10636    */
10637   struct stackframe_s {           /* proc_ptr points here */
10638   #if _WORD_SIZE == 4
10639     u16_t gs;                     /* last item pushed by save */
10640     u16_t fs;                     /*  ^ */
10641   #endif
10642     u16_t es;                     /*  | */
10643     u16_t ds;                     /*  | */
10644     reg_t di;                     /* di through cx are not accessed in C */
10645     reg_t si;                     /* order is to match pusha/popa */
10646     reg_t fp;                     /* bp */
10647     reg_t st;                     /* hole for another copy of sp */
10648     reg_t bx;                     /*  | */
10649     reg_t dx;                     /*  | */
10650     reg_t cx;                     /*  | */
10651     reg_t retreg;                 /* ax and above are all pushed by save */
10652     reg_t retadr;                 /* return address for assembly code save() */
10653     reg_t pc;                     /*  ^  last item pushed by interrupt */
10654     reg_t cs;                     /*  | */
10655     reg_t psw;                    /*  | */
10656     reg_t sp;                     /*  | */
10657     reg_t ss;                     /* these are pushed by CPU during interrupt */
10658   };
10659
10660   struct segdesc_s {              /* segment descriptor for protected mode */
10661     u16_t limit_low;
10662     u16_t base_low;
10663     u8_t base_middle;
10664     u8_t access;                  /* |P|DL|1|X|E|R|A| */
10665   #if _WORD_SIZE == 4
10666     u8_t granularity;             /* |G|X|0|A|LIMT| */
10667     u8_t base_high;
10668   #else
10669     u16_t reserved;
```

```
10670   #endif
10671   };
10672
10673   typedef _PROTOTYPE( int (*irq_handler_t), (int irq) );
10674
10675   #endif /* (CHIP == INTEL) */
10676
10677   #if (CHIP == M68000)
10678   typedef _PROTOTYPE( void (*dmaint_t), (void) );
10679
10680   typedef u32_t reg_t;            /* machine register */
10681
10682   /* The name and fields of this struct were chosen for PC compatibility. */
10683   struct stackframe_s {
10684     reg_t retreg;                 /* d0 */
10685     reg_t d1;
10686     reg_t d2;
10687     reg_t d3;
10688     reg_t d4;
10689     reg_t d5;
10690     reg_t d6;
10691     reg_t d7;
10692     reg_t a0;
10693     reg_t a1;
10694     reg_t a2;
10695     reg_t a3;
10696     reg_t a4;
10697     reg_t a5;
10698     reg_t fp;                     /* also known as a6 */
10699     reg_t sp;                     /* also known as a7 */
10700     reg_t pc;
10701     u16_t psw;
10702     u16_t dummy;                  /* make size multiple of reg_t for system.c */
10703   };
10704
10705   struct fsave {
10706     struct cpu_state {
10707           u16_t i_format;
10708           u32_t i_addr;
10709           u16_t i_state[4];
10710     } cpu_state;
10711     struct state_frame {
10712           u8_t frame_type;
10713           u8_t frame_size;
10714           u16_t reserved;
10715           u8_t frame[212];
10716     } state_frame;
10717     struct fpp_model {
10718           u32_t fpcr;
10719           u32_t fpsr;
10720           u32_t fpiar;
10721           struct fpN {
10722                   u32_t high;
10723                   u32_t low;
10724                   u32_t mid;
10725           } fpN[8];
10726     } fpp_model;
10727   };
10728   #endif /* (CHIP == M68000) */
10729
```

```
10730    #endif /* TYPE_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/wdeth.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
10800    /*
10801    wdeth.h
10802
10803    Created:        before Dec 28, 1992 by Philip Homburg
10804    */
10805
10806    #ifndef WDETH_H
10807    #define WDETH_H
10808
10809    /* Western Digital Ethercard Plus, or WD8003E card. */
10810
10811    #define EPL_REG0         0x0            /* Control(write) and status(read) */
10812    #define EPL_REG1         0x1
10813    #define EPL_REG2         0x2
10814    #define EPL_REG3         0x3
10815    #define EPL_REG4         0x4
10816    #define EPL_REG5         0x5
10817    #define EPL_REG6         0x6
10818    #define EPL_REG7         0x7
10819    #define EPL_EA0          0x8            /* Most significant eaddr byte */
10820    #define EPL_EA1          0x9
10821    #define EPL_EA2          0xA
10822    #define EPL_EA3          0xB
10823    #define EPL_EA4          0xC
10824    #define EPL_EA5          0xD            /* Least significant eaddr byte */
10825    #define EPL_TLB          0xE
10826    #define EPL_CHKSUM       0xF            /* sum from epl_ea0 upto here is 0xFF */
10827    #define EPL_DP8390      0x10            /* NatSemi chip */
10828
10829    #define EPL_MSR         EPL_REG0        /* memory select register */
10830    #define EPL_ICR         EPL_REG1        /* interface configuration register */
10831    #define EPL_IRR         EPL_REG4        /* interrupt request register (IRR) */
10832    #define EPL_790_HWR     EPL_REG4        /* '790 hardware support register */
10833    #define EPL_LAAR        EPL_REG5        /* LA address register (write only) */
10834    #define EPL_790_ICR     EPL_REG6        /* '790 interrupt control register */
10835    #define EPL_GP2         EPL_REG7        /* general purpose register 2 */
10836    #define EPL_790_B       EPL_EA3         /* '790 memory register */
10837    #define EPL_790_GCR     EPL_EA5         /* '790 General Control Register */
10838
10839    /* Bits in EPL_MSR */
10840    #define E_MSR_MEMADDR   0x3F            /* Bits SA18-SA13, SA19 implicit 1 */
10841    #define E_MSR_MENABLE   0x40            /* Memory Enable */
10842    #define E_MSR_RESET     0x80            /* Software Reset */
10843
10844    /* Bits in EPL_ICR */
10845    #define E_ICR_16BIT     0x01            /* 16 bit bus */
10846    #define E_ICR_IR2       0x04            /* bit 2 of encoded IRQ */
10847    #define E_ICR_MEMBIT    0x08            /* 583 mem size mask */
10848
10849    /* Bits in EPL_IRR */
```

```
10850    #define E_IRR_IR0       0x20            /* bit 0 of encoded IRQ */
10851    #define E_IRR_IR1       0x40            /* bit 1 of encoded IRQ */
10852    #define E_IRR_IEN       0x80            /* enable interrupts */
10853
10854    /* Bits in EPL_LAAR */
10855    #define E_LAAR_A19      0x01            /* address lines for above 1M ram */
10856    #define E_LAAR_A20      0x02            /* address lines for above 1M ram */
10857    #define E_LAAR_A21      0x04            /* address lines for above 1M ram */
10858    #define E_LAAR_A22      0x08            /* address lines for above 1M ram */
10859    #define E_LAAR_A23      0x10            /* address lines for above 1M ram */
10860    #define E_LAAR_SOFTINT  0x20            /* enable software interrupt */
10861    #define E_LAAR_LAN16E   0x40            /* enables 16 bit RAM for LAN */
10862    #define E_LAAR_MEM16E   0x80            /* enables 16 bit RAM for host */
10863
10864    /* Bits and values in EPL_TLB */
10865    #define E_TLB_EB        0x05            /* WD8013EB */
10866    #define E_TLB_E         0x27            /* WD8013 Elite */
10867    #define E_TLB_SMCE      0x29            /* SMC Elite 16 */
10868    #define E_TLB_SMC8216C  0x2B            /* SMC 8216 C */
10869
10870    #define E_TLB_REV       0x1F            /* revision mask */
10871    #define E_TLB_SOFT      0x20            /* soft config */
10872    #define E_TLB_RAM       0x40            /* extra ram bit */
10873
10874    /* Bits in EPL_790_HWR */
10875    #define E_790_HWR_SWH   0x80            /* switch register set */
10876
10877    /* Bits in EPL_790_ICR */
10878    #define E_790_ICR_EIL   0x01            /* enable interrupts */
10879
10880    /* Bits in EPL_790_GCR when E_790_HWR_SWH is set in EPL_790_HWR */
10881    #define E_790_GCR_IR0   0x04            /* bit 0 of encoded IRQ */
10882    #define E_790_GCR_IR1   0x08            /* bit 1 of encoded IRQ */
10883    #define E_790_GCR_IR2   0x40            /* bit 2 of encoded IRQ */
10884
10885
10886    #define inb_we(dep, reg) (in_byte(dep->de_base_port+reg))
10887    #define outb_we(dep, reg, data) (out_byte(dep->de_base_port+reg, data))
10888
10889    #endif /* WDETH_H */
10890
10891    /*
10892     * $PchHeader: /mount/hd2/minix/sys/kernel/ibm/RCS/wdeth.h,v 1.3 1994/10/31 10:36:29 philip Exp $
10893     */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/aha_scsi.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

10900    /*
10901     * This file contains the device dependent part of an experimental disk
10902     * and tape driver for the Adaptec 154x SCSI Host Adapter family, written
10903     * by James da Silva (jds@cs.umd.edu).
10904     *
10905     * I wrote this driver using the technical documentation for the AHA available
10906     * from the Adaptec BBS at 1-408-945-7727, and from the SCSI standard drafts
10907     * available on NCR's SCSI BBS at 1-316-636-8700.  I suggest you get both
10908     * these documents if you want to understand and hack this code.
10909     *
```

```
10910     * This code has been extensively modified by Kees J. Bot (kjb@cs.vu.nl) to
10911     * a point that James will barely recognize it as his.  It is completely
10912     * remodeled and doubled in both size and functionality.  It is no longer
10913     * considered experimental either.
10914     *
10915     * The supported device numbers are as follows:
10916     *   #  Name    Device
10917     *   0  sd0     disk 0, entire disk
10918     *   1  sd1     disk 0, partition 1
10919     *   2  sd2     disk 0, partition 2
10920     *   3  sd3     disk 0, partition 3
10921     *   4  sd4     disk 0, partition 4
10922     *   5  sd5     disk 1, entire disk
10923     *   6  sd6     disk 1, partition 1
10924     *  ..  ....    ....
10925     *  39  sd39    disk 7, partition 4
10926     *
10927     *  64  nrst0   tape 0, no rewind
10928     *  65  rst0    tape 0, rewind
10929     *  66  nrst1   tape 1, no rewind
10930     *  ..  ....    ....
10931     *  79  rst7    tape 7, rewind
10932     *
10933     * 128  sd1a    disk 0, partition 1, subpartition 1
10934     * 129  sd1b    disk 0, partition 1, subpartition 2
10935     * ...  ....    ....
10936     * 255  sd39d   disk 7, partition 4, subpartition 4
10937     *
10938     * The translation of device numbers to targets and logical units is very
10939     * simple:  The target is the same as the disk or tape number, the logical
10940     * unit is always zero.  Devices with logical unit numbers other then zero
10941     * are virtually extinct.  If you happen to have such a dinosaur device,
10942     * then you can reprogram (e.g.) sd35 and st7 to target 0, lun 1 from the
10943     * Boot Monitor with 'sd35=0,1'.
10944     *
10945     *
10946     * The file contains one entry point:
10947     *
10948     *   aha_scsi_task:     main entry when system is brought up
10949     *
10950     *
10951     * Changes:
10952     *       5 May 1992 by Kees J. Bot: device dependent/independent split.
10953     *       7 Jul 1992 by Kees J. Bot: speedup & features.
10954     *      28 Dec 1992 by Kees J. Bot: completely remodeled & virtual memory.
10955     *      18 Sep 1994 by Kees J. Bot: removed "send 2 commands at once" junk.
10956     */
10957    #include "kernel.h"
10958    #include "driver.h"
10959    #include "drvlib.h"
10960    #if ENABLE_ADAPTEC_SCSI
10961    #include <fcntl.h>
10962    #include <sys/ioctl.h>
10963    #include <sys/mtio.h>
10964    #include "assert.h"
10965    INIT_ASSERT
10966
10967
10968    #ifndef AHA_DEBUG
10969    #define AHA_DEBUG       0       /* 1=print all SCSI errors | 2=dump ccb
```

```
10970                                   * 4=show request | 8=dump scsi cmd
10971                                   */
10972   #endif
10973
10974   /* AHA-154x port addresses */
10975   #define AHA_BASEREG     0x330   /* default base port address of AHA registers */
10976   #define AHA_CNTLREG     aha_basereg+0   /* Control Register - write only */
10977   #define AHA_STATREG     aha_basereg+0   /* Status Register - read only */
10978   #define AHA_DATAREG     aha_basereg+1   /* Data Register - read/write */
10979   #define AHA_INTRREG     aha_basereg+2   /* Interrupt Flags - read only */
10980
10981   /* control register bits */
10982   #define AHA_HRST        0x80    /* bit 7 - Hard Reset */
10983   #define AHA_SRST        0x40    /* bit 6 - Soft Reset */
10984   #define AHA_IRST        0x20    /* bit 5 - Interrupt Reset */
10985   #define AHA_SCRST       0x10    /* bit 4 - SCSI Bus Reset */
10986   /*                      0x08     * bit 3 - Reserved (set to 0) */
10987   /*                      0x04     * bit 2 - Reserved (set to 0) */
10988   /*                      0x02     * bit 1 - Reserved (set to 0) */
10989   /*                      0x01     * bit 0 - Reserved (set to 0) */
10990
10991   /* status register bits */
10992   #define AHA_STST        0x80    /* bit 7 - Self Test in Progress */
10993   #define AHA_DIAGF       0x40    /* bit 6 - Internal Diagnostic Failure */
10994   #define AHA_INIT        0x20    /* bit 5 - Mailbox Initialization Required */
10995   #define AHA_IDLE        0x10    /* bit 4 - SCSI Host Adapter Idle */
10996   #define AHA_CDF         0x08    /* bit 3 - Command/Data Out Port Full */
10997   #define AHA_DF          0x04    /* bit 2 - Data In Port Full */
10998   /*                      0x02     * bit 1 - Reserved */
10999   #define AHA_INVDCMD     0x01    /* bit 0 - Invalid Host Adapter Command */
11000
11001   /* interrupt flags register bits */
11002   #define AHA_ANYINT      0x80    /* bit 7 - Any Interrupt */
11003   /*                      0x40     * bit 6 - Reserved */
11004   /*                      0x20     * bit 5 - Reserved */
11005   /*                      0x10     * bit 4 - Reserved */
11006   #define AHA_SCRD        0x08    /* bit 3 - SCSI Reset Detected */
11007   #define AHA_HACC        0x04    /* bit 2 - Host Adapter Command Complete */
11008   #define AHA_MBOE        0x02    /* bit 1 - Mailbox Out Empty */
11009   #define AHA_MBIF        0x01    /* bit 0 - Mailbox In Full */
11010
11011   /* AHA board models */
11012   #define AHA1540         0x30
11013   #define AHA1540A        0x41
11014   #define AHA1640         0x42
11015   #define AHA1740         0x43
11016   #define AHA1540C        0x44
11017   #define AHA1540CF       0x45
11018   #define BT545           0x20    /* BusLogic */
11019
11020   /* AHA Command Codes */
11021   #define AHACOM_INITBOX          0x01    /* Mailbox Initialization */
11022   #define AHACOM_STARTSCSI        0x02    /* Start SCSI Command */
11023   #define AHACOM_HAINQUIRY        0x04    /* Host Adapter Inquiry */
11024   #define AHACOM_SETIMEOUT        0x06    /* Set SCSI selection time out value */
11025   #define AHACOM_BUSON            0x07    /* Set DMA bus on time */
11026   #define AHACOM_BUSOFF           0x08    /* Set DMA bus off time */
11027   #define AHACOM_SPEED            0x09    /* Set DMA transfer speed */
11028   #define AHACOM_INSTALLED        0x0A    /* Return Installed Devices */
11029   #define AHACOM_GETCONFIG        0x0B    /* Return Configuration Data */
```

```
11030    #define AHACOM_GETSETUP         0x0D    /* Return Setup Data */
11031    #define AHACOM_EXTBIOS          0x28    /* Return Extended BIOS Info */
11032    #define AHACOM_MBOX_ENABLE      0x29    /* Enable Mailbox Interface */
11033
11034    /* AHA Mailbox Out Codes */
11035    #define AHA_MBOXFREE    0x00    /* Mailbox is Free */
11036    #define AHA_MBOXSTART   0x01    /* Start Command */
11037    #define AHA_MBOXABORT   0x02    /* Abort Command */
11038    /* AHA Mailbox In Codes */
11039    #define AHA_MBOXOK      0x01    /* Command Completed Successfully */
11040    #define AHA_MBOXERR     0x04    /* Command Completed with Error */
11041
11042
11043    /* Basic types */
11044    typedef unsigned char byte;
11045    typedef byte big16[2];  /* 16 bit big-endian values */
11046    typedef byte big24[3];  /* AHA uses 24 bit, big-endian values! */
11047    typedef byte big32[4];  /* Group 1 SCSI commands use 32 bit big-endian values */
11048
11049    /* AHA Mailbox structure */
11050    typedef struct {
11051      byte status;          /* Command or Status byte */
11052      big24 ccbptr;         /* pointer to Command Control Block */
11053    } mailbox_t;
11054
11055    /* SCSI Group 0 Command Descriptor Block structure */
11056    typedef union {
11057        struct {    /* Disk i/o commands */
11058            byte d_scsi_op;         /* SCSI Operation Code */
11059    #           define SCSI_UNITRDY  0x00   /* Test Unit Ready */
11060    #           define SCSI_REWIND   0x01   /* Rewind */
11061    #           define SCSI_REQSENSE 0x03   /* Request sense */
11062    #           define SCSI_RDLIMITS 0x05   /* Read Block Limits Opcode */
11063    #           define SCSI_READ     0x08   /* Group 0 Read Opcode */
11064    #           define SCSI_WRITE    0x0A   /* Group 0 Write Opcode */
11065    #           define SCSI_WREOF    0x10   /* Write File Marks */
11066    #           define SCSI_SPACE    0x11   /* Space over filemarks/blocks */
11067    #           define SCSI_INQUIRY  0x12   /* Group 0 Inquiry Opcode */
11068    #           define SCSI_MDSELECT 0x15   /* Group 0 Mode Select Opcode */
11069    #           define SCSI_ERASE    0x19   /* Erase Tape */
11070    #           define SCSI_MDSENSE  0x1A   /* Group 0 Mode Sense Opcode */
11071    #           define SCSI_STRTSTP  0x1B   /* Start/Stop */
11072    #           define SCSI_LOADUNLD 0x1B   /* Load/Unload */
11073            big24 d_lba;            /* LUN and logical block address */
11074            byte d_nblocks;         /* Transfer size in blocks */
11075            byte d_control;         /* Reserved and link bit fields, set to 0 */
11076        } d;
11077        struct {    /* Tape i/o commands */
11078            byte t_scsi_op;         /* SCSI Operation Code */
11079            byte t_fixed;           /* Fixed length? */
11080            big24 t_trlength;       /* Transfer length */
11081            byte t_control;         /* reserved and link bit fields, set to 0 */
11082        } t;
11083    } cdb0_t;
11084    #define scsi_op         d.d_scsi_op
11085    #define lba             d.d_lba
11086    #define nblocks         d.d_nblocks
11087    #define fixed           t.t_fixed
11088    #define trlength        t.t_trlength
11089    #define control         d.d_control
```

```
11090
11091   /* SCSI Group 1 Command Descriptor Block structure */
11092   typedef union {
11093       struct {    /* Disk i/o commands */
11094           byte d_scsi_op;         /* SCSI Operation Code */
11095   #           define SCSI_CAPACITY 0x25   /* Read Capacity */
11096   #           define SCSI_READ1    0x28   /* Group 1 Read Opcode */
11097   #           define SCSI_WRITE1   0x2A   /* Group 1 Write Opcode */
11098           byte d_lunra;           /* LUN etc. */
11099           big32 d_lba;            /* Logical Block Address */
11100           byte reserved;
11101           big16 d_nblocks;        /* transfer size in blocks */
11102           byte d_control;         /* reserved and link bit fields, set to 0 */
11103       } d;
11104   } cdb1_t;
11105   #define lunra           d.d_lunra
11106
11107   /* SCSI Request Sense Information */
11108   typedef struct {
11109       byte errc;                  /* Error Code, Error Class, and Valid bit */
11110       byte segnum;                /* Segment Number */
11111       byte key;                   /* Sense Key */
11112   #       define sense_key(key)   (key & 0x0F)    /* the key portion */
11113   #       define sense_ili(key)   (key & 0x20)    /* illegal block size */
11114   #       define sense_eom(key)   (key & 0x40)    /* end-of-media */
11115   #       define sense_eof(key)   (key & 0x80)    /* filemark reached */
11116       big32 info;                 /* sense info */
11117       byte len;                   /* additional length */
11118       big32 comspec;              /* command specific info */
11119       byte add_code;              /* additional sense code */
11120       byte add_qual;              /* additional sense code qualifier */
11121   } sense_t;
11122
11123   /* Interesting SCSI sense key types. */
11124   #define SENSE_NO_SENSE          0x00
11125   #define SENSE_RECOVERED         0x01
11126   #define SENSE_NOT_READY         0x02
11127   #define SENSE_HARDWARE          0x04
11128   #define SENSE_UNIT_ATT          0x06
11129   #define SENSE_BLANK_CHECK       0x08
11130   #define SENSE_VENDOR            0x09
11131   #define SENSE_ABORTED_CMD       0x0B
11132
11133   /* SCSI Inquiry Information */
11134   typedef struct {
11135       byte devtype;               /* Peripheral Device Type */
11136   #       define SCSI_DEVDISK     0       /* Direct-access */
11137   #       define SCSI_DEVTAPE     1       /* Sequential-access */
11138   #       define SCSI_DEVPRN      2       /* Printer */
11139   #       define SCSI_DEVCPU      3       /* Processor */
11140   #       define SCSI_DEVWORM     4       /* Write-Once Read-Multiple device */
11141   #       define SCSI_DEVCDROM    5       /* Read-Only Direct-access */
11142   #       define SCSI_DEVSCANNER  6       /* Scanner */
11143   #       define SCSI_DEVOPTICAL  7       /* Optical Memory */
11144   #       define SCSI_DEVJUKEBOX  8       /* Medium Changer device */
11145   #       define SCSI_DEVCOMM     9       /* Communications device */
11146   #       define SCSI_DEVMAX      9       /* Last device type we know about */
11147   #       define SCSI_DEVUNKNOWN  10      /* If we do not know or care. */
11148       byte devqual;               /* Device-Type Qualifier */
11149   #       define scsi_rmb(d)      (((d) & 0x80) != 0)     /* Removable? */
```

```
11150       byte stdver;                /* Version of standard compliance */
11151   #       define scsi_isover(v)   (((v) & 0xC0) >> 6)     /* ISO version */
11152   #       define scsi_ecmaver(v)  (((v) & 0x38) >> 3)     /* ECMA version */
11153   #       define scsi_ansiver(v)  ((v) & 0x07)            /* ANSI version */
11154       byte format;                /* Response data format */
11155       byte len;                   /* length of remaining info */
11156       byte reserved[2];
11157       byte flags;
11158   #       define scsi_sync(f)     (((f) & 0x10) != 0)     /* Sync SCSI? */
11159       char vendor[8];             /* Vendor name */
11160       char product[16];           /* Product name */
11161       char revision[4];           /* Revision level */
11162       char extra[20];             /* Vendor specific */
11163   } inquiry_t;
11164
11165   /* AHA Command Control Block structure */
11166   typedef struct {
11167       byte opcode;                /* Operation Code */
11168   #       define CCB_INIT         0x00            /* SCSI Initiator Command */
11169   #       define CCB_TARGET       0x01            /* Target Mode Command */
11170   #       define CCB_SCATTER      0x02         /* Initiator with scatter/gather */
11171       byte addrcntl;              /* Address and Direction Control: */
11172   #       define ccb_scid(id)     (((id)<<5)&0xE0) /* SCSI ID field */
11173   #       define CCB_OUTCHECK     0x10             /* Outbound length check */
11174   #       define CCB_INCHECK      0x08             /* Inbound length check */
11175   #       define CCB_NOCHECK      0x00             /* No length check */
11176   #       define ccb_lun(lun)     ((lun)&0x07)     /* SCSI LUN field */
11177       byte cmdlen;                /* SCSI Command Length (6 for Group 0) */
11178       byte senselen;              /* Request/Disable Sense, Allocation Length */
11179   #       define CCB_SENSEREQ     0x0E            /* Request Sense, 14 bytes */
11180   #       define CCB_SENSEOFF     0x01            /* Disable Request Sense */
11181       big24 datalen;              /* Data Length:  3 bytes, big endian */
11182       big24 dataptr;              /* Data Pointer: 3 bytes, big endian */
11183       big24 linkptr;              /* Link Pointer: 3 bytes, big endian */
11184       byte linkid;                /* Command Linking Identifier */
11185       byte hastat;                /* Host Adapter Status */
11186   #       define HST_TIMEOUT      0x11            /* SCSI selection timeout */
11187       byte tarstat;               /* Target Device Status */
11188   #       define TST_CHECK        0x02            /* Check status in sense[] */
11189   #       define TST_LUNBUSY      0x08            /* Unit is very busy */
11190       byte reserved[2];           /* reserved, set to 0 */
11191       byte cmd[sizeof(cdb1_t)];   /* SCSI Command Descriptor Block */
11192       byte sense[sizeof(sense_t)];/* SCSI Request Sense Information */
11193   } ccb_t;
11194
11195
11196           /* End of one chunk must be as "odd" as the start of the next. */
11197   #define DMA_CHECK(end, start)   ((((int) (end) ^ (int) (start)) & 1) == 0)
11198
11199   /* Scatter/Gather DMA list */
11200   typedef struct {
11201       big24 datalen;              /* length of a memory segment */
11202       big24 dataptr;              /* address of a memory segment */
11203   } dma_t;
11204
11205
11206   /* Miscellaneous parameters */
11207   #define SCSI_TIMEOUT     250    /* SCSI selection timeout (ms), 0 = none */
11208   #define AHA_TIMEOUT      500    /* max msec wait for controller reset */
11209
```

```
11210   #define MAX_DEVICES        8    /* 8 devices for the 8 SCSI targets */
11211   #define NR_DISKDEVS      (MAX_DEVICES * DEV_PER_DRIVE)
11212   #define NR_TAPEDEVS      (MAX_DEVICES * 2)
11213   #define NR_GENDEVS       (MAX_DEVICES)
11214   #define SUB_PER_DRIVE    (NR_PARTITIONS * NR_PARTITIONS)
11215   #define NR_SUBDEVS       (MAX_DEVICES * SUB_PER_DRIVE)
11216   #define MINOR_st0         64
11217
11218   #define TYPE_SD            0    /* disk device number */
11219   #define TYPE_NRST          1    /* non rewind-on-close tape device */
11220   #define TYPE_RST           2    /* rewind-on-close tape device */
11221
11222
11223   /* Variables */
11224   PRIVATE struct scsi {   /* Per-device table */
11225       char targ;                  /* SCSI Target ID */
11226       char lun;                   /* SCSI Logical Unit Number */
11227       char state;                 /* online? */
11228   #       define S_PRESENT        0x01    /* Device exists */
11229   #       define S_READY          0x02    /* Device is ready */
11230   #       define S_RDONLY         0x04    /* Device is read-only */
11231       char devtype;               /* SCSI_DEVDISK, SCSI_DEVTAPE, ... */
11232       unsigned block_size;        /* device or media block size */
11233       unsigned count_max;         /* maximum single read or write */
11234       unsigned open_ct;           /* number of processes using the device */
11235       union {
11236           struct {                /* Tape data */
11237               char open_mode;     /* open for reading or writing? */
11238               char at_eof;        /* got EOF mark */
11239               char need_eof;      /* need to write an eof mark */
11240               char tfixed;        /* tape in fixed mode */
11241               struct mtget tstat; /* tape status info */
11242               struct device dummypart;  /* something for s_prepare to return */
11243           } tape;
11244           struct {                /* Disk data */
11245               struct device part[DEV_PER_DRIVE];    /* primaries: sd[0-4] */
11246               struct device subpart[SUB_PER_DRIVE]; /* subparts: sd[1-4][a-d] */
11247           } disk;
11248       } u;
11249   } scsi[MAX_DEVICES];
11250
11251   #define open_mode       u.tape.open_mode
11252   #define at_eof          u.tape.at_eof
11253   #define need_eof        u.tape.need_eof
11254   #define tfixed          u.tape.tfixed
11255   #define tstat           u.tape.tstat
11256   #define dummypart       u.tape.dummypart
11257   #define part            u.disk.part
11258   #define subpart         u.disk.subpart
11259
11260   /* Tape device status (tstat.mt_dsreg). */
11261   #define DS_OK           0       /* Device OK */
11262   #define DS_ERR          1       /* Error state */
11263   #define DS_EOF          2       /* Last read or space hit EOF */
11264
11265   /* SCSI device types */
11266   PRIVATE char *scsi_devstr[SCSI_DEVMAX+1] = {
11267     "DISK", "TAPE", "PRINTER", "CPU", "WORM", "CDROM", "SCANNER", "OPTICAL",
11268     "JUKEBOX", "COMM"
11269   };
```

```
11270
11271   /* SCSI sense key types */
11272   PRIVATE char *str_scsi_sense[] = {
11273     "NO SENSE INFO", "RECOVERED ERROR", "NOT READY", "MEDIUM ERROR",
11274     "HARDWARE ERROR", "ILLEGAL REQUEST", "UNIT ATTENTION", "DATA PROTECT",
11275     "BLANK CHECK", "VENDOR UNIQUE ERROR", "COPY ABORTED", "ABORTED COMMAND",
11276     "EQUAL", "VOLUME OVERFLOW", "MISCOMPARE", "SENSE RESERVED"
11277   };
11278
11279   /* Some of the above errors must be printed on the console. */
11280   #if AHA_DEBUG & 1
11281   #define sense_serious(key)      ((key) != 0)
11282   #else
11283   #define sense_serious(key)      ((0xFE1C & (1 << (key))) != 0)
11284   #endif
11285
11286   /* Administration for one SCSI request. */
11287   typedef struct request {
11288     unsigned count;               /* number of bytes to transfer */
11289     unsigned retry;               /* number of tries allowed if retryable */
11290     unsigned long pos;            /* first byte on the device to transfer */
11291     ccb_t ccb;                    /* Command Control Block */
11292     dma_t dmalist[NR_IOREQS];     /* scatter/gather dma list */
11293     dma_t *dmaptr;                /* to add scatter/gather entries */
11294     dma_t *dmalimit;              /* adapter model dependent limit to list */
11295     struct iorequest_s *iov[NR_IOREQS];   /* affected I/O requests */
11296   } request_t;
11297
11298   PRIVATE request_t request;
11299   #define rq (&request)           /* current request (there is only one) */
11300
11301   #define ccb_cmd0(rq)    (* (cdb0_t *) (rq)->ccb.cmd)
11302   #define ccb_cmd1(rq)    (* (cdb1_t *) (rq)->ccb.cmd)
11303   #define ccb_sense(rq)   (* (sense_t *) ((rq)->ccb.cmd + (rq)->ccb.cmdlen))
11304
11305   PRIVATE int aha_basereg;        /* base I/O register */
11306   PRIVATE int aha_model;          /* board model */
11307   PRIVATE struct scsi *s_sp;      /* active SCSI device struct */
11308   PRIVATE struct device *s_dv;    /* active partition */
11309   PRIVATE int s_type;             /* sd, rst, nrst? */
11310   PRIVATE unsigned long s_nextpos;/* next byte on the device to transfer */
11311   PRIVATE unsigned long s_buf_blk;/* disk block currently in tmp_buf */
11312   PRIVATE int s_opcode;           /* DEV_READ or DEV_WRITE */
11313   PRIVATE int s_must;             /* must finish the current request? */
11314   PRIVATE int aha_irq;            /* configured IRQ */
11315   PRIVATE mailbox_t mailbox[2];   /* out and in mailboxes */
11316   PRIVATE inquiry_t inqdata;      /* results of Inquiry command */
11317
11318
11319   /* Functions */
11320
11321   FORWARD _PROTOTYPE( struct device *s_prepare, (int device) );
11322   FORWARD _PROTOTYPE( char *s_name, (void) );
11323   FORWARD _PROTOTYPE( int s_do_open, (struct driver *dp, message *m_ptr) );
11324   FORWARD _PROTOTYPE( int scsi_probe, (void) );
11325   FORWARD _PROTOTYPE( int scsi_sense, (void) );
11326   FORWARD _PROTOTYPE( int scsi_inquiry, (void) );
11327   FORWARD _PROTOTYPE( int scsi_ndisk, (void) );
11328   FORWARD _PROTOTYPE( int scsi_ntape, (void) );
11329   FORWARD _PROTOTYPE( int s_schedule, (int proc_nr, struct iorequest_s *iop) );
```

```
11330    FORWARD _PROTOTYPE( int s_finish, (void) );
11331    FORWARD _PROTOTYPE( int s_rdcdrom, (int proc_nr, struct iorequest_s *iop,
11332                    unsigned long pos, unsigned nbytes, phys_bytes user_phys) );
11333    FORWARD _PROTOTYPE( int s_do_close, (struct driver *dp, message *m_ptr) );
11334    FORWARD _PROTOTYPE( int s_do_ioctl, (struct driver *dp, message *m_ptr) );
11335    FORWARD _PROTOTYPE( int scsi_simple, (int opcode, int count) );
11336    FORWARD _PROTOTYPE( void group0, (void) );
11337    FORWARD _PROTOTYPE( void group1, (void) );
11338    FORWARD _PROTOTYPE( int scsi_command, (phys_bytes data, vir_bytes len) );
11339    FORWARD _PROTOTYPE( void aha_command, (int outlen, byte *outptr,
11340                                                int inlen, byte *inptr) );
11341    FORWARD _PROTOTYPE( int aha_reset, (void) );
11342    FORWARD _PROTOTYPE( int s_handler, (int irq) );
11343
11344    FORWARD _PROTOTYPE( void h2b16, (big16 b, U16_t l) );
11345    FORWARD _PROTOTYPE( void h2b24, (big24 b, u32_t l) );
11346    FORWARD _PROTOTYPE( void h2b32, (big32 b, u32_t l) );
11347    FORWARD _PROTOTYPE( u16_t b2h16, (big16 b) );
11348    FORWARD _PROTOTYPE( u32_t b2h24, (big24 b) );
11349    FORWARD _PROTOTYPE( u32_t b2h32, (big32 b) );
11350
11351
11352    #if AHA_DEBUG & 2
11353    FORWARD _PROTOTYPE( void errordump, (void) );
11354    #else
11355    #define errordump()
11356    #endif
11357
11358    #if AHA_DEBUG & 4
11359    FORWARD _PROTOTYPE( void show_req, (void) );
11360    #else
11361    #define show_req()
11362    #endif
11363
11364    #if AHA_DEBUG & 8
11365    FORWARD _PROTOTYPE( void dump_scsi_cmd, (void) );
11366    #else
11367    #define dump_scsi_cmd()
11368    #endif
11369
11370    FORWARD _PROTOTYPE( void s_geometry, (struct partition *entry));
11371
11372
11373    /* Entry points to this driver. */
11374    PRIVATE struct driver s_dtab = {
11375      s_name,       /* current device's name */
11376      s_do_open,    /* open or mount request, initialize device */
11377      s_do_close,   /* release device */
11378      s_do_ioctl,   /* tape and partition ioctls */
11379      s_prepare,    /* prepare for I/O on a given minor device */
11380      s_schedule,   /* precompute SCSI transfer parameters, etc. */
11381      s_finish,     /* do the I/O */
11382      nop_cleanup,  /* no cleanup needed */
11383      s_geometry    /* tell the geometry of the disk */
11384    };
11385
11386
11387    /*============================================================================*
11388     *                              aha_scsi_task                                 *
11389     *============================================================================*/
```

```
11390   PUBLIC void aha_scsi_task()
11391   {
11392   /* Set target and logical unit numbers, then call the generic main loop. */
11393     int i;
11394     struct scsi *sp;
11395     long v;
11396     char *name;
11397     static char fmt[] = "d,d";
11398
11399     for (i = 0; i < MAX_DEVICES; i++) {
11400           (void) s_prepare(i * DEV_PER_DRIVE);
11401           sp = s_sp;
11402
11403           /* Look into the environment for special parameters. */
11404           name = s_name();
11405
11406           v = i;
11407           (void) env_parse(name, fmt, 0, &v, 0L, 7L);
11408           sp->targ = v;
11409
11410           v = 0;
11411           (void) env_parse(name, fmt, 1, &v, 0L, 7L);
11412           sp->lun = v;
11413     }
11414     driver_task(&s_dtab);
11415   }


11418   /*============================================================================*
11419    *                              s_prepare                                     *
11420    *============================================================================*/
11421   PRIVATE struct device *s_prepare(device)
11422   int device;
11423   {
11424   /* Prepare for I/O on a device. */
11425
11426     rq->count = 0;        /* no requests as yet */
11427     s_must = TRUE;        /* the first transfers must be done */
11428     s_buf_blk = -1;       /* invalidate s_buf_blk */
11429
11430     if (device < NR_DISKDEVS) {                  /* sd0, sd1, ... */
11431           s_type = TYPE_SD;
11432           s_sp = &scsi[device / DEV_PER_DRIVE];
11433           s_dv = &s_sp->part[device % DEV_PER_DRIVE];
11434     } else
11435     if ((unsigned) (device - MINOR_hd1a) < NR_SUBDEVS) {  /* sd1a, sd1b, ... */
11436           device -= MINOR_hd1a;
11437           s_type = TYPE_SD;
11438           s_sp = &scsi[device / SUB_PER_DRIVE];
11439           s_dv = &s_sp->subpart[device % SUB_PER_DRIVE];
11440     } else
11441     if ((unsigned) (device - MINOR_st0) < NR_TAPEDEVS) {  /* nrst0, rst0, ... */
11442           device -= MINOR_st0;
11443           s_type = device & 1 ? TYPE_RST : TYPE_NRST;
11444           s_sp = &scsi[device >> 1];
11445           s_dv = &s_sp->dummypart;
11446     } else {
11447           return(NIL_DEV);
11448     }
11449
```

```
11450       return(s_dv);
11451     }


11454     /*==========================================================================*
11455      *                              s_name                                      *
11456      *==========================================================================*/
11457     PRIVATE char *s_name()
11458     {
11459     /* Return a name for the current device. */
11460       static char name[] = "sd35";
11461       int n = (s_sp - scsi);
11462
11463       switch (s_type) {
11464       case TYPE_SD:                 /* Disk device: sd* */
11465             name[1] = 'd';
11466             n *= DEV_PER_DRIVE;
11467             break;
11468       case TYPE_RST:                /* Tape device: st* */
11469       case TYPE_NRST:
11470             name[1] = 't';
11471             break;
11472       }
11473       if (n < 10) {
11474             name[2] = '0' + n;
11475             name[3] = 0;
11476       } else {
11477             name[2] = '0' + n / 10;
11478             name[3] = '0' + n % 10;
11479       }
11480       return name;
11481     }


11484     /*==========================================================================*
11485      *                              s_do_open                                   *
11486      *==========================================================================*/
11487     PRIVATE int s_do_open(dp, m_ptr)
11488     struct driver *dp;
11489     message *m_ptr;
11490     {
11491       struct scsi *sp;
11492       int r;
11493
11494       if (aha_irq == 0 && !aha_reset()) return(EIO); /* no controller, forget it */
11495
11496       if (s_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
11497       sp = s_sp;
11498
11499       if ((r = scsi_probe()) != OK) return(r);
11500
11501       if (sp->state & S_RDONLY && m_ptr->COUNT & W_BIT) return(EACCES);
11502
11503       switch (sp->devtype) {
11504       case SCSI_DEVDISK:
11505       case SCSI_DEVWORM:
11506       case SCSI_DEVCDROM:
11507       case SCSI_DEVOPTICAL:
11508             /* Read partition tables on first open. */
11509             if (sp->open_ct == 0) {
```

```
11510                   partition(&s_dtab, (int) (sp-scsi) * DEV_PER_DRIVE, P_PRIMARY);
11511           }
11512           break;
11513     case SCSI_DEVTAPE:
11514           /* Make sure tape is not already open. */
11515           if (sp->open_ct > 0) return(EBUSY);
11516
11517           sp->open_mode = m_ptr->COUNT;
11518           /* If open(..., O_WRONLY) then write a filemark on close even if no
11519            * write is done.
11520            */
11521           sp->need_eof = ((sp->open_mode & (R_BIT|W_BIT)) == W_BIT);
11522           break;
11523     }
11524     sp->open_ct++;
11525     return(OK);
11526   }


11529   /*============================================================================*
11530    *                              scsi_probe                                    *
11531    *============================================================================*/
11532   PRIVATE int scsi_probe()
11533   {
11534   /* See if a device exists and if it is ready. */
11535     struct scsi *sp = s_sp;
11536     sense_t *sense;
11537     int r, key;
11538
11539     /* Something out there? */
11540     if ((r = scsi_sense()) != OK) {
11541           if (sp->state & S_PRESENT) {
11542                   printf("%s: offline\n", s_name());
11543                   sp->state = 0;
11544           }
11545           return(r);
11546     }
11547
11548     if (!(sp->state & S_PRESENT)) {
11549           /* First contact with a new device, what type is it? */
11550
11551           if ((r = scsi_inquiry()) != OK) return(r);
11552
11553           sp->devtype = inqdata.devtype;
11554     }
11555
11556     if (!(sp->state & S_READY)) {
11557           /* If it's a disk: start it, if it's a tape: load it. */
11558           (void) scsi_simple(SCSI_STRTSTP, 1);
11559     }
11560
11561     /* See if the unit is ready for I/O.  A disk may be spinning up, a
11562      * floppy or tape drive may be empty.
11563      */
11564     while ((key = scsi_simple(SCSI_UNITRDY, 0)) != SENSE_NO_SENSE) {
11565           /* Not ready, why? */
11566
11567           sp->state &= ~S_READY;
11568
11569           switch (key) {
```

```
11570           case SENSE_UNIT_ATT:
11571                   /* A media change or something, try again. */
11572                   break;
11573           case SENSE_NOT_READY:
11574                   /* Look at the additional sense data to see why it isn't
11575                    * ready.
11576                    */
11577                   sense = &ccb_sense(rq);
11578                   switch ((sense->add_code << 8) | sense->add_qual) {
11579                   case 0x0401:
11580                           /* "It is becoming ready."  Fine, we wait. */
11581                           milli_delay(1000);
11582                           break;
11583                   case 0x0402:
11584                           /* "Initialization command required."  So we tell it
11585                            * to spin up.
11586                            */
11587                           if (scsi_simple(SCSI_STRTSTP, 1) != SENSE_NO_SENSE)
11588                                   return(EIO);
11589                           break;
11590                   case 0x0403:
11591                           /* "Manual intervention required." */
11592                   case 0x3A00:
11593                           /* "No media present." */
11594                           printf("%s: no media loaded\n", s_name());
11595                           return(EIO);
11596                   default:
11597                           /* For some reason it is not usable. */
11598                           printf("%s: not ready\n", s_name());
11599                           return(EIO);
11600                   }
11601                   break;
11602           default:
11603                   /* The device is in some odd state.  */
11604                   if (key != SENSE_NOT_READY) {
11605                           printf("%s: hardware error\n", s_name());
11606                           return(EIO);
11607                   }
11608           }
11609   }
11610
11611   if (!(sp->state & S_PRESENT)) {
11612           /* Do the inquiry again, the message may have changed. */
11613           if (scsi_inquiry() != OK) return(EIO);
11614
11615           /* Tell what kind of device it is we have found. */
11616
11617           printf("%s: %-7s %.48s\n",
11618                   s_name(),
11619                   inqdata.devtype > SCSI_DEVMAX ? "UNKNOWN"
11620                                   : scsi_devstr[inqdata.devtype],
11621                   inqdata.vendor /* + product + revision + extra */);
11622   }
11623
11624   if (!(sp->state & S_READY)) {
11625           /* Get the geometry, limits, etc. */
11626
11627           switch (sp->devtype) {
11628           case SCSI_DEVDISK:
11629           case SCSI_DEVWORM:
```

```
11630           case SCSI_DEVCDROM:
11631           case SCSI_DEVOPTICAL:
11632                   if (scsi_ndisk() != OK) return(EIO);
11633                   break;
11634           case SCSI_DEVTAPE:
11635                   if (scsi_ntape() != OK) return(EIO);
11636                   break;
11637           default:
11638                   printf("%s: unsupported\n", s_name());
11639                   return(EIO);
11640           }
11641     }
11642     return(OK);
11643   }


11646   /*==========================================================================*
11647    *                              scsi_sense                                  *
11648    *==========================================================================*/
11649   PRIVATE int scsi_sense()
11650   {
11651     int key;
11652     sense_t *sense = (sense_t *) tmp_buf;
11653
11654     /* Do a request sense to find out if a target exists or to check out
11655      * a unit attention condition.
11656      */
11657     key = scsi_simple(SCSI_REQSENSE, sizeof(sense_t));
11658
11659     if (rq->ccb.hastat == HST_TIMEOUT) return(ENXIO);     /* nothing there */
11660     if (rq->ccb.hastat != 0) return(EIO);         /* something very bad */
11661
11662     /* There is something out there for sure. */
11663     if (key == SENSE_UNIT_ATT || sense_key(sense->key) == SENSE_UNIT_ATT) {
11664           /* Device is in a "look at me" state, probably changed media. */
11665           s_sp->state &= ~S_READY;
11666     }
11667     return(OK);
11668   }


11671   /*==========================================================================*
11672    *                              scsi_inquiry                                *
11673    *==========================================================================*/
11674   PRIVATE int scsi_inquiry()
11675   {
11676     /* Prefill with nulls. */
11677     memset(tmp_buf, '\0', sizeof(inquiry_t));
11678
11679     /* Do a SCSI inquiry. */
11680     if (scsi_simple(SCSI_INQUIRY, sizeof(inquiry_t)) != SENSE_NO_SENSE)
11681           return(EIO);
11682     inqdata = * (inquiry_t *) tmp_buf;
11683
11684     if (inqdata.len == 0) {
11685           /* The device doesn't return meaningful text fields. */
11686           strcpy(inqdata.vendor, "(unknown)");
11687     }
11688
11689     /* The top three bits of devtype must be zero for the lun to exist. */
```

```
11690     if ((inqdata.devtype & 0xE0) != 0) return(ENXIO);
11691
11692     return(OK);
11693   }


11696   /*============================================================================*
11697    *                              scsi_ndisk                                    *
11698    *============================================================================*/
11699   PRIVATE int scsi_ndisk()
11700   {
11701   /* Gather disk data, capacity and block size. */
11702
11703     struct scsi *sp = s_sp;
11704     unsigned long capacity = -1, block_size = SECTOR_SIZE;
11705     byte *buf = tmp_buf;
11706
11707     /* Minor device type must be for a disk. */
11708     if (s_type != TYPE_SD) return(EIO);
11709
11710     if (sp->devtype == SCSI_DEVCDROM) {
11711           /* Read-only by definition. */
11712           sp->state |= S_RDONLY;
11713     } else {
11714           /* SCSI modesense to find out if the disk is write protected. */
11715           if (scsi_simple(SCSI_MDSENSE, 255) != SENSE_NO_SENSE) return(EIO);
11716
11717           /* Write protected? */
11718           sp->state &= ~S_RDONLY;
11719           if (buf[2] & 0x80) sp->state |= S_RDONLY;
11720
11721           /* Don't write a worm disk, not wise at the moment. */
11722           if (sp->devtype == SCSI_DEVWORM) sp->state |= S_RDONLY;
11723     }
11724
11725     /* Get drive capacity and block size. */
11726     group1();
11727     rq->ccb.opcode = CCB_INIT;
11728     ccb_cmd1(rq).scsi_op = SCSI_CAPACITY;
11729
11730     if (scsi_command(tmp_phys, 8) == SENSE_NO_SENSE) {
11731           capacity = b2h32(buf + 0) + 1;
11732           block_size = b2h32(buf + 4);
11733           printf("%s: capacity %lu x %lu bytes\n",
11734                                           s_name(), capacity, block_size);
11735     } else {
11736           printf("%s: unknown capacity\n", s_name());
11737     }
11738
11739     /* We do not believe block sizes over 4 kb. */
11740     if (block_size > 4096) {
11741           printf("%s: can't handle %lu byte blocks\n", s_name(), block_size);
11742           return(EIO);
11743     }
11744
11745     sp->block_size = block_size;
11746   #if _WORD_SIZE > 2
11747     /* Keep it within reach of a group 0 command. */
11748     sp->count_max = 0x100 * block_size;
11749   #else
```

```
11750       sp->count_max = block_size > UINT_MAX/0x100 ? UINT_MAX : 0x100 * block_size;
11751     #endif
11752
11753       /* The fun ends at 4GB. */
11754       if (capacity > ((unsigned long) -1) / block_size)
11755             sp->part[0].dv_size = -1;
11756       else
11757             sp->part[0].dv_size = capacity * block_size;
11758
11759       /* Finally we recognize its existence. */
11760       sp->state |= S_PRESENT|S_READY;
11761
11762       return(OK);
11763     }


11766     /*============================================================================*
11767      *                              scsi_ntape                                    *
11768      *============================================================================*/
11769     PRIVATE int scsi_ntape()
11770     {
11771     /* Gather tape data, block limits, fixed block size or not. */
11772       struct scsi *sp = s_sp;
11773       unsigned minblk;
11774       unsigned long maxblk;
11775       byte *buf = tmp_buf;
11776
11777       /* Minor device type must be for a tape. */
11778       if (s_type != TYPE_RST && s_type != TYPE_NRST) return(EIO);
11779
11780       /* Read limits. */
11781       if (scsi_simple(SCSI_RDLIMITS, 6) != SENSE_NO_SENSE) return(EIO);
11782       minblk = b2h16(buf + 4);
11783       maxblk = b2h24(buf + 1);
11784
11785       printf("%s: limits: min block len %u, max block len %lu\n",
11786             s_name(), minblk, maxblk);
11787
11788       if (sp->state & S_PRESENT) {
11789             /* Keep the current block size. */
11790             if (sp->tfixed) minblk= maxblk= sp->block_size;
11791       }
11792
11793       sp->tstat.mt_dsreg = DS_OK;
11794       sp->tstat.mt_erreg = 0;
11795       sp->tstat.mt_fileno = 0;
11796       sp->tstat.mt_blkno = 0;
11797       sp->tstat.mt_resid = 0;
11798
11799       if (minblk == maxblk) {
11800             /* Fixed block length. */
11801             sp->tfixed = TRUE;
11802             sp->block_size = minblk;
11803             sp->tstat.mt_blksize = minblk;
11804             sp->count_max = UINT_MAX;
11805       } else {
11806             /* Variable block length. */
11807             sp->tfixed = FALSE;
11808             sp->block_size = 1;
11809             sp->tstat.mt_blksize = 0;
```

```
11810           sp->count_max = maxblk == 0 ? UINT_MAX : maxblk;
11811       }
11812
11813       /* SCSI modesense. */
11814       if (scsi_simple(SCSI_MDSENSE, 255) != SENSE_NO_SENSE) return(EIO);
11815
11816       /* Write protected? */
11817       sp->state &= ~S_RDONLY;
11818       if (buf[2] & 0x80) sp->state |= S_RDONLY;
11819
11820       /* Density and block size. */
11821       if (buf[3] >= 8) {
11822           printf("%s: density 0x%02x, nblocks %lu, block len ",
11823                   s_name(),
11824                   buf[4],
11825                   b2h24(buf + 4 + 1));
11826           printf(sp->tfixed ? "%lu\n" : "variable\n", b2h24(buf + 4 + 5));
11827       }
11828
11829       sp->state |= S_PRESENT|S_READY;
11830       return(OK);
11831   }


11834   /*============================================================================*
11835    *                              s_schedule                                    *
11836    *============================================================================*/
11837   PRIVATE int s_schedule(proc_nr, iop)
11838   int proc_nr;                    /* process doing the request */
11839   struct iorequest_s *iop;        /* pointer to read or write request */
11840   {
11841   /* Gather I/O requests on consecutive blocks so they may be read/written
11842    * in one SCSI command using scatter/gather DMA.
11843    */
11844     struct scsi *sp = s_sp;
11845     int r, opcode, spanning;
11846     unsigned nbytes, count;
11847     unsigned long pos;
11848     phys_bytes user_phys, dma_phys;
11849     static unsigned dma_count;
11850     static struct iorequest_s **iopp;     /* to add I/O request pointers */
11851     static phys_bytes dma_last;   /* address of end of the last added entry */
11852
11853     /* This many bytes to read/write */
11854     nbytes = iop->io_nbytes;
11855
11856     /* From/to this position on the device */
11857     pos = iop->io_position;
11858
11859     /* To/from this user address */
11860     user_phys = numap(proc_nr, (vir_bytes) iop->io_buf, nbytes);
11861     if (user_phys == 0) return(iop->io_nbytes = EINVAL);
11862
11863     /* Read or write? */
11864     opcode = iop->io_request & ~OPTIONAL_IO;
11865
11866     switch (sp->devtype) {
11867     case SCSI_DEVCDROM:
11868     case SCSI_DEVWORM:
11869     case SCSI_DEVDISK:
```

```
11870       case SCSI_DEVOPTICAL:
11871             /* Which block on disk and how close to EOF? */
11872             if (pos >= s_dv->dv_size) return(OK);           /* At EOF */
11873             if (pos + nbytes > s_dv->dv_size) nbytes = s_dv->dv_size - pos;
11874             pos += s_dv->dv_base;
11875
11876             if ((nbytes % sp->block_size) != 0 || (pos % sp->block_size) != 0) {
11877                     /* Not on a device block boundary.  CD-ROM? */
11878                     return(s_rdcdrom(proc_nr, iop, pos, nbytes, user_phys));
11879             }
11880             break;
11881
11882       case SCSI_DEVTAPE:
11883             if ((nbytes % sp->block_size) != 0)
11884                     return(iop->io_nbytes = EINVAL);
11885
11886             /* Old error condition? */
11887             if (sp->tstat.mt_dsreg == DS_ERR) return(iop->io_nbytes = EIO);
11888
11889             if (opcode == DEV_READ && sp->at_eof) return(OK);
11890
11891             s_nextpos = pos = 0;    /* pos is ignored */
11892             break;
11893
11894       default:
11895             return(iop->io_nbytes = EIO);
11896       }
11897
11898       /* Probe a device that isn't ready. */
11899       if (!(sp->state & S_READY) && scsi_probe() != OK) return(EIO);
11900
11901       if (rq->count > 0 && pos != s_nextpos) {
11902             /* This new request can't be chained to the job being built. */
11903             if ((r = s_finish()) != OK) return(r);
11904       }
11905
11906       /* The next consecutive block starts at byte position... */
11907       s_nextpos = pos + nbytes;
11908
11909       spanning = FALSE;     /* set if a request spans several DMA vectors */
11910
11911       /* While there are "unscheduled" bytes in the request: */
11912       do {
11913             dma_phys = user_phys;
11914
11915             if (rq->count > 0 && (
11916                     rq->count == sp->count_max
11917                     || rq->dmaptr == rq->dmalimit
11918                     || !DMA_CHECK(dma_last, dma_phys)
11919             )) {
11920                     /* This request can not be added to the scatter/gather list. */
11921                     if ((r = s_finish()) != OK) return(r);
11922                     s_must = spanning;
11923
11924                     continue;       /* try again */
11925             }
11926
11927             if (rq->count == 0) {
11928                     /* The first request in a row, initialize. */
11929                     rq->pos = pos;
```

```
11930                   s_opcode = opcode;
11931                   iopp = rq->iov;
11932                   rq->dmaptr = rq->dmalist;
11933                   rq->retry = 2;
11934           }
11935
11936           count = nbytes;
11937
11938           /* Don't exceed the maximum transfer count. */
11939           if (rq->count + count > sp->count_max)
11940                   count = sp->count_max - rq->count;
11941
11942           /* New scatter/gather entry. */
11943           h2b24(rq->dmaptr->dataptr, dma_phys);
11944           h2b24(rq->dmaptr->datalen, (u32_t) (dma_count = count));
11945           rq->dmaptr++;
11946           dma_last = dma_phys + count;
11947
11948           /* Which I/O request? */
11949           *iopp++ = iop;
11950
11951           /* Update counters. */
11952           rq->count += count;
11953           pos += count;
11954           user_phys += count;
11955           nbytes -= count;
11956           if (!(iop->io_request & OPTIONAL_IO)) s_must = TRUE;
11957
11958           spanning = TRUE;        /* the rest of the request must be done */
11959     } while (nbytes > 0);
11960
11961     return(OK);
11962   }


11965   /*==========================================================================*
11966    *                              s_finish                                    *
11967    *==========================================================================*/
11968   PRIVATE int s_finish()
11969   {
11970   /* Send the I/O requests gathered in *rq to the host adapter. */
11971
11972     struct scsi *sp = s_sp;
11973     unsigned long block;
11974     struct iorequest_s **iopp, *iop;
11975     int key;
11976
11977     if (rq->count == 0) return(OK);       /* spurious finish */
11978
11979     show_req();
11980
11981     /* If all the requests are optional then don't do just a few. */
11982     if (!s_must && rq->count < 0x2000) {
11983           rq->count = 0;
11984           return(OK);
11985     }
11986
11987     iopp = rq->iov;
11988     iop = *iopp++;
11989
```

```
11990   retry:
11991     switch (sp->devtype) {
11992     case SCSI_DEVCDROM:
11993     case SCSI_DEVWORM:
11994     case SCSI_DEVDISK:
11995     case SCSI_DEVOPTICAL:
11996           /* A read or write SCSI command for a random access device. */
11997           block = rq->pos / sp->block_size;
11998
11999           if (block < (1L << 21)) {
12000                   /* We can use a group 0 command for small disks. */
12001                   group0();
12002                   rq->ccb.opcode = CCB_SCATTER;
12003                   ccb_cmd0(rq).scsi_op =
12004                           s_opcode == DEV_WRITE ? SCSI_WRITE : SCSI_READ;
12005                   h2b24(ccb_cmd0(rq).lba, block);
12006                   ccb_cmd0(rq).nblocks = rq->count / sp->block_size;
12007           } else {
12008                   /* Large disks require a group 1 command. */
12009                   group1();
12010                   rq->ccb.opcode = CCB_SCATTER;
12011                   ccb_cmd1(rq).scsi_op =
12012                           s_opcode == DEV_WRITE ? SCSI_WRITE1 : SCSI_READ1;
12013                   h2b32(ccb_cmd1(rq).lba, block);
12014                   h2b16(ccb_cmd1(rq).nblocks, rq->count / sp->block_size);
12015           }
12016
12017           key = scsi_command(0L, 0);
12018
12019           if (key == SENSE_NO_SENSE) {
12020                   /* fine */;
12021           } else
12022           if (key == SENSE_UNIT_ATT || key == SENSE_ABORTED_CMD) {
12023                   /* Check condition?  Bus reset most likely. */
12024                   /* Aborted command?  Maybe retrying will help. */
12025                   if (--rq->retry > 0) goto retry;
12026                   return(iop->io_nbytes = EIO);
12027           } else
12028           if (key == SENSE_RECOVERED) {
12029                   /* Disk drive managed to recover from a read error. */
12030                   printf("%s: soft read error at block %lu (recovered)\n",
12031                           s_name(), b2h32(ccb_sense(rq).info));
12032                   key = SENSE_NO_SENSE;
12033                   break;
12034           } else {
12035                   /* A fatal error occurred, bail out. */
12036                   return(iop->io_nbytes = EIO);
12037           }
12038           break;
12039
12040     case SCSI_DEVTAPE:
12041           /* A read or write SCSI command for a sequential access device. */
12042           group0();
12043           rq->ccb.opcode = CCB_SCATTER;
12044           ccb_cmd0(rq).scsi_op = s_opcode == DEV_WRITE ? SCSI_WRITE : SCSI_READ;
12045           ccb_cmd0(rq).fixed = sp->tfixed;
12046           h2b24(ccb_cmd0(rq).trlength, rq->count / sp->block_size);
12047
12048           key = scsi_command(0L, 0);
12049
```

```
12050           if (key != SENSE_NO_SENSE) {
12051                   /* Either at EOF or EOM, or an I/O error. */
12052
12053                   if (sense_eof(key) || sense_eom(key)) {
12054                           /* Not an error, but EOF or EOM. */
12055                           sp->at_eof = TRUE;
12056                           sp->tstat.mt_dsreg = DS_EOF;
12057
12058                           /* The residual tells how much has not been read. */
12059                           rq->count -= sp->tstat.mt_resid * sp->block_size;
12060
12061                           if (sense_eof(key)) {
12062                                   /* Went over a filemark. */
12063                                   sp->tstat.mt_blkno = !sp->tfixed ? -1 :
12064                                           - (int) (rq->count / sp->block_size);
12065                                   sp->tstat.mt_fileno++;
12066                           }
12067                   }
12068                   if (sense_ili(key)) {
12069                           /* Incorrect length on a variable block length tape. */
12070
12071                           if (sp->tstat.mt_resid <= 0) {
12072                                   /* Large block could not be read. */
12073                                   return(iop->io_nbytes = EIO);
12074                           }
12075                           /* Small block read, this is ok. */
12076                           rq->count -= sp->tstat.mt_resid;
12077                           sp->tstat.mt_dsreg = DS_OK;
12078                   }
12079                   if (key == SENSE_RECOVERED) {
12080                           /* Tape drive managed to recover from an error. */
12081                           printf("%s: soft %s error (recovered)\n",
12082                                   s_name(),
12083                                   s_opcode == DEV_READ ? "read" : "write");
12084                           key = SENSE_NO_SENSE;
12085                           sp->tstat.mt_dsreg = DS_OK;
12086                   }
12087                   if (sp->tstat.mt_dsreg == DS_ERR) {
12088                           /* Error was fatal. */
12089                           return(iop->io_nbytes = EIO);
12090                   }
12091           } else {
12092                   sp->tstat.mt_dsreg = DS_OK;
12093           }
12094           if (!sp->tfixed) {
12095                   /* Variable block length tape reads record by record. */
12096                   sp->tstat.mt_blkno++;
12097           } else {
12098                   /* Fixed length tape, multiple blocks transferred. */
12099                   sp->tstat.mt_blkno += rq->count / sp->block_size;
12100           }
12101           sp->need_eof = (s_opcode == DEV_WRITE);
12102           break;
12103
12104   default:
12105           assert(0);
12106   }
12107
12108   /* Remove bytes transferred from the I/O requests. */
12109   for (;;) {
```

```
12110           if (rq->count > iop->io_nbytes) {
12111                   rq->count -= iop->io_nbytes;
12112                   iop->io_nbytes = 0;
12113           } else {
12114                   iop->io_nbytes -= rq->count;
12115                   rq->count = 0;
12116                   break;
12117           }
12118           iop = *iopp++;
12119     }
12120     return(key == SENSE_NO_SENSE ? OK : EIO);     /* may return EIO for EOF */
12121   }


12124   /*==========================================================================*
12125    *                              s_rdcdrom                                   *
12126    *==========================================================================*/
12127   PRIVATE int s_rdcdrom(proc_nr, iop, pos, nbytes, user_phys)
12128   int proc_nr;                    /* process doing the request */
12129   struct iorequest_s *iop;        /* pointer to read or write request */
12130   unsigned long pos;              /* byte position */
12131   unsigned nbytes;                /* number of bytes */
12132   phys_bytes user_phys;           /* user address */
12133   {
12134   /* CD-ROM's have a basic block size of 2k.  We could try to set a smaller
12135    * virtual block size, but many don't support it.  So we use this function.
12136    */
12137     struct scsi *sp = s_sp;
12138     int r, key;
12139     unsigned offset, count;
12140     unsigned long block;
12141
12142     /* Only do reads. */
12143     if ((iop->io_request & ~OPTIONAL_IO) != DEV_READ)
12144           return(iop->io_nbytes = EINVAL);
12145
12146     /* Finish any outstanding I/O. */
12147     if ((r = s_finish()) != OK) return(r);
12148
12149     do {
12150           /* Probe a device that isn't ready. */
12151           if (!(sp->state & S_READY) && scsi_probe() != OK) return(EIO);
12152
12153           block = pos / sp->block_size;
12154           if (block == s_buf_blk) {
12155                   /* Some of the requested bytes are in the buffer. */
12156                   offset = pos % sp->block_size;
12157                   count = sp->block_size - offset;
12158                   if (count > nbytes) count = nbytes;
12159                   phys_copy(tmp_phys + offset, user_phys, (phys_bytes) count);
12160                   pos += count;
12161                   user_phys += count;
12162                   nbytes -= count;
12163                   iop->io_nbytes -= count;
12164           } else {
12165                   /* Read a block that contains (some of) the bytes wanted. */
12166                   rq->retry = 2;
12167                   do {
12168                           group1();
12169                           rq->ccb.opcode = CCB_INIT;
```

```
12170                           ccb_cmd1(rq).scsi_op = SCSI_READ1;
12171                           h2b32(ccb_cmd1(rq).lba, block);
12172                           h2b16(ccb_cmd1(rq).nblocks, 1);
12173                           key = scsi_command(tmp_phys, sp->block_size);
12174                   } while (key == SENSE_UNIT_ATT && --rq->retry > 0);
12175
12176                   if (key != SENSE_NO_SENSE) return(iop->io_nbytes = EIO);
12177
12178                   s_buf_blk = block;      /* remember block in buffer */
12179           }
12180     } while (nbytes > 0);
12181     return(OK);
12182   }


12185   /*============================================================================*
12186    *                               s_do_close                                   *
12187    *============================================================================*/
12188   PRIVATE int s_do_close(dp, m_ptr)
12189   struct driver *dp;
12190   message *m_ptr;
12191   {
12192     struct scsi *sp;
12193
12194     if (s_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
12195     sp = s_sp;
12196
12197     sp->open_ct--;
12198
12199     /* Disks and such don't make trouble. */
12200     if (sp->devtype != SCSI_DEVTAPE) return(OK);
12201
12202     sp->at_eof = FALSE;
12203
12204     /* Write filemark if writes have been done. */
12205     if (sp->need_eof && sp->tstat.mt_dsreg != DS_ERR) {
12206           if (scsi_simple(SCSI_WREOF, 1) != SENSE_NO_SENSE) {
12207                   printf("%s: failed to add filemark\n", s_name());
12208           } else {
12209                   sp->tstat.mt_dsreg = DS_OK;
12210                   sp->tstat.mt_blkno = 0;
12211                   sp->tstat.mt_fileno++;
12212           }
12213     }
12214
12215     /* Rewind if rewind device. */
12216     if (s_type == TYPE_RST) {
12217           if (scsi_simple(SCSI_REWIND, 1) != SENSE_NO_SENSE) {
12218                   printf("%s: failed to rewind\n", s_name());
12219           } else {
12220                   sp->tstat.mt_dsreg = DS_OK;
12221                   sp->tstat.mt_blkno = 0;
12222                   sp->tstat.mt_fileno = 0;
12223           }
12224     }
12225     return(OK);
12226   }


12229   /*============================================================================*
```

```
12230  *                              s_do_ioctl                                *
12231  *========================================================================*/
12232 PRIVATE int s_do_ioctl(dp, m_ptr)
12233 struct driver *dp;
12234 message *m_ptr;
12235 {
12236   struct scsi *sp;
12237
12238   if (s_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
12239   sp = s_sp;
12240
12241   /* Ioctls are device specific. */
12242   switch (sp->devtype) {
12243   case SCSI_DEVDISK:
12244   case SCSI_DEVWORM:
12245   case SCSI_DEVCDROM:
12246   case SCSI_DEVOPTICAL:
12247         if (m_ptr->REQUEST == DIOCEJECT) {
12248                 /* Eject disk. */
12249                 if (sp->open_ct > 1) return(EBUSY);
12250
12251                 /* Send a start/stop command with code 2: stop and eject. */
12252                 if (scsi_simple(SCSI_STRTSTP, 2) != SENSE_NO_SENSE)
12253                         return(EIO);
12254                 return(OK);
12255         }
12256         /* Call the common code for disks and disk like devices. */
12257         return(do_diocntl(dp, m_ptr));
12258
12259   default:
12260         return(ENOTTY);
12261
12262   case SCSI_DEVTAPE:
12263         break;
12264   }
12265   /* Further ioctls are for tapes. */
12266
12267   if (m_ptr->REQUEST == MTIOCTOP) {
12268         struct mtop op;
12269         phys_bytes op_phys;
12270         long delta;
12271         int key;
12272         byte *buf = tmp_buf;
12273
12274         /* Basic tape commands: rewind, space, write eof marks, ... */
12275         op_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, sizeof(op));
12276         if (op_phys == 0) return(EINVAL);
12277         phys_copy(op_phys, vir2phys(&op), (phys_bytes) sizeof(op));
12278
12279         switch(op.mt_op) {
12280         case MTREW:
12281         case MTOFFL:
12282         case MTRETEN:
12283         case MTFSF:
12284         case MTFSR:
12285         case MTBSF:
12286         case MTBSR:
12287         case MTEOM:
12288                 /* Write an EOF mark before spacing. */
12289                 if (sp->need_eof && sp->tstat.mt_dsreg != DS_ERR) {
```

```
12290                           if (scsi_simple(SCSI_WREOF, 1) != SENSE_NO_SENSE)
12291                                   return(EIO);
12292                           sp->tstat.mt_blkno = 0;
12293                           sp->tstat.mt_fileno++;
12294                           sp->need_eof = FALSE;
12295                   }
12296                   sp->at_eof = FALSE;
12297           }
12298
12299           switch(op.mt_op) {
12300           case MTREW:
12301           case MTOFFL:
12302           case MTRETEN:
12303           case MTERASE:
12304                   /* Rewind, Offline, Retension, Erase. */
12305                   switch(op.mt_op) {
12306                   case MTOFFL:
12307                           if (scsi_simple(SCSI_LOADUNLD, 0) != SENSE_NO_SENSE)
12308                                   return(EIO);
12309                           sp->state &= ~S_READY;
12310                           break;
12311                   case MTRETEN:
12312                           if (scsi_simple(SCSI_LOADUNLD, 3) != SENSE_NO_SENSE)
12313                                   return(EIO);
12314                           break;
12315                   case MTERASE:
12316                           if (scsi_simple(SCSI_REWIND, 0) != SENSE_NO_SENSE)
12317                                   return(EIO);
12318                           if (scsi_simple(SCSI_ERASE, 1) != SENSE_NO_SENSE)
12319                                   return(EIO);
12320                           /* Rewind once more. */
12321                           /*FALL THROUGH*/
12322                   case MTREW:
12323                           if (scsi_simple(SCSI_REWIND, 0) != SENSE_NO_SENSE)
12324                                   return(EIO);
12325                   }
12326                   sp->tstat.mt_dsreg = DS_OK;
12327                   sp->tstat.mt_blkno = 0;
12328                   sp->tstat.mt_fileno = 0;
12329                   break;
12330           case MTFSF:
12331           case MTFSR:
12332           case MTBSF:
12333           case MTBSR:
12334                   if (sp->tstat.mt_dsreg == DS_ERR) return(EIO);
12335                   group0();
12336                   rq->ccb.opcode = CCB_INIT;
12337                   ccb_cmd0(rq).scsi_op = SCSI_SPACE;
12338                   delta = op.mt_count;
12339                   if (op.mt_op == MTBSR) delta = -delta;
12340                   if (op.mt_op == MTBSF) delta = -delta - 1;
12341                   h2b24(ccb_cmd0(rq).trlength, delta);
12342                   ccb_cmd0(rq).fixed =
12343                                   op.mt_op == MTFSR || op.mt_op == MTBSR ? 0 : 1;
12344                   if ((key = scsi_command(0L, 0)) != SENSE_NO_SENSE) {
12345                           if (sense_key(key) != SENSE_NO_SENSE) return(EIO);
12346
12347                           if (sense_eom(key)) {
12348                                   /* Banging into end of tape. */
12349                                   if (op.mt_op == MTBSF || op.mt_op == MTBSR) {
```

```
12350                                   /* Backspacing to start of tape. */
12351                                   sp->tstat.mt_dsreg = DS_EOF;
12352                                   sp->tstat.mt_blkno = 0;
12353                                   sp->tstat.mt_fileno = 0;
12354                           } else {
12355                                   /* Not forwards please! */
12356                                   return(EIO);
12357                           }
12358                   }
12359                   if (sense_eof(key)) {
12360                           /* Reaching a filemark. */
12361                           sp->tstat.mt_dsreg = DS_EOF;
12362                           sp->at_eof = TRUE;
12363                           if (op.mt_op == MTFSR) {
12364                                   /* Forwards. */
12365                                   sp->tstat.mt_blkno = 0;
12366                                   sp->tstat.mt_fileno++;
12367                           } else {
12368                                   /* Backwards (bad idea!) */
12369                                   sp->tstat.mt_blkno = -1;
12370                                   sp->tstat.mt_fileno--;
12371                           }
12372                   }
12373           } else {
12374                   if (op.mt_op == MTFSR || op.mt_op == MTBSR) {
12375                           sp->tstat.mt_blkno += delta;
12376                   } else {
12377                           sp->tstat.mt_blkno = 0;
12378                           sp->tstat.mt_fileno += delta;
12379                   }
12380                   if (op.mt_op == MTBSF) {
12381                           /* n+1 backwards, and 1 forward. */
12382                           group0();
12383                           rq->ccb.opcode = CCB_INIT;
12384                           ccb_cmd0(rq).scsi_op = SCSI_SPACE;
12385                           h2b24(ccb_cmd0(rq).trlength, 1L);
12386                           ccb_cmd0(rq).fixed = 1;
12387                           if (scsi_command(0L, 0) != SENSE_NO_SENSE)
12388                                   return(EIO);
12389                           sp->tstat.mt_fileno++;
12390                   }
12391                   sp->tstat.mt_dsreg = DS_OK;
12392           }
12393           break;
12394   case MTWEOF:
12395           /* Write EOF marks. */
12396           if (sp->tstat.mt_dsreg == DS_ERR) return(EIO);
12397           if (op.mt_count < 0) return(EIO);
12398           if (op.mt_count == 0) return(OK);
12399           group0();
12400           rq->ccb.opcode = CCB_INIT;
12401           ccb_cmd0(rq).scsi_op = SCSI_WREOF;
12402           h2b24(ccb_cmd0(rq).trlength, op.mt_count);
12403           if (scsi_command(0L, 0) != SENSE_NO_SENSE) return(EIO);
12404           sp->tstat.mt_dsreg = DS_OK;
12405           sp->tstat.mt_blkno = 0;
12406           sp->tstat.mt_fileno += op.mt_count;
12407           sp->need_eof = FALSE;
12408           break;
12409   case MTEOM:
```

```
12410                   /* Forward space to end of media. */
12411                   if (sp->tstat.mt_dsreg == DS_ERR) return(EIO);
12412                   do {
12413                           group0();
12414                           rq->ccb.opcode = CCB_INIT;
12415                           ccb_cmd0(rq).scsi_op = SCSI_SPACE;
12416                           h2b24(ccb_cmd0(rq).trlength, 0x7FFFFF);
12417                           ccb_cmd0(rq).fixed = 1;
12418                           key = scsi_command(0L, 0);
12419                           sp->tstat.mt_blkno = 0;
12420                           sp->tstat.mt_fileno += 0x7FFFFF;
12421                           if (key != SENSE_NO_SENSE) {
12422                                   if (key != SENSE_BLANK_CHECK) return(EIO);
12423                                   sp->tstat.mt_fileno -= sp->tstat.mt_resid;
12424                           }
12425                   } while (key == SENSE_NO_SENSE);
12426                   sp->tstat.mt_dsreg = DS_OK;
12427                   break;
12428           case MTBLKZ:
12429           case MTMODE:
12430                   /* Select tape block size or tape density. */
12431
12432                   /* Rewind tape. */
12433                   if (scsi_simple(SCSI_REWIND, 0) != SENSE_NO_SENSE)
12434                           return(EIO);
12435
12436                   sp->tstat.mt_dsreg = DS_OK;
12437                   sp->tstat.mt_blkno = 0;
12438                   sp->tstat.mt_fileno = 0;
12439
12440                   if (op.mt_op == MTBLKZ && op.mt_count == 0) {
12441                           /* Request for variable block size mode. */
12442                           sp->tfixed = FALSE;
12443                           sp->block_size = 1;
12444                   } else {
12445                           /* First a modesense to get the current values. */
12446                           if (scsi_simple(SCSI_MDSENSE, 255) != SENSE_NO_SENSE)
12447                                   return(EIO);
12448
12449                           /* Must at least have one block descriptor. */
12450                           if (buf[3] < 8) return(EIO);
12451                           buf[0] = 0;
12452                           buf[1] = 0;
12453                           /* buf[2]: buffered mode & speed */
12454                           buf[3] = 8;
12455                           if (op.mt_op == MTMODE)         /* New density */
12456                                   buf[4 + 0] = op.mt_count;
12457                           /* buf[4 + 1]: number of blocks */
12458                           buf[4 + 4] = 0;
12459                           if (op.mt_op == MTBLKZ)         /* New block size */
12460                                   h2b24(buf + 4 + 5, (long) op.mt_count);
12461
12462                           /* Set the new density/blocksize. */
12463                           if (scsi_simple(SCSI_MDSELECT, 4+8) != SENSE_NO_SENSE)
12464                                   return(EIO);
12465                           if (op.mt_op == MTBLKZ) {
12466                                   sp->tfixed = TRUE;
12467                                   sp->block_size= op.mt_count;
12468                           }
12469                   }
```

```
12470                   sp->state &= ~S_READY;
12471                   if (scsi_probe() != OK) return(EIO);
12472                   break;
12473           default:
12474                   /* Not implemented. */
12475                   return(ENOTTY);
12476           }
12477     } else
12478     if (m_ptr->REQUEST == MTIOCGET) {
12479           /* Request tape status. */
12480           phys_bytes get_phys;
12481
12482           get_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
12483                                                           sizeof(sp->tstat));
12484           if (get_phys == 0) return(EINVAL);
12485
12486           if (sp->tstat.mt_dsreg == DS_OK) {
12487                   /* Old error data is never cleared (until now). */
12488                   sp->tstat.mt_erreg = 0;
12489                   sp->tstat.mt_resid = 0;
12490           }
12491           phys_copy(vir2phys(&sp->tstat), get_phys,
12492                                           (phys_bytes) sizeof(sp->tstat));
12493     } else {
12494           /* Not implemented. */
12495           return(ENOTTY);
12496     }
12497     return(OK);
12498   }


12501   /*==========================================================================*
12502    *                              scsi_simple                                 *
12503    *==========================================================================*/
12504   PRIVATE int scsi_simple(opcode, count)
12505   int opcode;                             /* SCSI opcode */
12506   int count;                              /* count or flag */
12507   {
12508   /* The average group 0 SCSI command with just a simple flag or count. */
12509
12510     vir_bytes len = 0;    /* Sometimes a buffer is used. */
12511
12512     group0();
12513     rq->ccb.opcode = CCB_INIT;
12514     ccb_cmd0(rq).scsi_op = opcode;
12515
12516     /* Fill in the count argument at the proper place. */
12517     switch (opcode) {
12518     case SCSI_REQSENSE:
12519     case SCSI_INQUIRY:
12520     case SCSI_MDSENSE:
12521     case SCSI_MDSELECT:
12522           ccb_cmd0(rq).nblocks = count;
12523           len = count;
12524           break;
12525
12526     case SCSI_STRTSTP:
12527       /* SCSI_LOADUNLD: (synonym) */
12528           ccb_cmd0(rq).nblocks = count;
12529           break;
```

```
12530
12531     case SCSI_RDLIMITS:
12532           len = count;
12533           break;
12534
12535     case SCSI_WREOF:
12536           h2b24(ccb_cmd0(rq).trlength, (long) count);
12537           break;
12538
12539     case SCSI_REWIND:
12540     case SCSI_ERASE:
12541           ccb_cmd0(rq).fixed = count;
12542           break;
12543     }
12544     return(scsi_command(tmp_phys, len));
12545   }


12548   /*============================================================================*
12549    *                              group0                                        *
12550    *============================================================================*/
12551   PRIVATE void group0()
12552   {
12553     /* Prepare the ccb for a group 0 SCSI command. */
12554
12555     rq->ccb.cmdlen = sizeof(cdb0_t);
12556
12557     /* Clear cdb to zeros the ugly way. */
12558     * (u32_t *) (rq->ccb.cmd + 0) = 0;
12559     * (u16_t *) (rq->ccb.cmd + 4) = 0;
12560   }


12563   /*============================================================================*
12564    *                              group1                                        *
12565    *============================================================================*/
12566   PRIVATE void group1()
12567   {
12568     rq->ccb.cmdlen = sizeof(cdb1_t);
12569     * (u32_t *) (rq->ccb.cmd + 0) = 0;
12570     * (u32_t *) (rq->ccb.cmd + 4) = 0;
12571     * (u16_t *) (rq->ccb.cmd + 8) = 0;
12572   }


12575   /*============================================================================*
12576    *                              scsi_command                                  *
12577    *============================================================================*/
12578   PRIVATE int scsi_command(data, len)
12579   phys_bytes data;
12580   vir_bytes len;
12581   {
12582   /* Execute a SCSI command and return the results.  Unlike most other routines,
12583    * this routine returns the sense key of a SCSI command instead of OK or EIO.
12584    */
12585     struct scsi *sp = s_sp;
12586     int key;
12587     message intr_mess;
12588
12589     rq->ccb.addrcntl = ccb_scid(s_sp->targ) | ccb_lun(s_sp->lun);
```

```
12590
12591     if (rq->ccb.opcode == CCB_SCATTER) {
12592           /* Device read/write; add checks and use scatter/gather vector. */
12593           rq->ccb.addrcntl |= s_opcode == DEV_READ ? CCB_INCHECK : CCB_OUTCHECK;
12594           data = vir2phys(rq->dmalist);
12595           len = (byte *) rq->dmaptr - (byte *) rq->dmalist;
12596           if (aha_model == AHA1540) {
12597                   /* A plain 1540 can't do s/g. */
12598                   rq->ccb.opcode = CCB_INIT;
12599                   data = b2h24(rq->dmalist[0].dataptr);
12600                   len = b2h24(rq->dmalist[0].datalen);
12601           }
12602     }
12603     h2b24(rq->ccb.datalen, (u32_t) len);
12604     h2b24(rq->ccb.dataptr, data);
12605     dump_scsi_cmd();
12606
12607     mailbox[0].status = AHA_MBOXSTART;
12608
12609     out_byte(AHA_DATAREG, AHACOM_STARTSCSI);  /* hey, you've got mail! */
12610
12611     /* Wait for the SCSI command to complete. */
12612     while (mailbox[1].status == AHA_MBOXFREE) {
12613           /* No mail yet, wait for an interrupt. */
12614           receive(HARDWARE, &intr_mess);
12615     }
12616     mailbox[1].status = AHA_MBOXFREE;     /* free up inbox */
12617
12618     /* Check the results of the operation. */
12619     if (rq->ccb.hastat != 0) {
12620           /* Weird host adapter status. */
12621           printf("%s: host adapter error 0x%02x%s\n", s_name(), rq->ccb.hastat,
12622                   rq->ccb.hastat == HST_TIMEOUT ? " (Selection timeout)" : "");
12623           errordump();
12624           if (sp->devtype == SCSI_DEVTAPE) sp->tstat.mt_dsreg = DS_ERR;
12625           memset((void *) &ccb_sense(rq), 0, sizeof(sense_t));
12626           return(SENSE_HARDWARE);
12627     }
12628
12629     if (rq->ccb.tarstat != 0) {
12630           /* A SCSI error has occurred. */
12631           sense_t *sense = &ccb_sense(rq);
12632
12633           if (sense->len < 2) {
12634                   /* No additional code and qualifier, zero them. */
12635                   sense->add_code = sense->add_qual = 0;
12636           }
12637
12638           /* Check sense data, report error if interesting. */
12639           if (rq->ccb.tarstat == TST_CHECK) {
12640                   if ((sense->errc & 0x7E) == 0x70) {
12641                           /* Standard SCSI error. */
12642                           key = sense->key;
12643                   } else {
12644                           /* Blame the vendor for any other nonsense. */
12645                           key = SENSE_VENDOR;
12646                   }
12647           } else {
12648                   if (rq->ccb.tarstat == TST_LUNBUSY) {
12649                           /* Logical unit is too busy to react... */
```

```
12650                           key = SENSE_NOT_READY;
12651                   } else {
12652                           /* The adapter shoudn't do this... */
12653                           key = SENSE_HARDWARE;
12654                   }
12655                   memset((void *) sense, 0, sizeof(sense_t));
12656           }
12657
12658           if (sense_serious(sense_key(key))) {
12659                   /* Something bad happened. */
12660                   printf("%s: error on command 0x%02x, ", s_name(),
12661                                                           rq->ccb.cmd[0]);
12662                   if (rq->ccb.tarstat != TST_CHECK) {
12663                           printf("target status 0x%02x\n", rq->ccb.tarstat);
12664                   } else {
12665                           printf("sense key 0x%02x (%s), additional 0x%02x%02x\n",
12666                                   sense->key,
12667                                   str_scsi_sense[sense_key(key)],
12668                                   sense->add_code, sense->add_qual);
12669                   }
12670                   errordump();
12671           }
12672
12673           if (sp->devtype == SCSI_DEVTAPE) {
12674                   /* Store details of tape error. */
12675                   sp->tstat.mt_dsreg = DS_ERR;
12676                   sp->tstat.mt_erreg = key;
12677                   sp->tstat.mt_resid = b2h32(sense->info);
12678           }
12679
12680           /* Keep only the ILI, EOM and EOF bits of key 0. */
12681           if (sense_key(key) != SENSE_NO_SENSE) key = sense_key(key);
12682
12683           return(key);
12684     }
12685     return(SENSE_NO_SENSE);
12686   }


12689   /*==========================================================================*
12690    *                              aha_command                                 *
12691    *==========================================================================*/
12692   PRIVATE void aha_command(outlen, outptr, inlen, inptr)
12693   int outlen, inlen;
12694   byte *outptr, *inptr;
12695   {
12696     /* Send a low level command to the host adapter. */
12697     int i;
12698
12699     /* Send command bytes. */
12700     for (i = 0; i < outlen; i++) {
12701           while (in_byte(AHA_STATREG) & AHA_CDF) {}       /* !! timeout */
12702           out_byte(AHA_DATAREG, *outptr++);
12703     }
12704
12705     /* Receive data bytes. */
12706     for (i = 0; i < inlen; i++) {
12707           while (!(in_byte(AHA_STATREG) & AHA_DF)
12708                   && !(in_byte(AHA_INTRREG) & AHA_HACC)) {}  /* !! timeout */
12709           *inptr++ = in_byte(AHA_DATAREG);
```

```
12710     }
12711
12712     /* Wait for command completion. */
12713     while (!(in_byte(AHA_INTRREG) & AHA_HACC)) {} /* !! timeout */
12714     out_byte(AHA_CNTLREG, AHA_IRST);      /* clear interrupt */
12715     if (aha_irq != 0) enable_irq(aha_irq);
12716
12717     /* !! should check status register here for invalid command */
12718   }


12721   /*===========================================================================*
12722    *                               aha_reset                                   *
12723    *===========================================================================*/
12724   PRIVATE int aha_reset()
12725   {
12726     int stat;
12727     int irq, bus_on, bus_off, tr_speed;
12728     unsigned sg_max;
12729     long v;
12730     static char aha0_env[] = "AHA0", aha_fmt[] = "x:d:d:x";
12731     byte cmd[5], haidata[4], getcdata[3], extbios[2];
12732     struct milli_state ms;
12733
12734     /* Get the configuration info from the environment. */
12735     v = AHA_BASEREG;
12736     if (env_parse(aha0_env, aha_fmt, 0, &v, 0x000L, 0x3FFL) == EP_OFF) return 0;
12737     aha_basereg = v;
12738
12739     v = 15;
12740     (void) env_parse(aha0_env, aha_fmt, 1, &v, 2L, 15L);
12741     bus_on = v;
12742
12743     v = 1;
12744     (void) env_parse(aha0_env, aha_fmt, 2, &v, 1L, 64L);
12745     bus_off = v;
12746
12747     v = 0x00;
12748     (void) env_parse(aha0_env, aha_fmt, 3, &v, 0x00L, 0xFFL);
12749     tr_speed = v;
12750
12751     /* Reset controller, wait for self test to complete. */
12752     out_byte(AHA_CNTLREG, AHA_HRST);
12753     milli_start(&ms);
12754     while ((stat = in_byte(AHA_STATREG)) & AHA_STST) {
12755           if (milli_elapsed(&ms) >= AHA_TIMEOUT) {
12756                   printf("aha0: AHA154x controller not responding\n");
12757                   return(0);
12758           }
12759     }
12760
12761     /* Check for self-test failure. */
12762     if ((stat & (AHA_DIAGF | AHA_INIT | AHA_IDLE | AHA_CDF | AHA_DF))
12763                                                 != (AHA_INIT | AHA_IDLE)) {
12764           printf("aha0: AHA154x controller failed self-test\n");
12765           return(0);
12766     }
12767
12768     /* !! maybe a santity check here: make sure IDLE and INIT are set? */
12769
```

```
12770     /* Get information about controller type and configuration. */
12771     cmd[0] = AHACOM_HAINQUIRY;
12772     aha_command(1, cmd, 4, haidata);
12773
12774     cmd[0] = AHACOM_GETCONFIG;
12775     aha_command(1, cmd, 3, getcdata);
12776
12777     /* First inquiry byte tells what type of board. */
12778     aha_model = haidata[0];
12779
12780     /* Unlock the 1540C or 1540CF's mailbox interface.  (This is to keep old
12781      * drivers from using the adapter if extended features are enabled.)
12782      */
12783     if (aha_model >= AHA1540C) {
12784           cmd[0] = AHACOM_EXTBIOS;        /* get extended BIOS information */
12785           aha_command(1, cmd, 2, extbios);
12786           if (extbios[1] != 0) {
12787                   /* Mailbox interface is locked, so unlock it. */
12788                   cmd[0] = AHACOM_MBOX_ENABLE;
12789                   cmd[1] = 0;             /* bit 0 = 0 (enable mailbox) */
12790                   cmd[2] = extbios[1];    /* lock code to unlock mailbox */
12791                   aha_command(3, cmd, 0, 0);
12792           }
12793     }
12794
12795     /* The maximum scatter/gather DMA list length depends on the board model. */
12796     sg_max = 16;
12797     if (aha_model == AHA1540) sg_max = 1;         /* 1540 has no s/g */
12798     if (aha_model >= AHA1540C) sg_max = 255;      /* 1540C has plenty */
12799
12800     /* Set up the DMA channel. */
12801     switch (getcdata[0]) {
12802     case 0x80:            /* channel 7 */
12803           out_byte(0xD6, 0xC3);
12804           out_byte(0xD4, 0x03);
12805           break;
12806     case 0x40:            /* channel 6 */
12807           out_byte(0xD6, 0xC2);
12808           out_byte(0xD4, 0x02);
12809           break;
12810     case 0x20:            /* channel 5 */
12811           out_byte(0xD6, 0xC1);
12812           out_byte(0xD4, 0x01);
12813           break;
12814     case 0x01:            /* channel 0 */
12815           out_byte(0x0B, 0x0C);
12816           out_byte(0x0A, 0x00);
12817           break;
12818     default:
12819           printf("aha0: AHA154x: strange DMA channel\n");
12820           return(0);
12821     }
12822
12823     /* Get the configured IRQ. */
12824     switch (getcdata[1]) {
12825     case 0x40:    irq = 15;       break;
12826     case 0x20:    irq = 14;       break;
12827     case 0x08:    irq = 12;       break;
12828     case 0x04:    irq = 11;       break;
12829     case 0x02:    irq = 10;       break;
```

```
12830       case 0x01:    irq =  9;       break;
12831       default:
12832             printf("aha0: strange IRQ setting\n");
12833             return(0);
12834       }
12835
12836       /* Enable interrupts on the given irq. */
12837       put_irq_handler(irq, s_handler);
12838       aha_irq = irq;
12839       enable_irq(irq);
12840
12841       /* Initialize request related data: Command Control Block, mailboxes.
12842        * (We want to have the mailboxes initialized early, because the 1540C
12843        * wants to know it now.)
12844        */
12845
12846       /* Init ccb. */
12847       rq->ccb.senselen = CCB_SENSEREQ;      /* always want sense info */
12848       h2b24(rq->ccb.linkptr, 0L);           /* never link commands */
12849       rq->ccb.linkid = 0;
12850       rq->ccb.reserved[0] = 0;
12851       rq->ccb.reserved[1] = 0;
12852
12853       /* Scatter/gather maximum. */
12854       rq->dmalimit = rq->dmalist + (sg_max < NR_IOREQS ? sg_max : NR_IOREQS);
12855
12856       /* Outgoing mailbox. */
12857       mailbox[0].status = AHA_MBOXFREE;
12858       h2b24(mailbox[0].ccbptr, vir2phys(&rq->ccb));
12859
12860       /* Incoming mailbox. */
12861       mailbox[1].status = AHA_MBOXFREE;
12862       /* mailbox[1].ccbptr filled by adapter after command execution. */
12863
12864       /* Tell controller where the mailboxes are and how many. */
12865       cmd[0] = AHACOM_INITBOX;
12866       cmd[1] = 1;
12867       h2b24(cmd + 2, vir2phys(mailbox));
12868       aha_command(5, cmd, 0, 0);
12869
12870       /* !! maybe sanity check: check status reg for initialization success */
12871
12872       /* Set bus on, bus off and transfer speed. */
12873       cmd[0] = AHACOM_BUSON;
12874       cmd[1] = bus_on;
12875       aha_command(2, cmd, 0, 0);
12876
12877       cmd[0] = AHACOM_BUSOFF;
12878       cmd[1] = bus_off;
12879       aha_command(2, cmd, 0, 0);
12880
12881       cmd[0] = AHACOM_SPEED;
12882       cmd[1] = tr_speed;
12883       aha_command(2, cmd, 0, 0);
12884
12885       /* Set SCSI selection timeout. */
12886       cmd[0] = AHACOM_SETIMEOUT;
12887       cmd[1] = SCSI_TIMEOUT != 0;           /* timeouts on/off */
12888       cmd[2] = 0;                           /* reserved */
12889       cmd[3] = SCSI_TIMEOUT / 256;          /* MSB */
```

```
12890       cmd[4] = SCSI_TIMEOUT % 256;          /* LSB */
12891       aha_command(5, cmd, 0, 0);
12892
12893       return(1);
12894     }


12897     /*==========================================================================*
12898      *                              s_handler                                   *
12899      *==========================================================================*/
12900     PRIVATE int s_handler(irq)
12901     int irq;
12902     {
12903     /* Host adapter interrupt, send message to SCSI task and reenable interrupts. */
12904
12905       if (in_byte(AHA_INTRREG) & AHA_HACC) {
12906             /* Simple commands are polled. */
12907             return 0;
12908       } else {
12909             out_byte(AHA_CNTLREG, AHA_IRST);        /* clear interrupt */
12910             interrupt(SCSI);
12911             return 1;
12912       }
12913     }


12916     /*==========================================================================*
12917      *                              h2b16                                       *
12918      *==========================================================================*/
12919     PRIVATE void h2b16(b, h)
12920     big16 b;
12921     U16_t h;
12922     {
12923     /* Host byte order to Big Endian conversion. */
12924       b[0] = h >> 8;
12925       b[1] = h >> 0;
12926     }


12929     /*==========================================================================*
12930      *                              h2b24                                       *
12931      *==========================================================================*/
12932     PRIVATE void h2b24(b, h)
12933     big24 b;
12934     u32_t h;
12935     {
12936       b[0] = h >> 16;
12937       b[1] = h >>  8;
12938       b[2] = h >>  0;
12939     }


12942     /*==========================================================================*
12943      *                              h2b32                                       *
12944      *==========================================================================*/
12945     PRIVATE void h2b32(b, h)
12946     big32 b;
12947     u32_t h;
12948     {
12949       b[0] = h >> 24;
```

```
12950     b[1] = h >> 16;
12951     b[2] = h >>  8;
12952     b[3] = h >>  0;
12953   }


12956   /*==========================================================================*
12957    *                              b2h16                                       *
12958    *==========================================================================*/
12959   PRIVATE u16_t b2h16(b)
12960   big16 b;
12961   {
12962     return  ((u16_t) b[0] << 8)
12963           | ((u16_t) b[1] << 0);
12964   }


12967   /*==========================================================================*
12968    *                              b2h24                                       *
12969    *==========================================================================*/
12970   PRIVATE u32_t b2h24(b)
12971   big24 b;
12972   {
12973     return  ((u32_t) b[0] << 16)
12974           | ((u32_t) b[1] <<  8)
12975           | ((u32_t) b[2] <<  0);
12976   }


12979   /*==========================================================================*
12980    *                              b2h32                                       *
12981    *==========================================================================*/
12982   PRIVATE u32_t b2h32(b)
12983   big32 b;
12984   {
12985     return  ((u32_t) b[0] << 24)
12986           | ((u32_t) b[1] << 16)
12987           | ((u32_t) b[2] <<  8)
12988           | ((u32_t) b[3] <<  0);
12989   }


12992   #if AHA_DEBUG & 2
12993   /*==========================================================================*
12994    *                              errordump                                   *
12995    *==========================================================================*/
12996   PRIVATE void errordump()
12997   {
12998     int i;
12999
13000     printf("aha ccb dump:");
13001     for (i = 0; i < sizeof(rq->ccb); i++) {
13002           if (i % 26 == 0) printf("\n");
13003           printf(" %02x", ((byte *) &rq->ccb)[i]);
13004     }
13005     printf("\n");
13006   }
13007   #endif /* AHA_DEBUG & 2 */
13008
13009
```

```
13010 #if AHA_DEBUG & 4
13011 /*===========================================================================*
13012  *                              show_req                                     *
13013  *===========================================================================*/
13014 PRIVATE void show_req()
13015 {
13016   struct iorequest_s **iopp;
13017   dma_t *dmap;
13018   unsigned count, nbytes, len;
13019
13020   iopp = rq->iov;
13021   dmap = rq->dmalist;
13022   count = rq->count;
13023   nbytes = 0;
13024
13025   printf("%lu:%u", rq->pos, count);
13026
13027   while (count > 0) {
13028         if (iopp == rq->iov || *iopp != iopp[-1])
13029                 nbytes = (*iopp)->io_nbytes;
13030
13031         printf(" (%u,%lx,%u)", nbytes, b2h24(dmap->dataptr),
13032                                        len = b2h24(dmap->datalen));
13033         dmap++;
13034         iopp++;
13035         count -= len;
13036         nbytes -= len;
13037   }
13038   if (nbytes > 0) printf(" ...(%u)", nbytes);
13039   printf("\n");
13040 }
13041 #endif /* AHA_DEBUG & 4 */
13042
13043
13044 #if AHA_DEBUG & 8
13045 /*===========================================================================*
13046  *                              dump_scsi_cmd                                *
13047  *===========================================================================*/
13048 PRIVATE void dump_scsi_cmd()
13049 {
13050   int i;
13051
13052   printf("scsi cmd:");
13053   for (i = 0; i < rq->ccb.cmdlen; i++) printf(" %02x", rq->ccb.cmd[i]);
13054   printf("\n");
13055 }
13056 #endif /* AHA_DEBUG & 8 */
13057
13058
13059 /*===========================================================================*
13060  *                              s_geometry                                   *
13061  *===========================================================================*/
13062 PRIVATE void s_geometry(entry)
13063 struct partition *entry;
13064 {
13065 /* The geometry of a SCSI drive is a complete fake, the Adaptec onboard BIOS
13066  * makes the drive look like a regular drive on the outside.  A DOS program
13067  * takes a logical block address, computes cylinder, head and sector like the
13068  * BIOS int 0x13 call expects, and the Adaptec turns this back into a block
13069  * address again.  The only reason we care is because some idiot put cylinder,
```

```
13070     * head and sector numbers in the partition table, so fdisk needs to know the
13071     * geometry.
13072     */
13073      unsigned long size = s_sp->part[0].dv_size;
13074      unsigned heads, sectors;
13075
13076      if (size < 1024L * 64 * 32 * 512) {
13077            /* Small drive. */
13078            heads = 64;
13079            sectors = 32;
13080      } else {
13081            /* Assume that this BIOS is configured for large drives. */
13082            heads = 255;
13083            sectors = 63;
13084      }
13085      entry->cylinders = (size >> SECTOR_SHIFT) / (heads * sectors);
13086      entry->heads = heads;
13087      entry->sectors = sectors;
13088    }
13089    #endif /* !ENABLE_ADAPTEC_SCSI */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/at_wini.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
13100    /* This file contains the device dependent part of a driver for the IBM-AT
13101     * winchester controller.
13102     * It was written by Adri Koppes.
13103     *
13104     * The file contains one entry point:
13105     *
13106     *   at_winchester_task:        main entry when system is brought up
13107     *
13108     *
13109     * Changes:
13110     *      13 Apr 1992 by Kees J. Bot: device dependent/independent split.
13111     */
13112
13113    #include "kernel.h"
13114    #include "driver.h"
13115    #include "drvlib.h"
13116
13117    #if ENABLE_AT_WINI
13118
13119    /* I/O Ports used by winchester disk controllers. */
13120
13121    /* Read and write registers */
13122    #define REG_BASE0       0x1F0   /* base register of controller 0 */
13123    #define REG_BASE1       0x170   /* base register of controller 1 */
13124    #define REG_DATA            0   /* data register (offset from the base reg.) */
13125    #define REG_PRECOMP         1   /* start of write precompensation */
13126    #define REG_COUNT           2   /* sectors to transfer */
13127    #define REG_SECTOR          3   /* sector number */
13128    #define REG_CYL_LO          4   /* low byte of cylinder number */
13129    #define REG_CYL_HI          5   /* high byte of cylinder number */
```

```
13130   #define REG_LDH             6   /* lba, drive and head */
13131   #define   LDH_DEFAULT           0xA0    /* ECC enable, 512 bytes per sector */
13132   #define   LDH_LBA               0x40    /* Use LBA addressing */
13133   #define   ldh_init(drive)       (LDH_DEFAULT | ((drive) << 4))
13134
13135   /* Read only registers */
13136   #define REG_STATUS          7   /* status */
13137   #define   STATUS_BSY            0x80    /* controller busy */
13138   #define   STATUS_RDY            0x40    /* drive ready */
13139   #define   STATUS_WF             0x20    /* write fault */
13140   #define   STATUS_SC             0x10    /* seek complete (obsolete) */
13141   #define   STATUS_DRQ            0x08    /* data transfer request */
13142   #define   STATUS_CRD            0x04    /* corrected data */
13143   #define   STATUS_IDX            0x02    /* index pulse */
13144   #define   STATUS_ERR            0x01    /* error */
13145   #define REG_ERROR           1   /* error code */
13146   #define   ERROR_BB              0x80    /* bad block */
13147   #define   ERROR_ECC             0x40    /* bad ecc bytes */
13148   #define   ERROR_ID              0x10    /* id not found */
13149   #define   ERROR_AC              0x04    /* aborted command */
13150   #define   ERROR_TK              0x02    /* track zero error */
13151   #define   ERROR_DM              0x01    /* no data address mark */
13152
13153   /* Write only registers */
13154   #define REG_COMMAND         7   /* command */
13155   #define   CMD_IDLE              0x00    /* for w_command: drive idle */
13156   #define   CMD_RECALIBRATE       0x10    /* recalibrate drive */
13157   #define   CMD_READ              0x20    /* read data */
13158   #define   CMD_WRITE             0x30    /* write data */
13159   #define   CMD_READVERIFY        0x40    /* read verify */
13160   #define   CMD_FORMAT            0x50    /* format track */
13161   #define   CMD_SEEK              0x70    /* seek cylinder */
13162   #define   CMD_DIAG              0x90    /* execute device diagnostics */
13163   #define   CMD_SPECIFY           0x91    /* specify parameters */
13164   #define   ATA_IDENTIFY          0xEC    /* identify drive */
13165   #define REG_CTL         0x206   /* control register */
13166   #define   CTL_NORETRY           0x80    /* disable access retry */
13167   #define   CTL_NOECC             0x40    /* disable ecc retry */
13168   #define   CTL_EIGHTHEADS        0x08    /* more than eight heads */
13169   #define   CTL_RESET             0x04    /* reset controller */
13170   #define   CTL_INTDISABLE        0x02    /* disable interrupts */
13171
13172   /* Interrupt request lines. */
13173   #define AT_IRQ0         14      /* interrupt number for controller 0 */
13174   #define AT_IRQ1         15      /* interrupt number for controller 1 */
13175
13176   /* Common command block */
13177   struct command {
13178     u8_t  precomp;        /* REG_PRECOMP, etc. */
13179     u8_t  count;
13180     u8_t  sector;
13181     u8_t  cyl_lo;
13182     u8_t  cyl_hi;
13183     u8_t  ldh;
13184     u8_t  command;
13185   };
13186
13187
13188   /* Error codes */
13189   #define ERR              (-1)   /* general error */
```

```
13190   #define ERR_BAD_SECTOR   (-2)   /* block marked bad detected */
13191
13192   /* Some controllers don't interrupt, the clock will wake us up. */
13193   #define WAKEUP          (32*HZ) /* drive may be out for 31 seconds max */
13194
13195   /* Miscellaneous. */
13196   #define MAX_DRIVES         4    /* this driver supports 4 drives (hd0 - hd19) */
13197   #if _WORD_SIZE > 2
13198   #define MAX_SECS         256    /* controller can transfer this many sectors */
13199   #else
13200   #define MAX_SECS         127    /* but not to a 16 bit process */
13201   #endif
13202   #define MAX_ERRORS         4    /* how often to try rd/wt before quitting */
13203   #define NR_DEVICES      (MAX_DRIVES * DEV_PER_DRIVE)
13204   #define SUB_PER_DRIVE   (NR_PARTITIONS * NR_PARTITIONS)
13205   #define NR_SUBDEVS      (MAX_DRIVES * SUB_PER_DRIVE)
13206   #define TIMEOUT        32000    /* controller timeout in ms */
13207   #define RECOVERYTIME     500    /* controller recovery time in ms */
13208   #define INITIALIZED     0x01    /* drive is initialized */
13209   #define DEAF            0x02    /* controller must be reset */
13210   #define SMART           0x04    /* drive supports ATA commands */
13211
13212
13213   /* Variables. */
13214   PRIVATE struct wini {           /* main drive struct, one entry per drive */
13215     unsigned state;               /* drive state: deaf, initialized, dead */
13216     unsigned base;                /* base register of the register file */
13217     unsigned irq;                 /* interrupt request line */
13218     unsigned lcylinders;          /* logical number of cylinders (BIOS) */
13219     unsigned lheads;              /* logical number of heads */
13220     unsigned lsectors;            /* logical number of sectors per track */
13221     unsigned pcylinders;          /* physical number of cylinders (translated) */
13222     unsigned pheads;              /* physical number of heads */
13223     unsigned psectors;            /* physical number of sectors per track */
13224     unsigned ldhpref;             /* top four bytes of the LDH (head) register */
13225     unsigned precomp;             /* write precompensation cylinder / 4 */
13226     unsigned max_count;           /* max request for this drive */
13227     unsigned open_ct;             /* in-use count */
13228     struct device part[DEV_PER_DRIVE];    /* primary partitions: hd[0-4] */
13229     struct device subpart[SUB_PER_DRIVE]; /* subpartitions: hd[1-4][a-d] */
13230   } wini[MAX_DRIVES], *w_wn;
13231
13232   PRIVATE struct trans {
13233     struct iorequest_s *iop;      /* belongs to this I/O request */
13234     unsigned long block;          /* first sector to transfer */
13235     unsigned count;               /* byte count */
13236     phys_bytes phys;              /* user physical address */
13237   } wtrans[NR_IOREQS];
13238
13239   PRIVATE struct trans *w_tp;             /* to add transfer requests */
13240   PRIVATE unsigned w_count;               /* number of bytes to transfer */
13241   PRIVATE unsigned long w_nextblock;      /* next block on disk to transfer */
13242   PRIVATE int w_opcode;                   /* DEV_READ or DEV_WRITE */
13243   PRIVATE int w_command;                  /* current command in execution */
13244   PRIVATE int w_status;                   /* status after interrupt */
13245   PRIVATE int w_drive;                    /* selected drive */
13246   PRIVATE struct device *w_dv;            /* device's base and size */
13247
13248   FORWARD _PROTOTYPE( void init_params, (void) );
13249   FORWARD _PROTOTYPE( int w_do_open, (struct driver *dp, message *m_ptr) );
```

```
13250   FORWARD _PROTOTYPE( struct device *w_prepare, (int device) );
13251   FORWARD _PROTOTYPE( int w_identify, (void) );
13252   FORWARD _PROTOTYPE( char *w_name, (void) );
13253   FORWARD _PROTOTYPE( int w_specify, (void) );
13254   FORWARD _PROTOTYPE( int w_schedule, (int proc_nr, struct iorequest_s *iop) );
13255   FORWARD _PROTOTYPE( int w_finish, (void) );
13256   FORWARD _PROTOTYPE( int com_out, (struct command *cmd) );
13257   FORWARD _PROTOTYPE( void w_need_reset, (void) );
13258   FORWARD _PROTOTYPE( int w_do_close, (struct driver *dp, message *m_ptr) );
13259   FORWARD _PROTOTYPE( int com_simple, (struct command *cmd) );
13260   FORWARD _PROTOTYPE( void w_timeout, (void) );
13261   FORWARD _PROTOTYPE( int w_reset, (void) );
13262   FORWARD _PROTOTYPE( int w_intr_wait, (void) );
13263   FORWARD _PROTOTYPE( int w_waitfor, (int mask, int value) );
13264   FORWARD _PROTOTYPE( int w_handler, (int irq) );
13265   FORWARD _PROTOTYPE( void w_geometry, (struct partition *entry) );
13266
13267   /* w_waitfor loop unrolled once for speed. */
13268   #define waitfor(mask, value)    \
13269           ((in_byte(w_wn->base + REG_STATUS) & mask) == value \
13270                   || w_waitfor(mask, value))
13271
13272
13273   /* Entry points to this driver. */
13274   PRIVATE struct driver w_dtab = {
13275     w_name,               /* current device's name */
13276     w_do_open,            /* open or mount request, initialize device */
13277     w_do_close,           /* release device */
13278     do_diocntl,           /* get or set a partition's geometry */
13279     w_prepare,            /* prepare for I/O on a given minor device */
13280     w_schedule,           /* precompute cylinder, head, sector, etc. */
13281     w_finish,             /* do the I/O */
13282     nop_cleanup,          /* nothing to clean up */
13283     w_geometry,           /* tell the geometry of the disk */
13284   };
13285
13286   #if ENABLE_ATAPI
13287   #include "atapi.c"      /* extra code for ATAPI CD-ROM */
13288   #endif
13289
13290
13291   /*===========================================================================*
13292    *                              at_winchester_task                           *
13293    *===========================================================================*/
13294   PUBLIC void at_winchester_task()
13295   {
13296   /* Set special disk parameters then call the generic main loop. */
13297
13298     init_params();
13299
13300     driver_task(&w_dtab);
13301   }


13304   /*===========================================================================*
13305    *                              init_params                                  *
13306    *===========================================================================*/
13307   PRIVATE void init_params()
13308   {
13309   /* This routine is called at startup to initialize the drive parameters. */
```

```
13310
13311     u16_t parv[2];
13312     unsigned int vector;
13313     int drive, nr_drives, i;
13314     struct wini *wn;
13315     u8_t params[16];
13316     phys_bytes param_phys = vir2phys(params);
13317
13318     /* Get the number of drives from the BIOS data area */
13319     phys_copy(0x475L, param_phys, 1L);
13320     if ((nr_drives = params[0]) > 2) nr_drives = 2;
13321
13322     for (drive = 0, wn = wini; drive < MAX_DRIVES; drive++, wn++) {
13323           if (drive < nr_drives) {
13324                   /* Copy the BIOS parameter vector */
13325                   vector = drive == 0 ? WINI_0_PARM_VEC : WINI_1_PARM_VEC;
13326                   phys_copy(vector * 4L, vir2phys(parv), 4L);
13327
13328                   /* Calculate the address of the parameters and copy them */
13329                   phys_copy(hclick_to_physb(parv[1]) + parv[0], param_phys, 16L);
13330
13331                   /* Copy the parameters to the structures of the drive */
13332                   wn->lcylinders = bp_cylinders(params);
13333                   wn->lheads = bp_heads(params);
13334                   wn->lsectors = bp_sectors(params);
13335                   wn->precomp = bp_precomp(params) >> 2;
13336           }
13337           wn->ldhpref = ldh_init(drive);
13338           wn->max_count = MAX_SECS << SECTOR_SHIFT;
13339           if (drive < 2) {
13340                   /* Controller 0. */
13341                   wn->base = REG_BASE0;
13342                   wn->irq = AT_IRQ0;
13343           } else {
13344                   /* Controller 1. */
13345                   wn->base = REG_BASE1;
13346                   wn->irq = AT_IRQ1;
13347           }
13348     }
13349   }


13352   /*============================================================================*
13353    *                              w_do_open                                     *
13354    *============================================================================*/
13355   PRIVATE int w_do_open(dp, m_ptr)
13356   struct driver *dp;
13357   message *m_ptr;
13358   {
13359   /* Device open: Initialize the controller and read the partition table. */
13360
13361     int r;
13362     struct wini *wn;
13363     struct command cmd;
13364
13365     if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
13366     wn = w_wn;
13367
13368     if (wn->state == 0) {
13369           /* Try to identify the device. */
```

```
13370           if (w_identify() != OK) {
13371                   printf("%s: probe failed\n", w_name());
13372                   if (wn->state & DEAF) w_reset();
13373                   wn->state = 0;
13374                   return(ENXIO);
13375           }
13376     }
13377     if (wn->open_ct++ == 0) {
13378           /* Partition the disk. */
13379           partition(&w_dtab, w_drive * DEV_PER_DRIVE, P_PRIMARY);
13380     }
13381     return(OK);
13382   }


13385   /*============================================================================*
13386    *                              w_prepare                                     *
13387    *============================================================================*/
13388   PRIVATE struct device *w_prepare(device)
13389   int device;
13390   {
13391   /* Prepare for I/O on a device. */
13392
13393     /* Nothing to transfer as yet. */
13394     w_count = 0;
13395
13396     if (device < NR_DEVICES) {                    /* hd0, hd1, ... */
13397           w_drive = device / DEV_PER_DRIVE;       /* save drive number */
13398           w_wn = &wini[w_drive];
13399           w_dv = &w_wn->part[device % DEV_PER_DRIVE];
13400     } else
13401     if ((unsigned) (device -= MINOR_hd1a) < NR_SUBDEVS) { /* hd1a, hd1b, ... */
13402           w_drive = device / SUB_PER_DRIVE;
13403           w_wn = &wini[w_drive];
13404           w_dv = &w_wn->subpart[device % SUB_PER_DRIVE];
13405     } else {
13406           return(NIL_DEV);
13407     }
13408     return(w_dv);
13409   }


13412   /*============================================================================*
13413    *                              w_identify                                    *
13414    *============================================================================*/
13415   PRIVATE int w_identify()
13416   {
13417   /* Find out if a device exists, if it is an old AT disk, or a newer ATA
13418    * drive, a removable media device, etc.
13419    */
13420
13421     struct wini *wn = w_wn;
13422     struct command cmd;
13423     char id_string[40];
13424     int i, r;
13425     unsigned long size;
13426   #define id_byte(n)      (&tmp_buf[2 * (n)])
13427   #define id_word(n)      (((u16_t) id_byte(n)[0] <<  0) \
13428                           |((u16_t) id_byte(n)[1] <<  8))
13429   #define id_longword(n)  (((u32_t) id_byte(n)[0] <<  0) \
```

```
13430                           |((u32_t) id_byte(n)[1] <<  8) \
13431                           |((u32_t) id_byte(n)[2] << 16) \
13432                           |((u32_t) id_byte(n)[3] << 24))
13433
13434       /* Check if the one of the registers exists. */
13435       r = in_byte(wn->base + REG_CYL_LO);
13436       out_byte(wn->base + REG_CYL_LO, ~r);
13437       if (in_byte(wn->base + REG_CYL_LO) == r) return(ERR);
13438
13439       /* Looks OK; register IRQ and try an ATA identify command. */
13440       put_irq_handler(wn->irq, w_handler);
13441       enable_irq(wn->irq);
13442
13443       cmd.ldh     = wn->ldhpref;
13444       cmd.command = ATA_IDENTIFY;
13445       if (com_simple(&cmd) == OK) {
13446             /* This is an ATA device. */
13447             wn->state |= SMART;
13448
13449             /* Device information. */
13450             port_read(wn->base + REG_DATA, tmp_phys, SECTOR_SIZE);
13451
13452             /* Why are the strings byte swapped??? */
13453             for (i = 0; i < 40; i++) id_string[i] = id_byte(27)[i^1];
13454
13455             /* Preferred CHS translation mode. */
13456             wn->pcylinders = id_word(1);
13457             wn->pheads = id_word(3);
13458             wn->psectors = id_word(6);
13459             size = (u32_t) wn->pcylinders * wn->pheads * wn->psectors;
13460
13461             if ((id_byte(49)[1] & 0x02) && size > 512L*1024*2) {
13462                     /* Drive is LBA capable and is big enough to trust it to
13463                      * not make a mess of it.
13464                      */
13465                     wn->ldhpref |= LDH_LBA;
13466                     size = id_longword(60);
13467             }
13468
13469             if (wn->lcylinders == 0) {
13470                     /* No BIOS parameters?  Then make some up. */
13471                     wn->lcylinders = wn->pcylinders;
13472                     wn->lheads = wn->pheads;
13473                     wn->lsectors = wn->psectors;
13474                     while (wn->lcylinders > 1024) {
13475                             wn->lheads *= 2;
13476                             wn->lcylinders /= 2;
13477                     }
13478             }
13479       } else {
13480             /* Not an ATA device; no translations, no special features.  Don't
13481              * touch it unless the BIOS knows about it.
13482              */
13483             if (wn->lcylinders == 0) return(ERR);   /* no BIOS parameters */
13484             wn->pcylinders = wn->lcylinders;
13485             wn->pheads = wn->lheads;
13486             wn->psectors = wn->lsectors;
13487             size = (u32_t) wn->pcylinders * wn->pheads * wn->psectors;
13488       }
13489       /* The fun ends at 4 GB. */
```

```
13490     if (size > ((u32_t) -1) / SECTOR_SIZE) size = ((u32_t) -1) / SECTOR_SIZE;
13491
13492     /* Base and size of the whole drive */
13493     wn->part[0].dv_base = 0;
13494     wn->part[0].dv_size = size << SECTOR_SHIFT;
13495
13496     if (w_specify() != OK && w_specify() != OK) return(ERR);
13497
13498     printf("%s: ", w_name());
13499     if (wn->state & SMART) {
13500           printf("%.40s\n", id_string);
13501     } else {
13502           printf("%ux%ux%u\n", wn->pcylinders, wn->pheads, wn->psectors);
13503     }
13504     return(OK);
13505   }


13508   /*============================================================================*
13509    *                              w_name                                        *
13510    *============================================================================*/
13511   PRIVATE char *w_name()
13512   {
13513   /* Return a name for the current device. */
13514     static char name[] = "at-hd15";
13515     unsigned device = w_drive * DEV_PER_DRIVE;
13516
13517     if (device < 10) {
13518           name[5] = '0' + device;
13519           name[6] = 0;
13520     } else {
13521           name[5] = '0' + device / 10;
13522           name[6] = '0' + device % 10;
13523     }
13524     return name;
13525   }


13528   /*============================================================================*
13529    *                              w_specify                                     *
13530    *============================================================================*/
13531   PRIVATE int w_specify()
13532   {
13533   /* Routine to initialize the drive after boot or when a reset is needed. */
13534
13535     struct wini *wn = w_wn;
13536     struct command cmd;
13537
13538     if ((wn->state & DEAF) && w_reset() != OK) return(ERR);
13539
13540     /* Specify parameters: precompensation, number of heads and sectors. */
13541     cmd.precomp = wn->precomp;
13542     cmd.count   = wn->psectors;
13543     cmd.ldh     = w_wn->ldhpref | (wn->pheads - 1);
13544     cmd.command = CMD_SPECIFY;            /* Specify some parameters */
13545
13546     if (com_simple(&cmd) != OK) return(ERR);
13547
13548     if (!(wn->state & SMART)) {
13549           /* Calibrate an old disk. */
```

```
13550           cmd.sector  = 0;
13551           cmd.cyl_lo  = 0;
13552           cmd.cyl_hi  = 0;
13553           cmd.ldh     = w_wn->ldhpref;
13554           cmd.command = CMD_RECALIBRATE;
13555
13556           if (com_simple(&cmd) != OK) return(ERR);
13557     }
13558
13559     wn->state |= INITIALIZED;
13560     return(OK);
13561   }


13564   /*============================================================================*
13565    *                              w_schedule                                    *
13566    *============================================================================*/
13567   PRIVATE int w_schedule(proc_nr, iop)
13568   int proc_nr;                    /* process doing the request */
13569   struct iorequest_s *iop;        /* pointer to read or write request */
13570   {
13571   /* Gather I/O requests on consecutive blocks so they may be read/written
13572    * in one controller command.  (There is enough time to compute the next
13573    * consecutive request while an unwanted block passes by.)
13574    */
13575     struct wini *wn = w_wn;
13576     int r, opcode;
13577     unsigned long pos;
13578     unsigned nbytes, count;
13579     unsigned long block;
13580     phys_bytes user_phys;
13581
13582     /* This many bytes to read/write */
13583     nbytes = iop->io_nbytes;
13584     if ((nbytes & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);
13585
13586     /* From/to this position on the device */
13587     pos = iop->io_position;
13588     if ((pos & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);
13589
13590     /* To/from this user address */
13591     user_phys = numap(proc_nr, (vir_bytes) iop->io_buf, nbytes);
13592     if (user_phys == 0) return(iop->io_nbytes = EINVAL);
13593
13594     /* Read or write? */
13595     opcode = iop->io_request & ~OPTIONAL_IO;
13596
13597     /* Which block on disk and how close to EOF? */
13598     if (pos >= w_dv->dv_size) return(OK);         /* At EOF */
13599     if (pos + nbytes > w_dv->dv_size) nbytes = w_dv->dv_size - pos;
13600     block = (w_dv->dv_base + pos) >> SECTOR_SHIFT;
13601
13602     if (w_count > 0 && block != w_nextblock) {
13603           /* This new request can't be chained to the job being built */
13604           if ((r = w_finish()) != OK) return(r);
13605     }
13606
13607     /* The next consecutive block */
13608     w_nextblock = block + (nbytes >> SECTOR_SHIFT);
13609
```

```
13610     /* While there are "unscheduled" bytes in the request: */
13611     do {
13612           count = nbytes;
13613
13614           if (w_count == wn->max_count) {
13615                   /* The drive can't do more then max_count at once */
13616                   if ((r = w_finish()) != OK) return(r);
13617           }
13618
13619           if (w_count + count > wn->max_count)
13620                   count = wn->max_count - w_count;
13621
13622           if (w_count == 0) {
13623                   /* The first request in a row, initialize. */
13624                   w_opcode = opcode;
13625                   w_tp = wtrans;
13626           }
13627
13628           /* Store I/O parameters */
13629           w_tp->iop = iop;
13630           w_tp->block = block;
13631           w_tp->count = count;
13632           w_tp->phys = user_phys;
13633
13634           /* Update counters */
13635           w_tp++;
13636           w_count += count;
13637           block += count >> SECTOR_SHIFT;
13638           user_phys += count;
13639           nbytes -= count;
13640     } while (nbytes > 0);
13641
13642     return(OK);
13643   }


13646   /*============================================================================*
13647    *                              w_finish                                      *
13648    *============================================================================*/
13649   PRIVATE int w_finish()
13650   {
13651   /* Carry out the I/O requests gathered in wtrans[]. */
13652
13653     struct trans *tp = wtrans;
13654     struct wini *wn = w_wn;
13655     int r, errors;
13656     struct command cmd;
13657     unsigned cylinder, head, sector, secspcyl;
13658
13659     if (w_count == 0) return(OK); /* Spurious finish. */
13660
13661     r = ERR;      /* Trigger the first com_out */
13662     errors = 0;
13663
13664     do {
13665           if (r != OK) {
13666                   /* The controller must be (re)programmed. */
13667
13668                   /* First check to see if a reinitialization is needed. */
13669                   if (!(wn->state & INITIALIZED) && w_specify() != OK)
```

```
13670                           return(tp->iop->io_nbytes = EIO);
13671
13672                   /* Tell the controller to transfer w_count bytes */
13673                   cmd.precomp = wn->precomp;
13674                   cmd.count   = (w_count >> SECTOR_SHIFT) & BYTE;
13675                   if (wn->ldhpref & LDH_LBA) {
13676                           cmd.sector  = (tp->block >>  0) & 0xFF;
13677                           cmd.cyl_lo  = (tp->block >>  8) & 0xFF;
13678                           cmd.cyl_hi  = (tp->block >> 16) & 0xFF;
13679                           cmd.ldh     = wn->ldhpref | ((tp->block >> 24) & 0xF);
13680                   } else {
13681                           secspcyl = wn->pheads * wn->psectors;
13682                           cylinder = tp->block / secspcyl;
13683                           head = (tp->block % secspcyl) / wn->psectors;
13684                           sector = tp->block % wn->psectors;
13685                           cmd.sector  = sector + 1;
13686                           cmd.cyl_lo  = cylinder & BYTE;
13687                           cmd.cyl_hi  = (cylinder >> 8) & BYTE;
13688                           cmd.ldh     = wn->ldhpref | head;
13689                   }
13690                   cmd.command = w_opcode == DEV_WRITE ? CMD_WRITE : CMD_READ;
13691
13692                   if ((r = com_out(&cmd)) != OK) {
13693                           if (++errors == MAX_ERRORS) {
13694                                   w_command = CMD_IDLE;
13695                                   return(tp->iop->io_nbytes = EIO);
13696                           }
13697                           continue;       /* Retry */
13698                   }
13699           }
13700
13701           /* For each sector, wait for an interrupt and fetch the data (read),
13702            * or supply data to the controller and wait for an interrupt (write).
13703            */
13704
13705           if (w_opcode == DEV_READ) {
13706                   if ((r = w_intr_wait()) == OK) {
13707                           /* Copy data from the device's buffer to user space. */
13708
13709                           port_read(wn->base + REG_DATA, tp->phys, SECTOR_SIZE);
13710
13711                           tp->phys += SECTOR_SIZE;
13712                           tp->iop->io_nbytes -= SECTOR_SIZE;
13713                           w_count -= SECTOR_SIZE;
13714                           if ((tp->count -= SECTOR_SIZE) == 0) tp++;
13715                   } else {
13716                           /* Any faulty data? */
13717                           if (w_status & STATUS_DRQ) {
13718                                   port_read(wn->base + REG_DATA, tmp_phys,
13719                                                                   SECTOR_SIZE);
13720                           }
13721                   }
13722           } else {
13723                   /* Wait for data requested. */
13724                   if (!waitfor(STATUS_DRQ, STATUS_DRQ)) {
13725                           r = ERR;
13726                   } else {
13727                           /* Fill the buffer of the drive. */
13728
13729                           port_write(wn->base + REG_DATA, tp->phys, SECTOR_SIZE);
```

```
13730                           r = w_intr_wait();
13731                   }
13732
13733                   if (r == OK) {
13734                           /* Book the bytes successfully written. */
13735
13736                           tp->phys += SECTOR_SIZE;
13737                           tp->iop->io_nbytes -= SECTOR_SIZE;
13738                           w_count -= SECTOR_SIZE;
13739                           if ((tp->count -= SECTOR_SIZE) == 0) tp++;
13740                   }
13741           }
13742
13743           if (r != OK) {
13744                   /* Don't retry if sector marked bad or too many errors */
13745                   if (r == ERR_BAD_SECTOR || ++errors == MAX_ERRORS) {
13746                           w_command = CMD_IDLE;
13747                           return(tp->iop->io_nbytes = EIO);
13748                   }
13749
13750                   /* Reset if halfway, but bail out if optional I/O. */
13751                   if (errors == MAX_ERRORS / 2) {
13752                           w_need_reset();
13753                           if (tp->iop->io_request & OPTIONAL_IO) {
13754                                   w_command = CMD_IDLE;
13755                                   return(tp->iop->io_nbytes = EIO);
13756                           }
13757                   }
13758                   continue;       /* Retry */
13759           }
13760           errors = 0;
13761     } while (w_count > 0);
13762
13763     w_command = CMD_IDLE;
13764     return(OK);
13765   }


13768   /*============================================================================*
13769    *                              com_out                                       *
13770    *============================================================================*/
13771   PRIVATE int com_out(cmd)
13772   struct command *cmd;            /* Command block */
13773   {
13774   /* Output the command block to the winchester controller and return status */
13775
13776     struct wini *wn = w_wn;
13777     unsigned base = wn->base;
13778
13779     if (!waitfor(STATUS_BSY, 0)) {
13780           printf("%s: controller not ready\n", w_name());
13781           return(ERR);
13782     }
13783
13784     /* Select drive. */
13785     out_byte(base + REG_LDH, cmd->ldh);
13786
13787     if (!waitfor(STATUS_BSY, 0)) {
13788           printf("%s: drive not ready\n", w_name());
13789           return(ERR);
```

```
13790       }
13791
13792       /* Schedule a wakeup call, some controllers are flaky. */
13793       clock_mess(WAKEUP, w_timeout);
13794
13795       out_byte(base + REG_CTL, wn->pheads >= 8 ? CTL_EIGHTHEADS : 0);
13796       out_byte(base + REG_PRECOMP, cmd->precomp);
13797       out_byte(base + REG_COUNT, cmd->count);
13798       out_byte(base + REG_SECTOR, cmd->sector);
13799       out_byte(base + REG_CYL_LO, cmd->cyl_lo);
13800       out_byte(base + REG_CYL_HI, cmd->cyl_hi);
13801       lock();
13802       out_byte(base + REG_COMMAND, cmd->command);
13803       w_command = cmd->command;
13804       w_status = STATUS_BSY;
13805       unlock();
13806       return(OK);
13807     }


13810     /*===========================================================================*
13811      *                              w_need_reset                                 *
13812      *===========================================================================*/
13813     PRIVATE void w_need_reset()
13814     {
13815     /* The controller needs to be reset. */
13816       struct wini *wn;
13817
13818       for (wn = wini; wn < &wini[MAX_DRIVES]; wn++) {
13819             wn->state |= DEAF;
13820             wn->state &= ~INITIALIZED;
13821       }
13822     }


13825     /*===========================================================================*
13826      *                              w_do_close                                   *
13827      *===========================================================================*/
13828     PRIVATE int w_do_close(dp, m_ptr)
13829     struct driver *dp;
13830     message *m_ptr;
13831     {
13832     /* Device close: Release a device. */
13833
13834       if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
13835       w_wn->open_ct--;
13836       return(OK);
13837     }


13840     /*===========================================================================*
13841      *                              com_simple                                   *
13842      *===========================================================================*/
13843     PRIVATE int com_simple(cmd)
13844     struct command *cmd;            /* Command block */
13845     {
13846     /* A simple controller command, only one interrupt and no data-out phase. */
13847       int r;
13848
13849       if ((r = com_out(cmd)) == OK) r = w_intr_wait();
```

```
13850       w_command = CMD_IDLE;
13851       return(r);
13852     }


13855     /*===========================================================================*
13856      *                              w_timeout                                    *
13857      *===========================================================================*/
13858     PRIVATE void w_timeout()
13859     {
13860       struct wini *wn = w_wn;
13861
13862       switch (w_command) {
13863       case CMD_IDLE:
13864             break;          /* fine */
13865       case CMD_READ:
13866       case CMD_WRITE:
13867             /* Impossible, but not on PC's:  The controller does not respond. */
13868
13869             /* Limiting multisector I/O seems to help. */
13870             if (wn->max_count > 8 * SECTOR_SIZE) {
13871                     wn->max_count = 8 * SECTOR_SIZE;
13872             } else {
13873                     wn->max_count = SECTOR_SIZE;
13874             }
13875             /*FALL THROUGH*/
13876       default:
13877             /* Some other command. */
13878             printf("%s: timeout on command %02x\n", w_name(), w_command);
13879             w_need_reset();
13880             w_status = 0;
13881             interrupt(WINCHESTER);
13882       }
13883     }


13886     /*===========================================================================*
13887      *                              w_reset                                      *
13888      *===========================================================================*/
13889     PRIVATE int w_reset()
13890     {
13891     /* Issue a reset to the controller.  This is done after any catastrophe,
13892      * like the controller refusing to respond.
13893      */
13894
13895       struct wini *wn;
13896       int err;
13897
13898       /* Wait for any internal drive recovery. */
13899       milli_delay(RECOVERYTIME);
13900
13901       /* Strobe reset bit */
13902       out_byte(w_wn->base + REG_CTL, CTL_RESET);
13903       milli_delay(1);
13904       out_byte(w_wn->base + REG_CTL, 0);
13905       milli_delay(1);
13906
13907       /* Wait for controller ready */
13908       if (!w_waitfor(STATUS_BSY | STATUS_RDY, STATUS_RDY)) {
13909             printf("%s: reset failed, drive busy\n", w_name());
```

```
13910           return(ERR);
13911     }
13912
13913     /* The error register should be checked now, but some drives mess it up. */
13914
13915     for (wn = wini; wn < &wini[MAX_DRIVES]; wn++) {
13916           if (wn->base == w_wn->base) wn->state &= ~DEAF;
13917     }
13918     return(OK);
13919   }


13922   /*============================================================================*
13923    *                              w_intr_wait                                   *
13924    *============================================================================*/
13925   PRIVATE int w_intr_wait()
13926   {
13927   /* Wait for a task completion interrupt and return results. */
13928
13929     message mess;
13930     int r;
13931
13932     /* Wait for an interrupt that sets w_status to "not busy". */
13933     while (w_status & STATUS_BSY) receive(HARDWARE, &mess);
13934
13935     /* Check status. */
13936     lock();
13937     if ((w_status & (STATUS_BSY | STATUS_RDY | STATUS_WF | STATUS_ERR))
13938                                                         == STATUS_RDY) {
13939           r = OK;
13940           w_status |= STATUS_BSY; /* assume still busy with I/O */
13941     } else
13942     if ((w_status & STATUS_ERR) && (in_byte(w_wn->base + REG_ERROR) & ERROR_BB)) {
13943           r = ERR_BAD_SECTOR;     /* sector marked bad, retries won't help */
13944     } else {
13945           r = ERR;                /* any other error */
13946     }
13947     unlock();
13948     return(r);
13949   }


13952   /*============================================================================*
13953    *                              w_waitfor                                     *
13954    *============================================================================*/
13955   PRIVATE int w_waitfor(mask, value)
13956   int mask;                       /* status mask */
13957   int value;                      /* required status */
13958   {
13959   /* Wait until controller is in the required state.  Return zero on timeout. */
13960
13961     struct milli_state ms;
13962
13963     milli_start(&ms);
13964     do {
13965          if ((in_byte(w_wn->base + REG_STATUS) & mask) == value) return 1;
13966     } while (milli_elapsed(&ms) < TIMEOUT);
13967
13968     w_need_reset();       /* Controller gone deaf. */
13969     return(0);
```

```
13970    }


13973    /*==========================================================================*
13974     *                              w_handler                                   *
13975     *==========================================================================*/
13976    PRIVATE int w_handler(irq)
13977    int irq;
13978    {
13979    /* Disk interrupt, send message to winchester task and reenable interrupts. */
13980
13981      w_status = in_byte(w_wn->base + REG_STATUS);  /* acknowledge interrupt */
13982      interrupt(WINCHESTER);
13983      return 1;
13984    }


13987    /*==========================================================================*
13988     *                              w_geometry                                  *
13989     *==========================================================================*/
13990    PRIVATE void w_geometry(entry)
13991    struct partition *entry;
13992    {
13993      entry->cylinders = w_wn->lcylinders;
13994      entry->heads = w_wn->lheads;
13995      entry->sectors = w_wn->lsectors;
13996    }
13997    #endif /* ENABLE_AT_WINI */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/bios_wini.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

14000    /* This file contains the "device dependent" part of a hard disk driver that
14001     * uses the ROM BIOS.  It makes a call and just waits for the transfer to
14002     * happen.  It is not interrupt driven and thus will (*) have poor performance.
14003     * The advantage is that it should work on virtually any PC, XT, 386, PS/2
14004     * or clone.  The demo disk uses this driver.  It is suggested that all
14005     * MINIX users try the other drivers, and use this one only as a last resort,
14006     * if all else fails.
14007     *
14008     * (*) The performance is within 10% of the AT driver for reads on any disk
14009     *     and writes on a 2:1 interleaved disk, it will be DMA_BUF_SIZE bytes
14010     *     per revolution for a minimum of 60 kb/s for writes to 1:1 disks.
14011     *
14012     * The file contains one entry point:
14013     *
14014     *       bios_winchester_task:  main entry when system is brought up
14015     *
14016     *
14017     * Changes:
14018     *      30 Apr 1992 by Kees J. Bot: device dependent/independent split.
14019     */
14020
14021    #include "kernel.h"
14022    #include "driver.h"
14023    #include "drvlib.h"
14024
```

```
14025   #if ENABLE_BIOS_WINI
14026
14027   /* If the DMA buffer is large enough then use it always. */
14028   #define USE_BUF         (DMA_BUF_SIZE > BLOCK_SIZE)
14029
14030   /* Error codes */
14031   #define ERR              (-1)   /* general error */
14032
14033   /* Parameters for the disk drive. */
14034   #define MAX_DRIVES         4    /* this driver supports 4 drives (hd0 - hd19)*/
14035   #define MAX_SECS         255    /* bios can transfer this many sectors */
14036   #define NR_DEVICES      (MAX_DRIVES * DEV_PER_DRIVE)
14037   #define SUB_PER_DRIVE   (NR_PARTITIONS * NR_PARTITIONS)
14038   #define NR_SUBDEVS      (MAX_DRIVES * SUB_PER_DRIVE)
14039
14040   /* BIOS parameters */
14041   #define BIOS_ASK        0x08    /* opcode for asking BIOS for parameters */
14042   #define BIOS_RESET      0x00    /* opcode for resetting disk BIOS */
14043   #define BIOS_READ       0x02    /* opcode for BIOS read */
14044   #define BIOS_WRITE      0x03    /* opcode for BIOS write */
14045   #define HD_CODE         0x80    /* BIOS code for hard disk drive 0 */
14046
14047   /* Variables. */
14048   PRIVATE struct wini {           /* main drive struct, one entry per drive */
14049     unsigned cylinders;           /* number of cylinders */
14050     unsigned heads;               /* number of heads */
14051     unsigned sectors;             /* number of sectors per track */
14052     unsigned open_ct;             /* in-use count */
14053     struct device part[DEV_PER_DRIVE];    /* primary partitions: hd[0-4] */
14054     struct device subpart[SUB_PER_DRIVE]; /* subpartitions: hd[1-4][a-d] */
14055   } wini[MAX_DRIVES], *w_wn;
14056
14057   PRIVATE struct trans {
14058     struct iorequest_s *iop;      /* belongs to this I/O request */
14059     unsigned long block;          /* first sector to transfer */
14060     unsigned count;               /* byte count */
14061     phys_bytes phys;              /* user physical address */
14062     phys_bytes dma;               /* DMA physical address */
14063   } wtrans[NR_IOREQS];
14064
14065   PRIVATE int nr_drives = MAX_DRIVES;     /* Number of drives */
14066   PRIVATE struct trans *w_tp;             /* to add transfer requests */
14067   PRIVATE unsigned w_count;               /* number of bytes to transfer */
14068   PRIVATE unsigned long w_nextblock;      /* next block on disk to transfer */
14069   PRIVATE int w_opcode;                   /* DEV_READ or DEV_WRITE */
14070   PRIVATE int w_drive;                    /* selected drive */
14071   PRIVATE struct device *w_dv;            /* device's base and size */
14072   extern unsigned Ax, Bx, Cx, Dx, Es;     /* to hold registers for BIOS calls */
14073
14074   FORWARD _PROTOTYPE( struct device *w_prepare, (int device) );
14075   FORWARD _PROTOTYPE( char *w_name, (void) );
14076   FORWARD _PROTOTYPE( int w_schedule, (int proc_nr, struct iorequest_s *iop) );
14077   FORWARD _PROTOTYPE( int w_finish, (void) );
14078   FORWARD _PROTOTYPE( int w_do_open, (struct driver *dp, message *m_ptr) );
14079   FORWARD _PROTOTYPE( int w_do_close, (struct driver *dp, message *m_ptr) );
14080   FORWARD _PROTOTYPE( void w_init, (void) );
14081   FORWARD _PROTOTYPE( void enable_vectors, (void) );
14082   FORWARD _PROTOTYPE( void w_geometry, (struct partition *entry));
14083
14084
```

```
14085   /* Entry points to this driver. */
14086   PRIVATE struct driver w_dtab = {
14087     w_name,       /* current device's name */
14088     w_do_open,    /* open or mount request, initialize device */
14089     w_do_close,   /* release device */
14090     do_diocntl,   /* get or set a partition's geometry */
14091     w_prepare,    /* prepare for I/O on a given minor device */
14092     w_schedule,   /* precompute cylinder, head, sector, etc. */
14093     w_finish,     /* do the I/O */
14094     nop_cleanup,  /* no cleanup needed */
14095     w_geometry    /* tell the geometry of the disk */
14096   };
14097
14098
14099   /*===========================================================================*
14100    *                               bios_winchester_task                        *
14101    *===========================================================================*/
14102   PUBLIC void bios_winchester_task()
14103   {
14104     driver_task(&w_dtab);
14105   }


14108   /*===========================================================================*
14109    *                               w_prepare                                   *
14110    *===========================================================================*/
14111   PRIVATE struct device *w_prepare(device)
14112   int device;
14113   {
14114   /* Prepare for I/O on a device. */
14115
14116     /* Nothing to transfer as yet. */
14117     w_count = 0;
14118
14119     if (device < NR_DEVICES) {                    /* hd0, hd1, ... */
14120           w_drive = device / DEV_PER_DRIVE;       /* save drive number */
14121           w_wn = &wini[w_drive];
14122           w_dv = &w_wn->part[device % DEV_PER_DRIVE];
14123     } else
14124     if ((unsigned) (device -= MINOR_hd1a) < NR_SUBDEVS) { /* hd1a, hd1b, ... */
14125           w_drive = device / SUB_PER_DRIVE;
14126           w_wn = &wini[w_drive];
14127           w_dv = &w_wn->subpart[device % SUB_PER_DRIVE];
14128     } else {
14129           return(NIL_DEV);
14130     }
14131     return(w_drive < nr_drives ? w_dv : NIL_DEV);
14132   }


14135   /*===========================================================================*
14136    *                               w_name                                      *
14137    *===========================================================================*/
14138   PRIVATE char *w_name()
14139   {
14140   /* Return a name for the current device. */
14141     static char name[] = "bios-hd5";
14142
14143     name[7] = '0' + w_drive * DEV_PER_DRIVE;
14144     return name;
```

```
14145   }


14148   /*===========================================================================*
14149    *                              w_schedule                                   *
14150    *===========================================================================*/
14151   PRIVATE int w_schedule(proc_nr, iop)
14152   int proc_nr;                    /* process doing the request */
14153   struct iorequest_s *iop;        /* pointer to read or write request */
14154   {
14155   /* Gather I/O requests on consecutive blocks so they may be read/written
14156    * in one bios command if using a buffer.  Check and gather all the requests
14157    * and try to finish them as fast as possible if unbuffered.
14158    */
14159     int r, opcode;
14160     unsigned long pos;
14161     unsigned nbytes, count, dma_count;
14162     unsigned long block;
14163     phys_bytes user_phys, dma_phys;
14164
14165     /* This many bytes to read/write */
14166     nbytes = iop->io_nbytes;
14167     if ((nbytes & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);
14168
14169     /* From/to this position on the device */
14170     pos = iop->io_position;
14171     if ((pos & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);
14172
14173     /* To/from this user address */
14174     user_phys = numap(proc_nr, (vir_bytes) iop->io_buf, nbytes);
14175     if (user_phys == 0) return(iop->io_nbytes = EINVAL);
14176
14177     /* Read or write? */
14178     opcode = iop->io_request & ~OPTIONAL_IO;
14179
14180     /* Which block on disk and how close to EOF? */
14181     if (pos >= w_dv->dv_size) return(OK);         /* At EOF */
14182     if (pos + nbytes > w_dv->dv_size) nbytes = w_dv->dv_size - pos;
14183     block = (w_dv->dv_base + pos) >> SECTOR_SHIFT;
14184
14185     if (USE_BUF && w_count > 0 && block != w_nextblock) {
14186           /* This new request can't be chained to the job being built */
14187           if ((r = w_finish()) != OK) return(r);
14188     }
14189
14190     /* The next consecutive block */
14191     if (USE_BUF) w_nextblock = block + (nbytes >> SECTOR_SHIFT);
14192
14193     /* While there are "unscheduled" bytes in the request: */
14194     do {
14195           count = nbytes;
14196
14197           if (USE_BUF) {
14198                   if (w_count == DMA_BUF_SIZE) {
14199                           /* Can't transfer more than the buffer allows. */
14200                           if ((r = w_finish()) != OK) return(r);
14201                   }
14202
14203                   if (w_count + count > DMA_BUF_SIZE)
14204                           count = DMA_BUF_SIZE - w_count;
```

```
14205           } else {
14206                   if (w_tp == wtrans + NR_IOREQS) {
14207                           /* All transfer slots in use. */
14208                           if ((r = w_finish()) != OK) return(r);
14209                   }
14210           }
14211
14212           if (w_count == 0) {
14213                   /* The first request in a row, initialize. */
14214                   w_opcode = opcode;
14215                   w_tp = wtrans;
14216           }
14217
14218           if (USE_BUF) {
14219                   dma_phys = tmp_phys + w_count;
14220           } else {
14221                   /* Memory chunk to DMA. */
14222                   dma_phys = user_phys;
14223                   dma_count = dma_bytes_left(dma_phys);
14224
14225                   if (dma_phys >= (1L << 20)) {
14226                           /* The BIOS can only address the first megabyte. */
14227                           count = SECTOR_SIZE;
14228                           dma_phys = tmp_phys;
14229                   } else
14230                   if (dma_count < count) {
14231                           /* Nearing a 64K boundary. */
14232                           if (dma_count >= SECTOR_SIZE) {
14233                                   /* Can read a few sectors before hitting the
14234                                    * boundary.
14235                                    */
14236                                   count = dma_count & ~SECTOR_MASK;
14237                           } else {
14238                                   /* Must use the special buffer for this. */
14239                                   count = SECTOR_SIZE;
14240                                   dma_phys = tmp_phys;
14241                           }
14242                   }
14243           }
14244
14245           /* Store I/O parameters */
14246           w_tp->iop = iop;
14247           w_tp->block = block;
14248           w_tp->count = count;
14249           w_tp->phys = user_phys;
14250           w_tp->dma = dma_phys;
14251
14252           /* Update counters */
14253           w_tp++;
14254           w_count += count;
14255           block += count >> SECTOR_SHIFT;
14256           user_phys += count;
14257           nbytes -= count;
14258     } while (nbytes > 0);
14259
14260     return(OK);
14261   }


14264   /*============================================================================*
```

```
14265     *                             w_finish                                  *
14266     *=======================================================================*/
14267    PRIVATE int w_finish()
14268    {
14269    /* Carry out the I/O requests gathered in wtrans[]. */
14270
14271      struct trans *tp = wtrans, *tp2;
14272      unsigned count, cylinder, sector, head, hicyl, locyl;
14273      unsigned secspcyl = w_wn->heads * w_wn->sectors;
14274      int many = USE_BUF;
14275
14276      if (w_count == 0) return(OK); /* Spurious finish. */
14277
14278      do {
14279            if (w_opcode == DEV_WRITE) {
14280                    tp2 = tp;
14281                    count = 0;
14282                    do {
14283                            if (USE_BUF || tp2->dma == tmp_phys) {
14284                                    phys_copy(tp2->phys, tp2->dma,
14285                                                    (phys_bytes) tp2->count);
14286                            }
14287                            count += tp2->count;
14288                            tp2++;
14289                    } while (many && count < w_count);
14290            } else {
14291                    count = many ? w_count : tp->count;
14292            }
14293
14294            /* Do the transfer */
14295            cylinder = tp->block / secspcyl;
14296            sector = (tp->block % w_wn->sectors) + 1;
14297            head = (tp->block % secspcyl) / w_wn->sectors;
14298
14299            Ax = w_opcode == DEV_WRITE ? BIOS_WRITE : BIOS_READ;
14300            Ax = (Ax << 8) | (count >> SECTOR_SHIFT);       /* opcode & count */
14301            Bx = (unsigned) tp->dma % HCLICK_SIZE;          /* low order 4 bits */
14302            Es = physb_to_hclick(tp->dma);          /* high order 16 bits */
14303            hicyl = (cylinder & 0x300) >> 8;        /* two high-order bits */
14304            locyl = (cylinder & 0xFF);              /* 8 low-order bits */
14305            Cx = sector | (hicyl << 6) | (locyl << 8);
14306            Dx = (HD_CODE + w_drive) | (head << 8);
14307            level0(bios13);
14308            if ((Ax & 0xFF00) != 0) {
14309                    /* An error occurred, try again block by block unless */
14310                    if (!many) return(tp->iop->io_nbytes = EIO);
14311                    many = 0;
14312                    continue;
14313            }
14314
14315            w_count -= count;
14316
14317            do {
14318                    if (w_opcode == DEV_READ) {
14319                            if (USE_BUF || tp->dma == tmp_phys) {
14320                                    phys_copy(tp->dma, tp->phys,
14321                                                    (phys_bytes) tp->count);
14322                            }
14323                    }
14324                    tp->iop->io_nbytes -= tp->count;
```

```
14325                   count -= tp->count;
14326                   tp++;
14327           } while (count > 0);
14328     } while (w_count > 0);
14329
14330     return(OK);
14331   }


14334   /*===========================================================================*
14335    *                              w_do_open                                    *
14336    *===========================================================================*/
14337   PRIVATE int w_do_open(dp, m_ptr)
14338   struct driver *dp;
14339   message *m_ptr;
14340   {
14341   /* Device open: Initialize the controller and read the partition table. */
14342
14343     static int init_done = FALSE;
14344
14345     if (!init_done) { w_init(); init_done = TRUE; }
14346
14347     if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
14348
14349     if (w_wn->open_ct++ == 0) {
14350           /* Partition the disk. */
14351           partition(&w_dtab, w_drive * DEV_PER_DRIVE, P_PRIMARY);
14352     }
14353     return(OK);
14354   }


14357   /*===========================================================================*
14358    *                              w_do_close                                   *
14359    *===========================================================================*/
14360   PRIVATE int w_do_close(dp, m_ptr)
14361   struct driver *dp;
14362   message *m_ptr;
14363   {
14364   /* Device close: Release a device. */
14365
14366     if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
14367     w_wn->open_ct--;
14368     return(OK);
14369   }


14372   /*===========================================================================*
14373    *                              w_init                                       *
14374    *===========================================================================*/
14375   PRIVATE void w_init()
14376   {
14377   /* This routine is called at startup to initialize the drive parameters. */
14378
14379     int drive;
14380     struct wini *wn;
14381
14382     /* Enable real mode BIOS vectors. */
14383     enable_vectors();
14384
```

```
14385     /* Set the geometry of the drives */
14386     for (drive = 0; drive < nr_drives; drive++) {
14387           (void) w_prepare(drive * DEV_PER_DRIVE);
14388           wn = w_wn;
14389           Dx = drive + HD_CODE;
14390           Ax = (BIOS_ASK << 8);
14391           level0(bios13);
14392           nr_drives = (Ax & 0xFF00) == 0 ? (Dx & 0xFF) : drive;
14393           if (nr_drives > MAX_DRIVES) nr_drives = MAX_DRIVES;
14394           if (drive >= nr_drives) break;
14395
14396           wn->heads = (Dx >> 8) + 1;
14397           wn->sectors = (Cx & 0x3F);
14398           wn->cylinders = ((Cx << 2) & 0x0300) + ((Cx >> 8) & 0x00FF) + 1;
14399
14400           wn->part[0].dv_size = ((unsigned long) wn->cylinders
14401                           * wn->heads * wn->sectors) << SECTOR_SHIFT;
14402
14403           printf("%s: %d cylinders, %d heads, %d sectors per track\n",
14404                   w_name(), wn->cylinders, wn->heads, wn->sectors);
14405     }
14406   }


14409   /*============================================================================*
14410    *                              enable_vectors                                *
14411    *============================================================================*/
14412   PRIVATE void enable_vectors()
14413   {
14414   /* Protected mode Minix has reprogrammed the interrupt controllers to
14415    * use different vectors from the BIOS.  This means that the BIOS vectors
14416    * must be copied to the Minix locations for use in real mode.  The bios13()
14417    * function enables all interrupts that Minix doesn't use, and masks all
14418    * interrupts that Minix does use when it makes the "INT 13" call.  Alas
14419    * more than one clock tick may occur while the disk is active, so we need
14420    * a special real mode clock interrupt handle to gather lost ticks.
14421    */
14422
14423     int vec, irq;
14424     static u8_t clock_handler[] = {
14425           0x50, 0xB0, 0x20, 0xE6, 0x20, 0xEB, 0x06, 0xE4,
14426           0x61, 0x0C, 0x80, 0xE6, 0x61, 0x58, 0x53, 0x1E,
14427           0xE8, 0x00, 0x00, 0x5B, 0x2E, 0xC5, 0x5F, 0x0A,
14428           0xFF, 0x07, 0x1F, 0x5B, 0xCF,
14429           0x00, 0x00, 0x00, 0x00,
14430     };
14431     u16_t vector[2];
14432
14433     if (!protected_mode) return;
14434
14435     for (irq = 0; irq < NR_IRQ_VECTORS; irq++) {
14436           phys_copy(BIOS_VECTOR(irq)*4L, VECTOR(irq)*4L, 4L);
14437     }
14438
14439     /* Install a clock interrupt handler to collect clock ticks when the
14440      * BIOS disk driver is active.  The handler is a simple bit of 8086 code
14441      * that increments "lost_ticks".
14442      */
14443
14444     /* Let the clock interrupt point to the handler. */
```

```
14445       vector[0] = vir2phys(clock_handler) % HCLICK_SIZE;
14446       vector[1] = vir2phys(clock_handler) / HCLICK_SIZE;
14447       phys_copy(vir2phys(vector), VECTOR(CLOCK_IRQ)*4L, 4L);
14448
14449       /* Store the address of lost_ticks in the handler. */
14450       vector[0] = vir2phys(&lost_ticks) % HCLICK_SIZE;
14451       vector[1] = vir2phys(&lost_ticks) / HCLICK_SIZE;
14452       memcpy(clock_handler + 0x1D, vector, 4);
14453
14454       if (ps_mca) clock_handler[6]= 0;      /* (PS/2 port B clock ack) */
14455     }


14458     /*==========================================================================*
14459      *                                w_geometry                                *
14460      *==========================================================================*/
14461     PRIVATE void w_geometry(entry)
14462     struct partition *entry;
14463     {
14464       entry->cylinders = w_wn->cylinders;
14465       entry->heads = w_wn->heads;
14466       entry->sectors = w_wn->sectors;
14467     }
14468     #endif /* ENABLE_BIOS_WINI */


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/clock.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

14500     /* This file contains the code and data for the clock task.  The clock task
14501      * accepts six message types:
14502      *
14503      *   HARD_INT:    a clock interrupt has occurred
14504      *   GET_UPTIME:  get the time since boot in ticks
14505      *   GET_TIME:    a process wants the real time in seconds
14506      *   SET_TIME:    a process wants to set the real time in seconds
14507      *   SET_ALARM:   a process wants to be alerted after a specified interval
14508      *   SET_SYN_AL:  set the sync alarm
14509      *
14510      *
14511      * The input message is format m6.  The parameters are as follows:
14512      *
14513      *     m_type    CLOCK_PROC   FUNC    NEW_TIME
14514      * ----------------------------------------
14515      * | HARD_INT   |          |         |         |
14516      * |------------+----------+---------+---------|
14517      * | GET_UPTIME |          |         |         |
14518      * |------------+----------+---------+---------|
14519      * | GET_TIME   |          |         |         |
14520      * |------------+----------+---------+---------|
14521      * | SET_TIME   |          |         | newtime |
14522      * |------------+----------+---------+---------|
14523      * | SET_ALARM  | proc_nr  |f to call|  delta  |
14524      * |------------+----------+---------+---------|
14525      * | SET_SYN_AL | proc_nr  |         |  delta  |
14526      * ----------------------------------------
14527      * NEW_TIME, DELTA_CLICKS, and SECONDS_LEFT all refer to the same field in
14528      * the message, depending upon the message type.
14529      *
```

```
14530     * Reply messages are of type OK, except in the case of a HARD_INT, to
14531     * which no reply is generated. For the GET_* messages the time is returned
14532     * in the NEW_TIME field, and for the SET_ALARM and SET_SYN_AL the time
14533     * in seconds remaining until the alarm is returned is returned in the same
14534     * field.
14535     *
14536     * When an alarm goes off, if the caller is a user process, a SIGALRM signal
14537     * is sent to it.  If it is a task, a function specified by the caller will
14538     * be invoked.  This function may, for example, send a message, but only if
14539     * it is certain that the task will be blocked when the timer goes off. A
14540     * synchronous alarm sends a message to the synchronous alarm task, which
14541     * in turn can dispatch a message to another server. This is the only way
14542     * to send an alarm to a server, since servers cannot use the function-call
14543     * mechanism available to tasks and servers cannot receive signals.
14544     */
14545
14546    #include "kernel.h"
14547    #include <signal.h>
14548    #include <minix/callnr.h>
14549    #include <minix/com.h>
14550    #include "proc.h"
14551
14552    /* Constant definitions. */
14553    #define MILLISEC         100    /* how often to call the scheduler (msec) */
14554    #define SCHED_RATE (MILLISEC*HZ/1000)   /* number of ticks per schedule */
14555
14556    /* Clock parameters. */
14557    #if (CHIP == INTEL)
14558    #define COUNTER_FREQ (2*TIMER_FREQ)     /* counter frequency using sqare wave*/
14559    #define LATCH_COUNT     0x00    /* cc00xxxx, c = channel, x = any */
14560    #define SQUARE_WAVE     0x36    /* ccaammmb, a = access, m = mode, b = BCD */
14561                                    /*   11x11, 11 = LSB then MSB, x11 = sq wave */
14562    #define TIMER_COUNT ((unsigned) (TIMER_FREQ/HZ)) /* initial value for counter*/
14563    #define TIMER_FREQ  1193182L    /* clock frequency for timer in PC and AT */
14564
14565    #define CLOCK_ACK_BIT   0x80    /* PS/2 clock interrupt acknowledge bit */
14566    #endif
14567
14568    #if (CHIP == M68000)
14569    #define TIMER_FREQ  2457600L    /* timer 3 input clock frequency */
14570    #endif
14571
14572    /* Clock task variables. */
14573    PRIVATE clock_t realtime;       /* real time clock */
14574    PRIVATE time_t boot_time;       /* time in seconds of system boot */
14575    PRIVATE clock_t next_alarm;     /* probable time of next alarm */
14576    PRIVATE message mc;             /* message buffer for both input and output */
14577    PRIVATE int watchdog_proc;      /* contains proc_nr at call of *watch_dog[]*/
14578    PRIVATE watchdog_t watch_dog[NR_TASKS+NR_PROCS];
14579
14580    /* Variables used by both clock task and synchronous alarm task */
14581    PRIVATE int syn_al_alive= TRUE; /* don't wake syn_alrm_task before inited*/
14582    PRIVATE int syn_table[NR_TASKS+NR_PROCS]; /* which tasks get CLOCK_INT*/
14583
14584    /* Variables changed by interrupt handler */
14585    PRIVATE clock_t pending_ticks;  /* ticks seen by low level only */
14586    PRIVATE int sched_ticks = SCHED_RATE;   /* counter: when 0, call scheduler */
14587    PRIVATE struct proc *prev_ptr;  /* last user process run by clock task */
14588
14589    FORWARD _PROTOTYPE( void common_setalarm, (int proc_nr,
```

```
14590                 long delta_ticks, watchdog_t fuction) );
14591   FORWARD _PROTOTYPE( void do_clocktick, (void) );
14592   FORWARD _PROTOTYPE( void do_get_time, (void) );
14593   FORWARD _PROTOTYPE( void do_getuptime, (void) );
14594   FORWARD _PROTOTYPE( void do_set_time, (message *m_ptr) );
14595   FORWARD _PROTOTYPE( void do_setalarm, (message *m_ptr) );
14596   FORWARD _PROTOTYPE( void init_clock, (void) );
14597   FORWARD _PROTOTYPE( void cause_alarm, (void) );
14598   FORWARD _PROTOTYPE( void do_setsyn_alrm, (message *m_ptr) );
14599   FORWARD _PROTOTYPE( int clock_handler, (int irq) );
14600
14601   /*===========================================================================*
14602    *                              clock_task                                   *
14603    *===========================================================================*/
14604   PUBLIC void clock_task()
14605   {
14606   /* Main program of clock task.  It corrects realtime by adding pending
14607    * ticks seen only by the interrupt service, then it determines which
14608    * of the 6 possible calls this is by looking at 'mc.m_type'.  Then
14609    * it dispatches.
14610    */
14611
14612     int opcode;
14613
14614     init_clock();                /* initialize clock task */
14615
14616     /* Main loop of the clock task.  Get work, process it, sometimes reply. */
14617     while (TRUE) {
14618        receive(ANY, &mc);        /* go get a message */
14619        opcode = mc.m_type;       /* extract the function code */
14620
14621        lock();
14622        realtime += pending_ticks; /* transfer ticks from low level handler */
14623        pending_ticks = 0;        /* so we don't have to worry about them */
14624        unlock();
14625
14626        switch (opcode) {
14627           case HARD_INT:   do_clocktick();        break;
14628           case GET_UPTIME: do_getuptime();        break;
14629           case GET_TIME:   do_get_time();         break;
14630           case SET_TIME:   do_set_time(&mc);      break;
14631           case SET_ALARM:  do_setalarm(&mc);      break;
14632           case SET_SYNC_AL:do_setsyn_alrm(&mc);   break;
14633           default: panic("clock task got bad message", mc.m_type);
14634        }
14635
14636       /* Send reply, except for clock tick. */
14637       mc.m_type = OK;
14638       if (opcode != HARD_INT) send(mc.m_source, &mc);
14639     }
14640   }


14643   /*===========================================================================*
14644    *                              do_clocktick                                 *
14645    *===========================================================================*/
14646   PRIVATE void do_clocktick()
14647   {
14648   /* Despite its name, this routine is not called on every clock tick. It
14649    * is called on those clock ticks when a lot of work needs to be done.
```

```
14650     */
14651
14652      register struct proc *rp;
14653      register int proc_nr;
14654
14655      if (next_alarm <= realtime) {
14656            /* An alarm may have gone off, but proc may have exited, so check. */
14657            next_alarm = LONG_MAX;  /* start computing next alarm */
14658            for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++) {
14659                    if (rp->p_alarm != 0) {
14660                            /* See if this alarm time has been reached. */
14661                            if (rp->p_alarm <= realtime) {
14662                                    /* A timer has gone off.  If it is a user proc,
14663                                     * send it a signal.  If it is a task, call the
14664                                     * function previously specified by the task.
14665                                     */
14666                                    proc_nr = proc_number(rp);
14667                                    if (watch_dog[proc_nr+NR_TASKS]) {
14668                                            watchdog_proc= proc_nr;
14669                                            (*watch_dog[proc_nr+NR_TASKS])();
14670                                    }
14671                                    else
14672                                            cause_sig(proc_nr, SIGALRM);
14673                                    rp->p_alarm = 0;
14674                            }
14675
14676                            /* Work on determining which alarm is next. */
14677                            if (rp->p_alarm != 0 && rp->p_alarm < next_alarm)
14678                                    next_alarm = rp->p_alarm;
14679                    }
14680            }
14681      }
14682
14683      /* If a user process has been running too long, pick another one. */
14684      if (--sched_ticks == 0) {
14685            if (bill_ptr == prev_ptr) lock_sched(); /* process has run too long */
14686            sched_ticks = SCHED_RATE;               /* reset quantum */
14687            prev_ptr = bill_ptr;                    /* new previous process */
14688      }
14689    #if (SHADOWING == 1)
14690      if (rdy_head[SHADOW_Q]) unshadow(rdy_head[SHADOW_Q]);
14691    #endif
14692    }


14695    /*===========================================================================*
14696     *                              do_getuptime                                 *
14697     *===========================================================================*/
14698    PRIVATE void do_getuptime()
14699    {
14700    /* Get and return the current clock uptime in ticks. */
14701
14702      mc.NEW_TIME = realtime;       /* current uptime */
14703    }


14706    /*===========================================================================*
14707     *                              get_uptime                                   *
14708     *===========================================================================*/
14709    PUBLIC clock_t get_uptime()
```

```
14710   {
14711   /* Get and return the current clock uptime in ticks.  This function is
14712    * designed to be called from other tasks, so they can get uptime without
14713    * the overhead of messages. It has to be careful about pending_ticks.
14714    */
14715
14716     clock_t uptime;
14717
14718     lock();
14719     uptime = realtime + pending_ticks;
14720     unlock();
14721     return(uptime);
14722   }


14725   /*===========================================================================*
14726    *                                do_get_time                                *
14727    *===========================================================================*/
14728   PRIVATE void do_get_time()
14729   {
14730   /* Get and return the current clock time in seconds. */
14731
14732     mc.NEW_TIME = boot_time + realtime/HZ;        /* current real time */
14733   }


14736   /*===========================================================================*
14737    *                                do_set_time                                *
14738    *===========================================================================*/
14739   PRIVATE void do_set_time(m_ptr)
14740   message *m_ptr;               /* pointer to request message */
14741   {
14742   /* Set the real time clock.  Only the superuser can use this call. */
14743
14744     boot_time = m_ptr->NEW_TIME - realtime/HZ;
14745   }


14748   /*===========================================================================*
14749    *                                do_setalarm                                *
14750    *===========================================================================*/
14751   PRIVATE void do_setalarm(m_ptr)
14752   message *m_ptr;               /* pointer to request message */
14753   {
14754   /* A process wants an alarm signal or a task wants a given watch_dog function
14755    * called after a specified interval.
14756    */
14757
14758     register struct proc *rp;
14759     int proc_nr;                /* which process wants the alarm */
14760     long delta_ticks;           /* in how many clock ticks does he want it? */
14761     watchdog_t function;        /* function to call (tasks only) */
14762
14763     /* Extract the parameters from the message. */
14764     proc_nr = m_ptr->CLOCK_PROC_NR;       /* process to interrupt later */
14765     delta_ticks = m_ptr->DELTA_TICKS;     /* how many ticks to wait */
14766     function = (watchdog_t) m_ptr->FUNC_TO_CALL;
14767                                           /* function to call (tasks only) */
14768     rp = proc_addr(proc_nr);
14769     mc.SECONDS_LEFT = (rp->p_alarm == 0 ? 0 : (rp->p_alarm - realtime)/HZ );
```

```
14770     if (!istaskp(rp)) function= 0;        /* user processes get signaled */
14771     common_setalarm(proc_nr, delta_ticks, function);
14772   }


14775   /*===========================================================================*
14776    *                                do_setsyn_alrm                             *
14777    *===========================================================================*/
14778   PRIVATE void do_setsyn_alrm(m_ptr)
14779   message *m_ptr;                 /* pointer to request message */
14780   {
14781   /* A process wants a synchronous alarm.
14782    */
14783
14784     register struct proc *rp;
14785     int proc_nr;                  /* which process wants the alarm */
14786     long delta_ticks;             /* in how many clock ticks does he want it? */
14787
14788     /* Extract the parameters from the message. */
14789     proc_nr = m_ptr->CLOCK_PROC_NR;       /* process to interrupt later */
14790     delta_ticks = m_ptr->DELTA_TICKS;     /* how many ticks to wait */
14791     rp = proc_addr(proc_nr);
14792     mc.SECONDS_LEFT = (rp->p_alarm == 0 ? 0 : (rp->p_alarm - realtime)/HZ );
14793     common_setalarm(proc_nr, delta_ticks, cause_alarm);
14794   }


14797   /*===========================================================================*
14798    *                                common_setalarm                            *
14799    *===========================================================================*/
14800   PRIVATE void common_setalarm(proc_nr, delta_ticks, function)
14801   int proc_nr;                    /* which process wants the alarm */
14802   long delta_ticks;               /* in how many clock ticks does he want it? */
14803   watchdog_t function;            /* function to call (0 if cause_sig is
14804                                     * to be called */
14805   {
14806   /* Finish up work of do_set_alarm and do_setsyn_alrm.  Record an alarm
14807    * request and check to see if it is the next alarm needed.
14808    */
14809
14810     register struct proc *rp;
14811
14812     rp = proc_addr(proc_nr);
14813     rp->p_alarm = (delta_ticks == 0 ? 0 : realtime + delta_ticks);
14814     watch_dog[proc_nr+NR_TASKS] = function;
14815
14816     /* Which alarm is next? */
14817     next_alarm = LONG_MAX;
14818     for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++)
14819           if(rp->p_alarm != 0 && rp->p_alarm < next_alarm)next_alarm=rp->p_alarm;
14820
14821   }


14824   /*===========================================================================*
14825    *                                cause_alarm                                *
14826    *===========================================================================*/
14827   PRIVATE void cause_alarm()
14828   {
14829   /* Routine called if a timer goes off and the process requested a synchronous
```

```
14830     * alarm. The process number is in the global variable watchdog_proc (HACK).
14831     */
14832      message mess;
14833
14834      syn_table[watchdog_proc + NR_TASKS]= TRUE;
14835      if (!syn_al_alive) send (SYN_ALRM_TASK, &mess);
14836    }


14839    /*===========================================================================*
14840     *                              syn_alrm_task                                *
14841     *===========================================================================*/
14842    PUBLIC void syn_alrm_task()
14843    {
14844    /* Main program of the synchronous alarm task.
14845     * This task receives messages only from cause_alarm in the clock task.
14846     * It sends a CLOCK_INT message to a process that requested a syn_alrm.
14847     * Synchronous alarms are so called because, unlike a signals or the
14848     * activation of a watchdog, a synchronous alarm is received by a process
14849     * when it is in a known part of its code, that is, when it has issued
14850     * a call to receive a message.
14851     */
14852
14853      message mess;
14854      int work_done;        /* ready to sleep ? */
14855      int *al_ptr;          /* pointer in syn_table */
14856      int i;
14857
14858      syn_al_alive= TRUE;
14859      for (i= 0, al_ptr= syn_table; i<NR_TASKS+NR_PROCS; i++, al_ptr++)
14860            *al_ptr= FALSE;
14861
14862      while (TRUE) {
14863            work_done= TRUE;
14864            for (i= 0, al_ptr= syn_table; i<NR_TASKS+NR_PROCS; i++, al_ptr++)
14865                    if (*al_ptr) {
14866                            *al_ptr= FALSE;
14867                            mess.m_type= CLOCK_INT;
14868                            send (i-NR_TASKS, &mess);
14869                            work_done= FALSE;
14870                    }
14871            if (work_done) {
14872                    syn_al_alive= FALSE;
14873                    receive (CLOCK, &mess);
14874                    syn_al_alive= TRUE;
14875            }
14876      }
14877    }


14880    /*===========================================================================*
14881     *                              clock_handler                                *
14882     *===========================================================================*/
14883    PRIVATE int clock_handler(irq)
14884    int irq;
14885    {
14886    /* This executes on every clock tick (i.e., every time the timer chip
14887     * generates an interrupt). It does a little bit of work so the clock
14888     * task does not have to be called on every tick.
14889     *
```

```
14890     * Switch context to do_clocktick if an alarm has gone off.
14891     * Also switch there to reschedule if the reschedule will do something.
14892     * This happens when
14893     *      (1) quantum has expired
14894     *      (2) current process received full quantum (as clock sampled it!)
14895     *      (3) something else is ready to run.
14896     * Also call TTY and PRINTER and let them do whatever is necessary.
14897     *
14898     * Many global global and static variables are accessed here.  The safety
14899     * of this must be justified.  Most of them are not changed here:
14900     *      k_reenter:
14901     *              This safely tells if the clock interrupt is nested.
14902     *      proc_ptr, bill_ptr:
14903     *              These are used for accounting.  It does not matter if proc.c
14904     *              is changing them, provided they are always valid pointers,
14905     *              since at worst the previous process would be billed.
14906     *      next_alarm, realtime, sched_ticks, bill_ptr, prev_ptr,
14907     *      rdy_head[USER_Q]:
14908     *              These are tested to decide whether to call interrupt().  It
14909     *              does not matter if the test is sometimes (rarely) backwards
14910     *              due to a race, since this will only delay the high-level
14911     *              processing by one tick, or call the high level unnecessarily.
14912     * The variables which are changed require more care:
14913     *      rp->user_time, rp->sys_time:
14914     *              These are protected by explicit locks in system.c.  They are
14915     *              not properly protected in dmp.c (the increment here is not
14916     *              atomic) but that hardly matters.
14917     *      pending_ticks:
14918     *              This is protected by explicit locks in clock.c.  Don't
14919     *              update realtime directly, since there are too many
14920     *              references to it to guard conveniently.
14921     *      lost_ticks:
14922     *              Clock ticks counted outside the clock task.
14923     *      sched_ticks, prev_ptr:
14924     *              Updating these competes with similar code in do_clocktick().
14925     *              No lock is necessary, because if bad things happen here
14926     *              (like sched_ticks going negative), the code in do_clocktick()
14927     *              will restore the variables to reasonable values, and an
14928     *              occasional missed or extra sched() is harmless.
14929     *
14930     * Are these complications worth the trouble?  Well, they make the system 15%
14931     * faster on a 5MHz 8088, and make task debugging much easier since there are
14932     * no task switches on an inactive system.
14933     */
14934
14935      register struct proc *rp;
14936      register unsigned ticks;
14937      clock_t now;
14938
14939      if (ps_mca) {
14940            /* Acknowledge the PS/2 clock interrupt. */
14941            out_byte(PORT_B, in_byte(PORT_B) | CLOCK_ACK_BIT);
14942      }
14943
14944      /* Update user and system accounting times.
14945       * First charge the current process for user time.
14946       * If the current process is not the billable process (usually because it
14947       * is a task), charge the billable process for system time as well.
14948       * Thus the unbillable tasks' user time is the billable users' system time.
14949       */
```

```
14950     if (k_reenter != 0)
14951           rp = proc_addr(HARDWARE);
14952     else
14953           rp = proc_ptr;
14954     ticks = lost_ticks + 1;
14955     lost_ticks = 0;
14956     rp->user_time += ticks;
14957     if (rp != bill_ptr && rp != proc_addr(IDLE)) bill_ptr->sys_time += ticks;
14958
14959     pending_ticks += ticks;
14960     now = realtime + pending_ticks;
14961     if (tty_timeout <= now) tty_wakeup(now);      /* possibly wake up TTY */
14962   #if (CHIP != M68000)
14963     pr_restart();                                 /* possibly restart printer */
14964   #endif
14965   #if (CHIP == M68000)
14966     kb_timer();                                   /* keyboard repeat */
14967     if (sched_ticks == 1) fd_timer();             /* floppy deselect */
14968   #endif
14969
14970     if (next_alarm <= now ||
14971         sched_ticks == 1 &&
14972         bill_ptr == prev_ptr &&
14973   #if (SHADOWING == 0)
14974         rdy_head[USER_Q] != NIL_PROC) {
14975   #else
14976         (rdy_head[USER_Q] != NIL_PROC || rdy_head[SHADOW_Q] != NIL_PROC)) {
14977   #endif
14978           interrupt(CLOCK);
14979           return 1;       /* Reenable interrupts */
14980     }
14981
14982     if (--sched_ticks == 0) {
14983           /* If bill_ptr == prev_ptr, no ready users so don't need sched(). */
14984           sched_ticks = SCHED_RATE;       /* reset quantum */
14985           prev_ptr = bill_ptr;            /* new previous process */
14986     }
14987     return 1;     /* Reenable clock interrupt */
14988   }

14990   #if (CHIP == INTEL)
14991
14992   /*============================================================================*
14993    *                              init_clock                                    *
14994    *============================================================================*/
14995   PRIVATE void init_clock()
14996   {
14997   /* Initialize channel 0 of the 8253A timer to e.g. 60 Hz. */
14998
14999     out_byte(TIMER_MODE, SQUARE_WAVE);    /* set timer to run continuously */
15000     out_byte(TIMER0, TIMER_COUNT);        /* load timer low byte */
15001     out_byte(TIMER0, TIMER_COUNT >> 8);   /* load timer high byte */
15002     put_irq_handler(CLOCK_IRQ, clock_handler);    /* set the interrupt handler */
15003     enable_irq(CLOCK_IRQ);                /* ready for clock interrupts */
15004   }


15007   /*============================================================================*
15008    *                              clock_stop                                    *
15009    *============================================================================*/
```

```
15010   PUBLIC void clock_stop()
15011   {
15012   /* Reset the clock to the BIOS rate. (For rebooting) */
15013
15014     out_byte(TIMER_MODE, 0x36);
15015     out_byte(TIMER0, 0);
15016     out_byte(TIMER0, 0);
15017   }


15020   /*===========================================================================*
15021    *                              milli_delay                                  *
15022    *===========================================================================*/
15023   PUBLIC void milli_delay(millisec)
15024   unsigned millisec;
15025   {
15026   /* Delay some milliseconds. */
15027
15028     struct milli_state ms;
15029
15030     milli_start(&ms);
15031     while (milli_elapsed(&ms) < millisec) {}
15032   }

15034   /*===========================================================================*
15035    *                              milli_start                                  *
15036    *===========================================================================*/
15037   PUBLIC void milli_start(msp)
15038   struct milli_state *msp;
15039   {
15040   /* Prepare for calls to milli_elapsed(). */
15041
15042     msp->prev_count = 0;
15043     msp->accum_count = 0;
15044   }


15047   /*===========================================================================*
15048    *                              milli_elapsed                                *
15049    *===========================================================================*/
15050   PUBLIC unsigned milli_elapsed(msp)
15051   struct milli_state *msp;
15052   {
15053   /* Return the number of milliseconds since the call to milli_start().  Must be
15054    * polled rapidly.
15055    */
15056     unsigned count;
15057
15058     /* Read the counter for channel 0 of the 8253A timer.  The counter
15059      * decrements at twice the timer frequency (one full cycle for each
15060      * half of square wave).  The counter normally has a value between 0
15061      * and TIMER_COUNT, but before the clock task has been initialized,
15062      * its maximum value is 65535, as set by the BIOS.
15063      */
15064     out_byte(TIMER_MODE, LATCH_COUNT);    /* make chip copy count to latch */
15065     count = in_byte(TIMER0);      /* countdown continues during 2-step read */
15066     count |= in_byte(TIMER0) << 8;
15067
15068     /* Add difference between previous and new count unless the counter has
15069      * increased (restarted its cycle).  We may lose a tick now and then, but
```

```
15070       * microsecond precision is not needed.
15071       */
15072      msp->accum_count += count <= msp->prev_count ? (msp->prev_count - count) : 1;
15073      msp->prev_count = count;
15074
15075      return msp->accum_count / (TIMER_FREQ / 1000);
15076    }
15077    #endif /* (CHIP == INTEL) */
15078
15079
15080    #if (CHIP == M68000)
15081    #include "staddr.h"
15082    #include "stmfp.h"
15083
15084    /*===========================================================================*
15085     *                              init_clock                                   *
15086     *===========================================================================*/
15087    PRIVATE void init_clock()
15088    {
15089    /* Initialize the timer C in the MFP 68901.
15090     * Reducing to HZ is not possible by hardware.  The resulting interrupt
15091     * rate is further reduced by software with a factor of 4.
15092     * Note that the expression below works for both HZ=50 and HZ=60.
15093     */
15094      do {
15095            MFP->mf_tcdr = TIMER_FREQ/(64*4*HZ);
15096      } while ((MFP->mf_tcdr & 0xFF) != TIMER_FREQ/(64*4*HZ));
15097      MFP->mf_tcdcr |= (T_Q064<<4);
15098    }
15099    #endif /* (CHIP == M68000) */
```

++
src/kernel/console.c
++

```
15100    /* Code and data for the IBM console driver.
15101     *
15102     * The 6845 video controller used by the IBM PC shares its video memory with
15103     * the CPU somewhere in the 0xB0000 memory bank.  To the 6845 this memory
15104     * consists of 16-bit words.  Each word has a character code in the low byte
15105     * and a so-called attribute byte in the high byte.  The CPU directly modifies
15106     * video memory to display characters, and sets two registers on the 6845 that
15107     * specify the video origin and the cursor position.  The video origin is the
15108     * place in video memory where the first character (upper left corner) can
15109     * be found.  Moving the origin is a fast way to scroll the screen.  Some
15110     * video adapters wrap around the top of video memory, so the origin can
15111     * move without bounds.  For other adapters screen memory must sometimes be
15112     * moved to reset the origin.  All computations on video memory use character
15113     * (word) addresses for simplicity and assume there is no wrapping.  The
15114     * assembly support functions translate the word addresses to byte addresses
15115     * and the scrolling function worries about wrapping.
15116     */
15117
15118    #include "kernel.h"
15119    #include <termios.h>
```

```
15120    #include <minix/callnr.h>
15121    #include <minix/com.h>
15122    #include "protect.h"
15123    #include "tty.h"
15124    #include "proc.h"
15125
15126    /* Definitions used by the console driver. */
15127    #define MONO_BASE    0xB0000L   /* base of mono video memory */
15128    #define COLOR_BASE   0xB8000L   /* base of color video memory */
15129    #define MONO_SIZE     0x1000    /* 4K mono video memory */
15130    #define COLOR_SIZE    0x4000    /* 16K color video memory */
15131    #define EGA_SIZE      0x8000    /* EGA & VGA have at least 32K */
15132    #define BLANK_COLOR   0x0700    /* determines cursor color on blank screen */
15133    #define SCROLL_UP          0    /* scroll forward */
15134    #define SCROLL_DOWN        1    /* scroll backward */
15135    #define BLANK_MEM ((u16_t *) 0) /* tells mem_vid_copy() to blank the screen */
15136    #define CONS_RAM_WORDS    80    /* video ram buffer size */
15137    #define MAX_ESC_PARMS      2    /* number of escape sequence params allowed */
15138
15139    /* Constants relating to the controller chips. */
15140    #define M_6845         0x3B4    /* port for 6845 mono */
15141    #define C_6845         0x3D4    /* port for 6845 color */
15142    #define EGA            0x3C4    /* port for EGA card */
15143    #define INDEX              0    /* 6845's index register */
15144    #define DATA               1    /* 6845's data register */
15145    #define VID_ORG           12    /* 6845's origin register */
15146    #define CURSOR            14    /* 6845's cursor register */
15147
15148    /* Beeper. */
15149    #define BEEP_FREQ     0x0533    /* value to put into timer to set beep freq */
15150    #define B_TIME             3    /* length of CTRL-G beep is ticks */
15151
15152    /* definitions used for font management */
15153    #define GA_SEQUENCER_INDEX      0x3C4
15154    #define GA_SEQUENCER_DATA       0x3C5
15155    #define GA_GRAPHICS_INDEX       0x3CE
15156    #define GA_GRAPHICS_DATA        0x3CF
15157    #define GA_VIDEO_ADDRESS        0xA0000L
15158    #define GA_FONT_SIZE            8192
15159
15160    /* Global variables used by the console driver. */
15161    PUBLIC unsigned vid_seg;        /* video ram selector (0xB0000 or 0xB8000) */
15162    PUBLIC unsigned vid_size;       /* 0x2000 for color or 0x0800 for mono */
15163    PUBLIC unsigned vid_mask;       /* 0x1FFF for color or 0x07FF for mono */
15164    PUBLIC unsigned blank_color = BLANK_COLOR; /* display code for blank */
15165
15166    /* Private variables used by the console driver. */
15167    PRIVATE int vid_port;           /* I/O port for accessing 6845 */
15168    PRIVATE int wrap;               /* hardware can wrap? */
15169    PRIVATE int softscroll;         /* 1 = software scrolling, 0 = hardware */
15170    PRIVATE unsigned vid_base;      /* base of video ram (0xB000 or 0xB800) */
15171    PRIVATE int beeping;            /* speaker is beeping? */
15172    #define scr_width       80      /* # characters on a line */
15173    #define scr_lines       25      /* # lines on the screen */
15174    #define scr_size        (80*25) /* # characters on the screen */
15175
15176    /* Per console data. */
15177    typedef struct console {
15178      tty_t *c_tty;                 /* associated TTY struct */
15179      int c_column;                 /* current column number (0-origin) */
```

```
15180     int c_row;                    /* current row (0 at top of screen) */
15181     int c_rwords;                 /* number of WORDS (not bytes) in outqueue */
15182     unsigned c_start;             /* start of video memory of this console */
15183     unsigned c_limit;             /* limit of this console's video memory */
15184     unsigned c_org;               /* location in RAM where 6845 base points */
15185     unsigned c_cur;               /* current position of cursor in video RAM */
15186     unsigned c_attr;              /* character attribute */
15187     unsigned c_blank;             /* blank attribute */
15188     char c_esc_state;             /* 0=normal, 1=ESC, 2=ESC[ */
15189     char c_esc_intro;             /* Distinguishing character following ESC */
15190     int *c_esc_parmp;             /* pointer to current escape parameter */
15191     int c_esc_parmv[MAX_ESC_PARMS];       /* list of escape parameters */
15192     u16_t c_ramqueue[CONS_RAM_WORDS];     /* buffer for video RAM */
15193   } console_t;
15194
15195   PRIVATE int nr_cons= 1;         /* actual number of consoles */
15196   PRIVATE console_t cons_table[NR_CONS];
15197   PRIVATE console_t *curcons;     /* currently visible */
15198
15199   /* Color if using a color controller. */
15200   #define color   (vid_port == C_6845)
15201
15202   /* Map from ANSI colors to the attributes used by the PC */
15203   PRIVATE int ansi_colors[8] = {0, 4, 2, 6, 1, 5, 3, 7};
15204
15205   /* Structure used for font management */
15206   struct sequence {
15207           unsigned short index;
15208           unsigned char port;
15209           unsigned char value;
15210   };
15211
15212   FORWARD _PROTOTYPE( void cons_write, (struct tty *tp)                   );
15213   FORWARD _PROTOTYPE( void cons_echo, (tty_t *tp, int c)                  );
15214   FORWARD _PROTOTYPE( void out_char, (console_t *cons, int c)             );
15215   FORWARD _PROTOTYPE( void beep, (void)                                   );
15216   FORWARD _PROTOTYPE( void do_escape, (console_t *cons, int c)            );
15217   FORWARD _PROTOTYPE( void flush, (console_t *cons)                       );
15218   FORWARD _PROTOTYPE( void parse_escape, (console_t *cons, int c)         );
15219   FORWARD _PROTOTYPE( void scroll_screen, (console_t *cons, int dir)      );
15220   FORWARD _PROTOTYPE( void set_6845, (int reg, unsigned val)              );
15221   FORWARD _PROTOTYPE( void stop_beep, (void)                              );
15222   FORWARD _PROTOTYPE( void cons_org0, (void)                              );
15223   FORWARD _PROTOTYPE( void ga_program, (struct sequence *seq) );
15224
15225
15226   /*===========================================================================*
15227    *                              cons_write                                   *
15228    *===========================================================================*/
15229   PRIVATE void cons_write(tp)
15230   register struct tty *tp;        /* tells which terminal is to be used */
15231   {
15232   /* Copy as much data as possible to the output queue, then start I/O.  On
15233    * memory-mapped terminals, such as the IBM console, the I/O will also be
15234    * finished, and the counts updated.  Keep repeating until all I/O done.
15235    */
15236
15237     int count;
15238     register char *tbuf;
15239     char buf[64];
```

```
15240       phys_bytes user_phys;
15241       console_t *cons = tp->tty_priv;
15242
15243       /* Check quickly for nothing to do, so this can be called often without
15244        * unmodular tests elsewhere.
15245        */
15246       if ((count = tp->tty_outleft) == 0 || tp->tty_inhibited) return;
15247
15248       /* Copy the user bytes to buf[] for decent addressing. Loop over the
15249        * copies, since the user buffer may be much larger than buf[].
15250        */
15251       do {
15252             if (count > sizeof(buf)) count = sizeof(buf);
15253             user_phys = proc_vir2phys(proc_addr(tp->tty_outproc), tp->tty_out_vir);
15254             phys_copy(user_phys, vir2phys(buf), (phys_bytes) count);
15255             tbuf = buf;
15256
15257             /* Update terminal data structure. */
15258             tp->tty_out_vir += count;
15259             tp->tty_outcum += count;
15260             tp->tty_outleft -= count;
15261
15262             /* Output each byte of the copy to the screen.  Avoid calling
15263              * out_char() for the "easy" characters, put them into the buffer
15264              * directly.
15265              */
15266             do {
15267                     if ((unsigned) *tbuf < ' ' || cons->c_esc_state > 0
15268                             || cons->c_column >= scr_width
15269                             || cons->c_rwords >= buflen(cons->c_ramqueue))
15270                     {
15271                             out_char(cons, *tbuf++);
15272                     } else {
15273                             cons->c_ramqueue[cons->c_rwords++] =
15274                                             cons->c_attr | (*tbuf++ & BYTE);
15275                             cons->c_column++;
15276                     }
15277             } while (--count != 0);
15278       } while ((count = tp->tty_outleft) != 0 && !tp->tty_inhibited);
15279
15280       flush(cons);                  /* transfer anything buffered to the screen */
15281
15282       /* Reply to the writer if all output is finished. */
15283       if (tp->tty_outleft == 0) {
15284             tty_reply(tp->tty_outrepcode, tp->tty_outcaller, tp->tty_outproc,
15285                                                     tp->tty_outcum);
15286             tp->tty_outcum = 0;
15287       }
15288   }


15291   /*===========================================================================*
15292    *                               cons_echo                                   *
15293    *===========================================================================*/
15294   PRIVATE void cons_echo(tp, c)
15295   register tty_t *tp;           /* pointer to tty struct */
15296   int c;                        /* character to be echoed */
15297   {
15298   /* Echo keyboard input (print & flush). */
15299     console_t *cons = tp->tty_priv;
```

```
15300
15301     out_char(cons, c);
15302     flush(cons);
15303   }


15306   /*==========================================================================*
15307    *                              out_char                                    *
15308    *==========================================================================*/
15309   PRIVATE void out_char(cons, c)
15310   register console_t *cons;       /* pointer to console struct */
15311   int c;                          /* character to be output */
15312   {
15313   /* Output a character on the console.  Check for escape sequences first. */
15314     if (cons->c_esc_state > 0) {
15315           parse_escape(cons, c);
15316           return;
15317     }
15318
15319     switch(c) {
15320           case 000:               /* null is typically used for padding */
15321                   return;         /* better not do anything */
15322
15323           case 007:               /* ring the bell */
15324                   flush(cons);    /* print any chars queued for output */
15325                   beep();
15326                   return;
15327
15328           case '\b':              /* backspace */
15329                   if (--cons->c_column < 0) {
15330                           if (--cons->c_row >= 0) cons->c_column += scr_width;
15331                   }
15332                   flush(cons);
15333                   return;
15334
15335           case '\n':              /* line feed */
15336                   if ((cons->c_tty->tty_termios.c_oflag & (OPOST|ONLCR))
15337                                                   == (OPOST|ONLCR)) {
15338                           cons->c_column = 0;
15339                   }
15340                   /*FALL THROUGH*/
15341           case 013:               /* CTRL-K */
15342           case 014:               /* CTRL-L */
15343                   if (cons->c_row == scr_lines-1) {
15344                           scroll_screen(cons, SCROLL_UP);
15345                   } else {
15346                           cons->c_row++;
15347                   }
15348                   flush(cons);
15349                   return;
15350
15351           case '\r':              /* carriage return */
15352                   cons->c_column = 0;
15353                   flush(cons);
15354                   return;
15355
15356           case '\t':              /* tab */
15357                   cons->c_column = (cons->c_column + TAB_SIZE) & ~TAB_MASK;
15358                   if (cons->c_column > scr_width) {
15359                           cons->c_column -= scr_width;
```

```
15360                           if (cons->c_row == scr_lines-1) {
15361                                   scroll_screen(cons, SCROLL_UP);
15362                           } else {
15363                                   cons->c_row++;
15364                           }
15365                   }
15366                   flush(cons);
15367                   return;
15368
15369           case 033:               /* ESC - start of an escape sequence */
15370                   flush(cons);    /* print any chars queued for output */
15371                   cons->c_esc_state = 1;  /* mark ESC as seen */
15372                   return;
15373
15374           default:                /* printable chars are stored in ramqueue */
15375                   if (cons->c_column >= scr_width) {
15376                           if (!LINEWRAP) return;
15377                           if (cons->c_row == scr_lines-1) {
15378                                   scroll_screen(cons, SCROLL_UP);
15379                           } else {
15380                                   cons->c_row++;
15381                           }
15382                           cons->c_column = 0;
15383                           flush(cons);
15384                   }
15385                   if (cons->c_rwords == buflen(cons->c_ramqueue)) flush(cons);
15386                   cons->c_ramqueue[cons->c_rwords++] = cons->c_attr | (c & BYTE);
15387                   cons->c_column++;                       /* next column */
15388                   return;
15389     }
15390   }


15393   /*===========================================================================*
15394    *                               scroll_screen                               *
15395    *===========================================================================*/
15396   PRIVATE void scroll_screen(cons, dir)
15397   register console_t *cons;       /* pointer to console struct */
15398   int dir;                        /* SCROLL_UP or SCROLL_DOWN */
15399   {
15400     unsigned new_line, new_org, chars;
15401
15402     flush(cons);
15403     chars = scr_size - scr_width;         /* one screen minus one line */
15404
15405     /* Scrolling the screen is a real nuisance due to the various incompatible
15406      * video cards.  This driver supports software scrolling (Hercules?),
15407      * hardware scrolling (mono and CGA cards) and hardware scrolling without
15408      * wrapping (EGA cards).  In the latter case we must make sure that
15409      *            c_start <= c_org && c_org + scr_size <= c_limit
15410      * holds, because EGA doesn't wrap around the end of video memory.
15411      */
15412     if (dir == SCROLL_UP) {
15413           /* Scroll one line up in 3 ways: soft, avoid wrap, use origin. */
15414           if (softscroll) {
15415                   vid_vid_copy(cons->c_start + scr_width, cons->c_start, chars);
15416           } else
15417           if (!wrap && cons->c_org + scr_size + scr_width >= cons->c_limit) {
15418                   vid_vid_copy(cons->c_org + scr_width, cons->c_start, chars);
15419                   cons->c_org = cons->c_start;
```

```
15420           } else {
15421                   cons->c_org = (cons->c_org + scr_width) & vid_mask;
15422           }
15423           new_line = (cons->c_org + chars) & vid_mask;
15424     } else {
15425           /* Scroll one line down in 3 ways: soft, avoid wrap, use origin. */
15426           if (softscroll) {
15427                   vid_vid_copy(cons->c_start, cons->c_start + scr_width, chars);
15428           } else
15429           if (!wrap && cons->c_org < cons->c_start + scr_width) {
15430                   new_org = cons->c_limit - scr_size;
15431                   vid_vid_copy(cons->c_org, new_org + scr_width, chars);
15432                   cons->c_org = new_org;
15433           } else {
15434                   cons->c_org = (cons->c_org - scr_width) & vid_mask;
15435           }
15436           new_line = cons->c_org;
15437     }
15438     /* Blank the new line at top or bottom. */
15439     blank_color = cons->c_blank;
15440     mem_vid_copy(BLANK_MEM, new_line, scr_width);
15441
15442     /* Set the new video origin. */
15443     if (cons == curcons) set_6845(VID_ORG, cons->c_org);
15444     flush(cons);
15445   }


15448   /*===========================================================================*
15449    *                              flush                                        *
15450    *===========================================================================*/
15451   PRIVATE void flush(cons)
15452   register console_t *cons;       /* pointer to console struct */
15453   {
15454   /* Send characters buffered in 'ramqueue' to screen memory, check the new
15455    * cursor position, compute the new hardware cursor position and set it.
15456    */
15457     unsigned cur;
15458     tty_t *tp = cons->c_tty;
15459
15460     /* Have the characters in 'ramqueue' transferred to the screen. */
15461     if (cons->c_rwords > 0) {
15462           mem_vid_copy(cons->c_ramqueue, cons->c_cur, cons->c_rwords);
15463           cons->c_rwords = 0;
15464
15465           /* TTY likes to know the current column and if echoing messed up. */
15466           tp->tty_position = cons->c_column;
15467           tp->tty_reprint = TRUE;
15468     }
15469
15470     /* Check and update the cursor position. */
15471     if (cons->c_column < 0) cons->c_column = 0;
15472     if (cons->c_column > scr_width) cons->c_column = scr_width;
15473     if (cons->c_row < 0) cons->c_row = 0;
15474     if (cons->c_row >= scr_lines) cons->c_row = scr_lines - 1;
15475     cur = cons->c_org + cons->c_row * scr_width + cons->c_column;
15476     if (cur != cons->c_cur) {
15477           if (cons == curcons) set_6845(CURSOR, cur);
15478           cons->c_cur = cur;
15479     }
```

```
15480   }


15483   /*===========================================================================*
15484    *                              parse_escape                                 *
15485    *===========================================================================*/
15486   PRIVATE void parse_escape(cons, c)
15487   register console_t *cons;       /* pointer to console struct */
15488   char c;                         /* next character in escape sequence */
15489   {
15490   /* The following ANSI escape sequences are currently supported.
15491    * If n and/or m are omitted, they default to 1.
15492    *   ESC [nA moves up n lines
15493    *   ESC [nB moves down n lines
15494    *   ESC [nC moves right n spaces
15495    *   ESC [nD moves left n spaces
15496    *   ESC [m;nH" moves cursor to (m,n)
15497    *   ESC [J clears screen from cursor
15498    *   ESC [K clears line from cursor
15499    *   ESC [nL inserts n lines ar cursor
15500    *   ESC [nM deletes n lines at cursor
15501    *   ESC [nP deletes n chars at cursor
15502    *   ESC [n@ inserts n chars at cursor
15503    *   ESC [nm enables rendition n (0=normal, 4=bold, 5=blinking, 7=reverse)
15504    *   ESC M scrolls the screen backwards if the cursor is on the top line
15505    */
15506
15507     switch (cons->c_esc_state) {
15508       case 1:                       /* ESC seen */
15509           cons->c_esc_intro = '\0';
15510           cons->c_esc_parmp = cons->c_esc_parmv;
15511           cons->c_esc_parmv[0] = cons->c_esc_parmv[1] = 0;
15512           switch (c) {
15513               case '[':   /* Control Sequence Introducer */
15514                   cons->c_esc_intro = c;
15515                   cons->c_esc_state = 2;
15516                   break;
15517               case 'M':   /* Reverse Index */
15518                   do_escape(cons, c);
15519                   break;
15520               default:
15521                   cons->c_esc_state = 0;
15522           }
15523           break;
15524
15525       case 2:                       /* ESC [ seen */
15526           if (c >= '0' && c <= '9') {
15527                   if (cons->c_esc_parmp < bufend(cons->c_esc_parmv))
15528                           *cons->c_esc_parmp = *cons->c_esc_parmp * 10 + (c-'0');
15529           } else
15530           if (c == ';') {
15531                   if (++cons->c_esc_parmp < bufend(cons->c_esc_parmv))
15532                           *cons->c_esc_parmp = 0;
15533           } else {
15534                   do_escape(cons, c);
15535           }
15536           break;
15537     }
15538   }
```

```
15541   /*===========================================================================*
15542    *                              do_escape                                    *
15543    *===========================================================================*/
15544   PRIVATE void do_escape(cons, c)
15545   register console_t *cons;       /* pointer to console struct */
15546   char c;                         /* next character in escape sequence */
15547   {
15548     int value, n;
15549     unsigned src, dst, count;
15550
15551     /* Some of these things hack on screen RAM, so it had better be up to date */
15552     flush(cons);
15553
15554     if (cons->c_esc_intro == '\0') {
15555           /* Handle a sequence beginning with just ESC */
15556           switch (c) {
15557               case 'M':           /* Reverse Index */
15558                   if (cons->c_row == 0) {
15559                           scroll_screen(cons, SCROLL_DOWN);
15560                   } else {
15561                           cons->c_row--;
15562                   }
15563                   flush(cons);
15564                   break;
15565
15566               default: break;
15567           }
15568     } else
15569     if (cons->c_esc_intro == '[') {
15570           /* Handle a sequence beginning with ESC [ and parameters */
15571           value = cons->c_esc_parmv[0];
15572           switch (c) {
15573               case 'A':           /* ESC [nA moves up n lines */
15574                   n = (value == 0 ? 1 : value);
15575                   cons->c_row -= n;
15576                   flush(cons);
15577                   break;
15578
15579               case 'B':           /* ESC [nB moves down n lines */
15580                   n = (value == 0 ? 1 : value);
15581                   cons->c_row += n;
15582                   flush(cons);
15583                   break;
15584
15585               case 'C':           /* ESC [nC moves right n spaces */
15586                   n = (value == 0 ? 1 : value);
15587                   cons->c_column += n;
15588                   flush(cons);
15589                   break;
15590
15591               case 'D':           /* ESC [nD moves left n spaces */
15592                   n = (value == 0 ? 1 : value);
15593                   cons->c_column -= n;
15594                   flush(cons);
15595                   break;
15596
15597               case 'H':           /* ESC [m;nH" moves cursor to (m,n) */
15598                   cons->c_row = cons->c_esc_parmv[0] - 1;
15599                   cons->c_column = cons->c_esc_parmv[1] - 1;
```

```
15600                   flush(cons);
15601                   break;
15602
15603               case 'J':           /* ESC [sJ clears in display */
15604                   switch (value) {
15605                       case 0:     /* Clear from cursor to end of screen */
15606                           count = scr_size - (cons->c_cur - cons->c_org);
15607                           dst = cons->c_cur;
15608                           break;
15609                       case 1:     /* Clear from start of screen to cursor */
15610                           count = cons->c_cur - cons->c_org;
15611                           dst = cons->c_org;
15612                           break;
15613                       case 2:     /* Clear entire screen */
15614                           count = scr_size;
15615                           dst = cons->c_org;
15616                           break;
15617                       default:    /* Do nothing */
15618                           count = 0;
15619                           dst = cons->c_org;
15620                   }
15621                   blank_color = cons->c_blank;
15622                   mem_vid_copy(BLANK_MEM, dst, count);
15623                   break;
15624
15625               case 'K':           /* ESC [sK clears line from cursor */
15626                   switch (value) {
15627                       case 0:     /* Clear from cursor to end of line */
15628                           count = scr_width - cons->c_column;
15629                           dst = cons->c_cur;
15630                           break;
15631                       case 1:     /* Clear from beginning of line to cursor */
15632                           count = cons->c_column;
15633                           dst = cons->c_cur - cons->c_column;
15634                           break;
15635                       case 2:     /* Clear entire line */
15636                           count = scr_width;
15637                           dst = cons->c_cur - cons->c_column;
15638                           break;
15639                       default:    /* Do nothing */
15640                           count = 0;
15641                           dst = cons->c_cur;
15642                   }
15643                   blank_color = cons->c_blank;
15644                   mem_vid_copy(BLANK_MEM, dst, count);
15645                   break;
15646
15647               case 'L':           /* ESC [nL inserts n lines at cursor */
15648                   n = value;
15649                   if (n < 1) n = 1;
15650                   if (n > (scr_lines - cons->c_row))
15651                           n = scr_lines - cons->c_row;
15652
15653                   src = cons->c_org + cons->c_row * scr_width;
15654                   dst = src + n * scr_width;
15655                   count = (scr_lines - cons->c_row - n) * scr_width;
15656                   vid_vid_copy(src, dst, count);
15657                   blank_color = cons->c_blank;
15658                   mem_vid_copy(BLANK_MEM, src, n * scr_width);
15659                   break;
```

```
15660
15661               case 'M':           /* ESC [nM deletes n lines at cursor */
15662                   n = value;
15663                   if (n < 1) n = 1;
15664                   if (n > (scr_lines - cons->c_row))
15665                           n = scr_lines - cons->c_row;
15666
15667                   dst = cons->c_org + cons->c_row * scr_width;
15668                   src = dst + n * scr_width;
15669                   count = (scr_lines - cons->c_row - n) * scr_width;
15670                   vid_vid_copy(src, dst, count);
15671                   blank_color = cons->c_blank;
15672                   mem_vid_copy(BLANK_MEM, dst + count, n * scr_width);
15673                   break;
15674
15675               case '@':           /* ESC [n@ inserts n chars at cursor */
15676                   n = value;
15677                   if (n < 1) n = 1;
15678                   if (n > (scr_width - cons->c_column))
15679                           n = scr_width - cons->c_column;
15680
15681                   src = cons->c_cur;
15682                   dst = src + n;
15683                   count = scr_width - cons->c_column - n;
15684                   vid_vid_copy(src, dst, count);
15685                   blank_color = cons->c_blank;
15686                   mem_vid_copy(BLANK_MEM, src, n);
15687                   break;
15688
15689               case 'P':           /* ESC [nP deletes n chars at cursor */
15690                   n = value;
15691                   if (n < 1) n = 1;
15692                   if (n > (scr_width - cons->c_column))
15693                           n = scr_width - cons->c_column;
15694
15695                   dst = cons->c_cur;
15696                   src = dst + n;
15697                   count = scr_width - cons->c_column - n;
15698                   vid_vid_copy(src, dst, count);
15699                   blank_color = cons->c_blank;
15700                   mem_vid_copy(BLANK_MEM, dst + count, n);
15701                   break;
15702
15703               case 'm':           /* ESC [nm enables rendition n */
15704                   switch (value) {
15705                       case 1:     /* BOLD  */
15706                           if (color) {
15707                                   /* Can't do bold, so use yellow */
15708                                   cons->c_attr = (cons->c_attr & 0xf0ff) | 0x0E00;
15709                           } else {
15710                                   /* Set intensity bit */
15711                                   cons->c_attr |= 0x0800;
15712                           }
15713                           break;
15714
15715                       case 4:     /* UNDERLINE */
15716                           if (color) {
15717                                   /* Use light green */
15718                                   cons->c_attr = (cons->c_attr & 0xf0ff) | 0x0A00;
15719                           } else {
```

```
15720                                   cons->c_attr = (cons->c_attr & 0x8900);
15721                               }
15722                               break;
15723
15724                           case 5:     /* BLINKING */
15725                               if (color) {
15726                                       /* Use magenta */
15727                                       cons->c_attr = (cons->c_attr & 0xf0ff) | 0x0500;
15728                               } else {
15729                                       /* Set the blink bit */
15730                                       cons->c_attr |= 0x8000;
15731                               }
15732                               break;
15733
15734                           case 7:     /* REVERSE */
15735                               if (color) {
15736                                       /* Swap fg and bg colors */
15737                                       cons->c_attr =
15738                                               ((cons->c_attr & 0xf000) >> 4) |
15739                                               ((cons->c_attr & 0x0f00) << 4);
15740                               } else
15741                               if ((cons->c_attr & 0x7000) == 0) {
15742                                       cons->c_attr = (cons->c_attr & 0x8800) | 0x7000;
15743                               } else {
15744                                       cons->c_attr = (cons->c_attr & 0x8800) | 0x0700;
15745                               }
15746                               break;
15747
15748                           default:    /* COLOR */
15749                               if (30 <= value && value <= 37) {
15750                                       cons->c_attr =
15751                                               (cons->c_attr & 0xf0ff) |
15752                                               (ansi_colors[(value - 30)] << 8);
15753                                       cons->c_blank =
15754                                               (cons->c_blank & 0xf0ff) |
15755                                               (ansi_colors[(value - 30)] << 8);
15756                               } else
15757                               if (40 <= value && value <= 47) {
15758                                       cons->c_attr =
15759                                               (cons->c_attr & 0x0fff) |
15760                                               (ansi_colors[(value - 40)] << 12);
15761                                       cons->c_blank =
15762                                               (cons->c_blank & 0x0fff) |
15763                                               (ansi_colors[(value - 40)] << 12);
15764                               } else {
15765                                       cons->c_attr = cons->c_blank;
15766                               }
15767                               break;
15768                   }
15769                   break;
15770           }
15771       }
15772       cons->c_esc_state = 0;
15773   }


15776   /*==========================================================================*
15777    *                              set_6845                                    *
15778    *==========================================================================*/
15779   PRIVATE void set_6845(reg, val)
```

```
15780   int reg;                        /* which register pair to set */
15781   unsigned val;                   /* 16-bit value to set it to */
15782   {
15783   /* Set a register pair inside the 6845.
15784    * Registers 12-13 tell the 6845 where in video ram to start
15785    * Registers 14-15 tell the 6845 where to put the cursor
15786    */
15787     lock();                       /* try to stop h/w loading in-between value */
15788     out_byte(vid_port + INDEX, reg);              /* set the index register */
15789     out_byte(vid_port + DATA, (val>>8) & BYTE);   /* output high byte */
15790     out_byte(vid_port + INDEX, reg + 1);          /* again */
15791     out_byte(vid_port + DATA, val&BYTE);          /* output low byte */
15792     unlock();
15793   }


15796   /*==========================================================================*
15797    *                              beep                                        *
15798    *==========================================================================*/
15799   PRIVATE void beep()
15800   {
15801   /* Making a beeping sound on the speaker (output for CRTL-G).
15802    * This routine works by turning on the bits 0 and 1 in port B of the 8255
15803    * chip that drive the speaker.
15804    */
15805
15806     message mess;
15807
15808     if (beeping) return;
15809     out_byte(TIMER_MODE, 0xB6);   /* set up timer channel 2 (square wave) */
15810     out_byte(TIMER2, BEEP_FREQ & BYTE);   /* load low-order bits of frequency */
15811     out_byte(TIMER2, (BEEP_FREQ >> 8) & BYTE);    /* now high-order bits */
15812     lock();                       /* guard PORT_B from keyboard intr handler */
15813     out_byte(PORT_B, in_byte(PORT_B) | 3);        /* turn on beep bits */
15814     unlock();
15815     beeping = TRUE;
15816
15817     mess.m_type = SET_ALARM;
15818     mess.CLOCK_PROC_NR = TTY;
15819     mess.DELTA_TICKS = B_TIME;
15820     mess.FUNC_TO_CALL = (sighandler_t) stop_beep;
15821     sendrec(CLOCK, &mess);
15822   }


15825   /*==========================================================================*
15826    *                              stop_beep                                   *
15827    *==========================================================================*/
15828   PRIVATE void stop_beep()
15829   {
15830   /* Turn off the beeper by turning off bits 0 and 1 in PORT_B. */
15831
15832     lock();                       /* guard PORT_B from keyboard intr handler */
15833     out_byte(PORT_B, in_byte(PORT_B) & ~3);
15834     beeping = FALSE;
15835     unlock();
15836   }


15839   /*==========================================================================*
```

```
15840     *                               scr_init                                 *
15841     *============================================================================*/
15842    PUBLIC void scr_init(tp)
15843    tty_t *tp;
15844    {
15845    /* Initialize the screen driver. */
15846      console_t *cons;
15847      phys_bytes vid_base;
15848      u16_t bios_crtbase;
15849      int line;
15850      unsigned page_size;
15851
15852      /* Associate console and TTY. */
15853      line = tp - &tty_table[0];
15854      if (line >= nr_cons) return;
15855      cons = &cons_table[line];
15856      cons->c_tty = tp;
15857      tp->tty_priv = cons;
15858
15859      /* Initialize the keyboard driver. */
15860      kb_init(tp);
15861
15862      /* Output functions. */
15863      tp->tty_devwrite = cons_write;
15864      tp->tty_echo = cons_echo;
15865
15866      /* Get the BIOS parameters that tells the VDU I/O base register. */
15867      phys_copy(0x463L, vir2phys(&bios_crtbase), 2L);
15868
15869      vid_port = bios_crtbase;
15870
15871      if (color) {
15872            vid_base = COLOR_BASE;
15873            vid_size = COLOR_SIZE;
15874      } else {
15875            vid_base = MONO_BASE;
15876            vid_size = MONO_SIZE;
15877      }
15878      if (ega) vid_size = EGA_SIZE;
15879      wrap = !ega;
15880
15881      vid_seg = protected_mode ? VIDEO_SELECTOR : physb_to_hclick(vid_base);
15882      init_dataseg(&gdt[VIDEO_INDEX], vid_base, (phys_bytes) vid_size,
15883                                                          TASK_PRIVILEGE);
15884      vid_size >>= 1;               /* word count */
15885      vid_mask = vid_size - 1;
15886
15887      /* There can be as many consoles as video memory allows. */
15888      nr_cons = vid_size / scr_size;
15889      if (nr_cons > NR_CONS) nr_cons = NR_CONS;
15890      if (nr_cons > 1) wrap = 0;
15891      page_size = vid_size / nr_cons;
15892      cons->c_start = line * page_size;
15893      cons->c_limit = cons->c_start + page_size;
15894      cons->c_org = cons->c_start;
15895      cons->c_attr = cons->c_blank = BLANK_COLOR;
15896
15897      /* Clear the screen. */
15898      blank_color = BLANK_COLOR;
15899      mem_vid_copy(BLANK_MEM, cons->c_start, scr_size);
```

```
15900       select_console(0);
15901     }


15904     /*===========================================================================*
15905      *                              putk                                         *
15906      *===========================================================================*/
15907     PUBLIC void putk(c)
15908     int c;                        /* character to print */
15909     {
15910     /* This procedure is used by the version of printf() that is linked with
15911      * the kernel itself.  The one in the library sends a message to FS, which is
15912      * not what is needed for printing within the kernel.  This version just queues
15913      * the character and starts the output.
15914      */
15915
15916       if (c != 0) {
15917             if (c == '\n') putk('\r');
15918             out_char(&cons_table[0], (int) c);
15919       } else {
15920             flush(&cons_table[0]);
15921       }
15922     }


15925     /*===========================================================================*
15926      *                              toggle_scroll                                *
15927      *===========================================================================*/
15928     PUBLIC void toggle_scroll()
15929     {
15930     /* Toggle between hardware and software scroll. */
15931
15932       cons_org0();
15933       softscroll = !softscroll;
15934       printf("%sware scrolling enabled.\n", softscroll ? "Soft" : "Hard");
15935     }


15938     /*===========================================================================*
15939      *                              cons_stop                                    *
15940      *===========================================================================*/
15941     PUBLIC void cons_stop()
15942     {
15943     /* Prepare for halt or reboot. */
15944       cons_org0();
15945       softscroll = 1;
15946       select_console(0);
15947       cons_table[0].c_attr = cons_table[0].c_blank = BLANK_COLOR;
15948     }


15951     /*===========================================================================*
15952      *                              cons_org0                                    *
15953      *===========================================================================*/
15954     PRIVATE void cons_org0()
15955     {
15956     /* Scroll video memory back to put the origin at 0. */
15957       int cons_line;
15958       console_t *cons;
15959       unsigned n;
```

```
15960
15961     for (cons_line = 0; cons_line < nr_cons; cons_line++) {
15962           cons = &cons_table[cons_line];
15963           while (cons->c_org > cons->c_start) {
15964                   n = vid_size - scr_size;        /* amount of unused memory */
15965                   if (n > cons->c_org - cons->c_start)
15966                           n = cons->c_org - cons->c_start;
15967                   vid_vid_copy(cons->c_org, cons->c_org - n, scr_size);
15968                   cons->c_org -= n;
15969           }
15970           flush(cons);
15971     }
15972     select_console(current);
15973   }


15976   /*===========================================================================*
15977    *                               select_console                              *
15978    *===========================================================================*/
15979   PUBLIC void select_console(int cons_line)
15980   {
15981   /* Set the current console to console number 'cons_line'. */
15982
15983     if (cons_line < 0 || cons_line >= nr_cons) return;
15984     current = cons_line;
15985     curcons = &cons_table[cons_line];
15986     set_6845(VID_ORG, curcons->c_org);
15987     set_6845(CURSOR, curcons->c_cur);
15988   }


15991   /*===========================================================================*
15992    *                               con_loadfont                                *
15993    *===========================================================================*/
15994
15995   PUBLIC int con_loadfont(user_phys)
15996   phys_bytes user_phys;
15997   {
15998   /* Load a font into the EGA or VGA adapter. */
15999     static struct sequence seq1[7] = {
16000           { GA_SEQUENCER_INDEX, 0x00, 0x01 },
16001           { GA_SEQUENCER_INDEX, 0x02, 0x04 },
16002           { GA_SEQUENCER_INDEX, 0x04, 0x07 },
16003           { GA_SEQUENCER_INDEX, 0x00, 0x03 },
16004           { GA_GRAPHICS_INDEX, 0x04, 0x02 },
16005           { GA_GRAPHICS_INDEX, 0x05, 0x00 },
16006           { GA_GRAPHICS_INDEX, 0x06, 0x00 },
16007     };
16008     static struct sequence seq2[7] = {
16009           { GA_SEQUENCER_INDEX, 0x00, 0x01 },
16010           { GA_SEQUENCER_INDEX, 0x02, 0x03 },
16011           { GA_SEQUENCER_INDEX, 0x04, 0x03 },
16012           { GA_SEQUENCER_INDEX, 0x00, 0x03 },
16013           { GA_GRAPHICS_INDEX, 0x04, 0x00 },
16014           { GA_GRAPHICS_INDEX, 0x05, 0x10 },
16015           { GA_GRAPHICS_INDEX, 0x06,    0 },
16016     };
16017
16018     seq2[6].value= color ? 0x0E : 0x0A;
16019
```

```
16020     if (!ega) return(ENOTTY);
16021
16022     lock();
16023     ga_program(seq1);     /* bring font memory into view */
16024
16025     phys_copy(user_phys, (phys_bytes)GA_VIDEO_ADDRESS, (phys_bytes)GA_FONT_SIZE);
16026
16027     ga_program(seq2);     /* restore */
16028     unlock();
16029
16030     return(OK);
16031   }


16034   /*==========================================================================*
16035    *                              ga_program                                  *
16036    *==========================================================================*/
16037
16038   PRIVATE void ga_program(seq)
16039   struct sequence *seq;
16040   {
16041     int len= 7;
16042     do {
16043           out_byte(seq->index, seq->port);
16044           out_byte(seq->index+1, seq->value);
16045           seq++;
16046     } while (--len > 0);
16047   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                       src/kernel/dmp.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

16100   /* This file contains some dumping routines for debugging. */
16101
16102   #include "kernel.h"
16103   #include <minix/com.h>
16104   #include "proc.h"
16105
16106   char *vargv;
16107
16108   FORWARD _PROTOTYPE(char *proc_name, (int proc_nr));
16109
16110   /*==========================================================================*
16111    *                              p_dmp                                       *
16112    *==========================================================================*/
16113   #if (CHIP == INTEL)
16114   PUBLIC void p_dmp()
16115   {
16116   /* Proc table dump */
16117
16118     register struct proc *rp;
16119     static struct proc *oldrp = BEG_PROC_ADDR;
16120     int n = 0;
16121     phys_clicks text, data, size;
16122     int proc_nr;
16123
16124     printf("\n--pid --pc- ---sp- flag -user --sys-- -text- -data- -size- -recv- command\n");
```

```
16125
16126     for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
16127           proc_nr = proc_number(rp);
16128           if (rp->p_flags & P_SLOT_FREE) continue;
16129           if (++n > 20) break;
16130           text = rp->p_map[T].mem_phys;
16131           data = rp->p_map[D].mem_phys;
16132           size = rp->p_map[T].mem_len
16133                   + ((rp->p_map[S].mem_phys + rp->p_map[S].mem_len) - data);
16134           printf("%5d %5lx %6lx %2x %7U %7U %5uK %5uK %5uK ",
16135                  proc_nr < 0 ? proc_nr : rp->p_pid,
16136                  (unsigned long) rp->p_reg.pc,
16137                  (unsigned long) rp->p_reg.sp,
16138                  rp->p_flags,
16139                  rp->user_time, rp->sys_time,
16140                  click_to_round_k(text), click_to_round_k(data),
16141                  click_to_round_k(size));
16142           if (rp->p_flags & RECEIVING) {
16143                   printf("%-7.7s", proc_name(rp->p_getfrom));
16144           } else
16145           if (rp->p_flags & SENDING) {
16146                   printf("S:%-5.5s", proc_name(rp->p_sendto));
16147           } else
16148           if (rp->p_flags == 0) {
16149                   printf("       ");
16150           }
16151           printf("%s\n", rp->p_name);
16152     }
16153     if (rp == END_PROC_ADDR) rp = BEG_PROC_ADDR; else printf("--more--\r");
16154     oldrp = rp;
16155   }
16156   #endif                          /* (CHIP == INTEL) */
16157
16158   /*==========================================================================*
16159    *                              map_dmp                                     *
16160    *==========================================================================*/
16161   #if (SHADOWING == 0)
16162   PUBLIC void map_dmp()
16163   {
16164     register struct proc *rp;
16165     static struct proc *oldrp = cproc_addr(HARDWARE);
16166     int n = 0;
16167     phys_clicks size;
16168
16169     printf("\nPROC NAME-  -----TEXT-----  -----DATA-----  ----STACK-----  -SIZE-\n");
16170     for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
16171           if (rp->p_flags & P_SLOT_FREE) continue;
16172           if (++n > 20) break;
16173           size = rp->p_map[T].mem_len
16174                   + ((rp->p_map[S].mem_phys + rp->p_map[S].mem_len)
16175                                                 - rp->p_map[D].mem_phys);
16176           printf("%3d %-6.6s  %4x %4x %4x  %4x %4x %4x  %4x %4x %4x  %5uK\n",
16177                  proc_number(rp),
16178                  rp->p_name,
16179                  rp->p_map[T].mem_vir, rp->p_map[T].mem_phys, rp->p_map[T].mem_len,
16180                  rp->p_map[D].mem_vir, rp->p_map[D].mem_phys, rp->p_map[D].mem_len,
16181                  rp->p_map[S].mem_vir, rp->p_map[S].mem_phys, rp->p_map[S].mem_len,
16182                  click_to_round_k(size));
16183     }
16184     if (rp == END_PROC_ADDR) rp = cproc_addr(HARDWARE); else printf("--more--\r");
```

```
16185       oldrp = rp;
16186     }

16188     #else
16189
16190     PUBLIC void map_dmp()
16191     {
16192       register struct proc *rp;
16193       static struct proc *oldrp = cproc_addr(HARDWARE);
16194       int n = 0;
16195       vir_clicks base, limit;
16196
16197       printf("\nPROC NAME-  --TEXT---  --DATA---  --STACK-- SHADOW FLIP P BASE  SIZE\n");
16198       for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
16199             if (rp->p_flags & P_SLOT_FREE) continue;
16200             if (++n > 20) break;
16201             base = rp->p_map[T].mem_phys;
16202             limit = rp->p_map[S].mem_phys + rp->p_map[S].mem_len;
16203             printf("%3d %-6.6s  %4x %4x  %4x %4x  %4x %4x   %4x %4d %d %4uK\n",
16204                    proc_number(rp),
16205                    rp->p_name,
16206                    rp->p_map[T].mem_phys, rp->p_map[T].mem_len,
16207                    rp->p_map[D].mem_phys, rp->p_map[D].mem_len,
16208                    rp->p_map[S].mem_phys, rp->p_map[S].mem_len,
16209                    rp->p_shadow, rp->p_nflips, rp->p_physio,
16210                    click_to_round_k(base), click_to_round_k(limit));
16211       }
16212       if (rp == END_PROC_ADDR) rp = cproc_addr(HARDWARE); else printf("--more--\r");
16213       oldrp = rp;
16214     }

16216     #endif
16217
16218     #if (CHIP == M68000)
16219     FORWARD _PROTOTYPE(void mem_dmp, (char *adr, int len));
16220
16221     /*==========================================================================*
16222      *                                p_dmp                                     *
16223      *==========================================================================*/
16224     PUBLIC void p_dmp()
16225     {
16226     /* Proc table dump */
16227
16228       register struct proc *rp;
16229       static struct proc *oldrp = BEG_PROC_ADDR;
16230       int n = 0;
16231       vir_clicks base, limit;
16232
16233       printf(
16234              "\nproc pid     pc     sp  splow flag  user    sys   recv   command\n");
16235
16236       for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
16237             if (rp->p_flags & P_SLOT_FREE) continue;
16238             if (++n > 20) break;
16239             base = rp->p_map[T].mem_phys;
16240             limit = rp->p_map[S].mem_phys + rp->p_map[S].mem_len;
16241             printf("%4u %4u %6lx %6lx %6lx %4x %5U %6U   ",
16242                    proc_number(rp),
16243                    rp->p_pid,
16244                    (unsigned long) rp->p_reg.pc,
```

```
16245                   (unsigned long) rp->p_reg.sp,
16246                   (unsigned long) rp->p_splow,
16247                   rp->p_flags,
16248                   rp->user_time, rp->sys_time);
16249           if (rp->p_flags & RECEIVING) {
16250                   printf("%-7.7s", proc_name(rp->p_getfrom));
16251           } else
16252           if (rp->p_flags & SENDING) {
16253                   printf("S:%-5.5s", proc_name(rp->p_sendto));
16254           } else
16255           if (rp->p_flags == 0) {
16256                   printf("       ");
16257           }
16258           printf("%s\n", rp->p_name);
16259     }
16260     if (rp == END_PROC_ADDR) rp = BEG_PROC_ADDR; else printf("--more--\r");
16261     oldrp = rp;
16262   }


16265   /*==========================================================================*
16266    *                              reg_dmp                                     *
16267    *==========================================================================*/
16268   PUBLIC void reg_dmp(rp)
16269   struct proc *rp;
16270   {
16271     register int i;
16272     static char *regs[NR_REGS] = {
16273                       "d0", "d1", "d2", "d3", "d4", "d5", "d6", "d7",
16274                       "a0", "a1", "a2", "a3", "a4", "a5", "a6"
16275     };
16276     reg_t *regptr = (reg_t *) & rp->p_reg;
16277
16278     printf("reg = %08lx, ", rp);
16279     printf("ksp = %08lx\n", (long) &rp + sizeof(rp));
16280     printf(" pc = %08lx, ", rp->p_reg.pc);
16281     printf(" sr =     %04x, ", rp->p_reg.psw);
16282     printf("trp =       %2x\n", rp->p_trap);
16283     for (i = 0; i < NR_REGS; i++)
16284           printf("%3s = %08lx%s",regs[i], *regptr++, (i&3) == 3 ? "\n" : ", ");
16285     printf(" a7 = %08lx\n", rp->p_reg.sp);
16286   #if (SHADOWING == 1)
16287       mem_dmp((char *) (((long) rp->p_reg.pc & ~31L) - 96), 128);
16288       mem_dmp((char *) (((long) rp->p_reg.sp & ~31L) - 32), 256);
16289   #else
16290       mem_dmp((char *) (((long) rp->p_reg.pc & ~31L) - 96 +
16291                         ((long)rp->p_map[T].mem_phys<<CLICK_SHIFT)), 128);
16292       mem_dmp((char *) (((long) rp->p_reg.sp & ~31L) - 32 +
16293                         ((long)rp->p_map[S].mem_phys<<CLICK_SHIFT)), 256);
16294   #endif
16295   }


16298   /*==========================================================================*
16299    *                              mem_dmp                                     *
16300    *==========================================================================*/
16301   PRIVATE void mem_dmp(adr, len)
16302   char *adr;
16303   int len;
16304   {
```

```
16305     register i;
16306     register long *p;
16307
16308     for (i = 0, p = (long *) adr; i < len; i += 4) {
16309   #if (CHIP == M68000)
16310           if ((i & 31) == 0) printf("\n%lX:", p);
16311           printf(" %8lX", *p++);
16312   #else
16313           if ((i & 31) == 0) printf("\n%X:", p);
16314           printf(" %8X", *p++);
16315   #endif /* (CHIP == M68000) */
16316     }
16317     printf("\n");
16318   }

16320   #endif                          /* (CHIP == M68000) */
16321
16322
16323   /*==========================================================================*
16324    *                                proc_name                                 *
16325    *==========================================================================*/
16326   PRIVATE char *proc_name(proc_nr)
16327   int proc_nr;
16328   {
16329     if (proc_nr == ANY) return "ANY";
16330     return proc_addr(proc_nr)->p_name;
16331   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/dp8390.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

16400   /*
16401    * dp8390.c
16402    *
16403    * This file contains a ethernet device driver for NS dp8390 based ethernet
16404    * cards.
16405    *
16406    * The valid messages and their parameters are:
16407    *
16408    *   m_type       DL_PORT    DL_PROC   DL_COUNT   DL_MODE   DL_ADDR
16409    * |------------+----------+---------+----------+---------+---------|
16410    * | HARDINT    |          |         |          |         |         |
16411    * |------------|----------|---------|----------|---------|---------|
16412    * | DL_WRITE   | port nr  | proc nr | count    | mode    | address |
16413    * |------------|----------|---------|----------|---------|---------|
16414    * | DL_WRITEV  | port nr  | proc nr | count    | mode    | address |
16415    * |------------|----------|---------|----------|---------|---------|
16416    * | DL_READ    | port nr  | proc nr | count    |         | address |
16417    * |------------|----------|---------|----------|---------|---------|
16418    * | DL_READV   | port nr  | proc nr | count    |         | address |
16419    * |------------|----------|---------|----------|---------|---------|
16420    * | DL_INIT    | port nr  | proc nr | mode     |         | address |
16421    * |------------|----------|---------|----------|---------|---------|
16422    * | DL_GETSTAT | port nr  | proc nr |          |         | address |
16423    * |------------|----------|---------|----------|---------|---------|
16424    * | DL_STOP    | port_nr  |         |          |         |         |
```

```
16425     * |------------|----------|---------|----------|---------|---------|
16426     *
16427     * The messages sent are:
16428     *
16429     *   m-type       DL_POR T   DL_PROC   DL_COUNT   DL_STAT   DL_CLCK
16430     * |------------|----------|---------|----------|---------|---------|
16431     * |DL_TASK_REPL| port nr  | proc nr | rd-count | err|stat| clock   |
16432     * |------------|----------|---------|----------|---------|---------|
16433     *
16434     *   m_type       m3_i1     m3_i2       m3_ca1
16435     * |------------+---------+-----------+---------------|
16436     * |DL_INIT_REPL| port nr | last port | ethernet addr |
16437     * |------------|---------|-----------|---------------|
16438     *
16439     * Created:     before Dec 28, 1992 by Philip Homburg <philip@cs.vu.nl>
16440     *
16441     *              Modified to become a generic dp8390 driver.
16442     *                      March 10, 1994, Philip Homburg
16443     */
16444
16445    #include "kernel.h"
16446    #include <stdlib.h>
16447    #include <minix/com.h>
16448    #include <net/hton.h>
16449    #include <net/gen/ether.h>
16450    #include <net/gen/eth_io.h>
16451    #include "assert.h"
16452    INIT_ASSERT
16453    #include "protect.h"
16454    #include "dp8390.h"
16455    #include "proc.h"
16456
16457    #if ENABLE_NETWORKING
16458
16459    #if !__minix_vmd
16460    #define printW()        (void) 0
16461    #define debug           0
16462    #endif
16463
16464    #define DE_PORT_NR      2
16465
16466    static dpeth_t de_table[DE_PORT_NR];
16467    static int int_pending[NR_IRQ_VECTORS];
16468    static int dpeth_tasknr= ANY;
16469    static u16_t eth_ign_proto;
16470
16471    /* Configuration */
16472    typedef struct dp_conf
16473    {
16474            port_t dpc_port;
16475            int dpc_irq;
16476            phys_bytes dpc_mem;
16477            char *dpc_envvar;
16478            segm_t dpc_prot_sel;
16479    } dp_conf_t;
16480
16481    dp_conf_t dp_conf[]=    /* Card addresses */
16482    {
16483            /* I/O port, IRQ,  Buffer address,  Env. var,   Buf selector. */
16484            {  0x280,     3,    0xD0000,        "DPETH0",  DP_ETH0_SELECTOR },
```

```
16485           {  0x300,     5,    0xCC000,        "DPETH1",  DP_ETH1_SELECTOR },
16486   };
16487
16488   /* Test if dp_conf has exactly DE_PORT_NR entries.  If not then you will see
16489    * the error: "array size is negative".
16490    */
16491   extern int ___dummy[DE_PORT_NR == sizeof(dp_conf)/sizeof(dp_conf[0]) ? 1 : -1];
16492
16493
16494   _PROTOTYPE( static void do_vwrite, (message *mp, int from_int,
16495                                                         int vectored)   );
16496   _PROTOTYPE( static void do_vread, (message *mp, int vectored)           );
16497   _PROTOTYPE( static void do_init, (message *mp)                          );
16498   _PROTOTYPE( static void do_int, (dpeth_t *dep)                          );
16499   _PROTOTYPE( static void do_getstat, (message *mp)                       );
16500   _PROTOTYPE( static void do_stop, (message *mp)                          );
16501   _PROTOTYPE( static void dp_init, (dpeth_t *dep)                         );
16502   _PROTOTYPE( static void dp_confaddr, (dpeth_t *dep)                     );
16503   _PROTOTYPE( static void dp_reinit, (dpeth_t *dep)                       );
16504   _PROTOTYPE( static void dp_reset, (dpeth_t *dep)                        );
16505   _PROTOTYPE( static void dp_check_ints, (dpeth_t *dep)                   );
16506   _PROTOTYPE( static void dp_recv, (dpeth_t *dep)                         );
16507   _PROTOTYPE( static void dp_send, (dpeth_t *dep)                         );
16508   _PROTOTYPE( static void dp_getblock, (dpeth_t *dep, int page,
16509                                   size_t offset, size_t size, void *dst)  );
16510   _PROTOTYPE( static void dp_pio8_getblock, (dpeth_t *dep, int page,
16511                                   size_t offset, size_t size, void *dst)  );
16512   _PROTOTYPE( static void dp_pio16_getblock, (dpeth_t *dep, int page,
16513                                   size_t offset, size_t size, void *dst)  );
16514   _PROTOTYPE( static int dp_pkt2user, (dpeth_t *dep, int page,
16515                                                         int length)     );
16516   _PROTOTYPE( static void dp_user2nic, (dpeth_t *dep, iovec_dat_t *iovp,
16517                   vir_bytes offset, int nic_addr, vir_bytes count)        );
16518   _PROTOTYPE( static void dp_pio8_user2nic, (dpeth_t *dep,
16519                                   iovec_dat_t *iovp, vir_bytes offset,
16520                                   int nic_addr, vir_bytes count)          );
16521   _PROTOTYPE( static void dp_pio16_user2nic, (dpeth_t *dep,
16522                                   iovec_dat_t *iovp, vir_bytes offset,
16523                                   int nic_addr, vir_bytes count)          );
16524   _PROTOTYPE( static void dp_nic2user, (dpeth_t *dep, int nic_addr,
16525                   iovec_dat_t *iovp, vir_bytes offset, vir_bytes count)   );
16526   _PROTOTYPE( static void dp_pio8_nic2user, (dpeth_t *dep, int nic_addr,
16527                   iovec_dat_t *iovp, vir_bytes offset, vir_bytes count)   );
16528   _PROTOTYPE( static void dp_pio16_nic2user, (dpeth_t *dep, int nic_addr,
16529                   iovec_dat_t *iovp, vir_bytes offset, vir_bytes count)   );
16530   _PROTOTYPE( static void dp_next_iovec, (iovec_dat_t *iovp)              );
16531   _PROTOTYPE( static int dp_handler, (int irq)                            );
16532   _PROTOTYPE( static void conf_hw, (dpeth_t *dep)                         );
16533   _PROTOTYPE( static void update_conf, (dpeth_t *dep, dp_conf_t *dcp)     );
16534   _PROTOTYPE( static int calc_iovec_size, (iovec_dat_t *iovp)             );
16535   _PROTOTYPE( static void reply, (dpeth_t *dep, int err, int may_block)   );
16536   _PROTOTYPE( static void mess_reply, (message *req, message *reply)      );
16537   _PROTOTYPE( static void get_userdata, (int user_proc,
16538                   vir_bytes user_addr, vir_bytes count, void *loc_addr)   );
16539   _PROTOTYPE( static void put_userdata, (int user_proc,
16540                   vir_bytes user_addr, vir_bytes count, void *loc_addr)   );
16541
16542   /*===========================================================================*
16543    *                              dpeth_task                                   *
16544    *===========================================================================*/
```

```
16545   void dp8390_task()
16546   {
16547           message m;
16548           int i, irq, r;
16549           dpeth_t *dep;
16550           long v;
16551
16552           dpeth_tasknr= proc_number(proc_ptr);
16553
16554           v= 0xFFFF;
16555           (void) env_parse("ETH_IGN_PROTO", "x", 0, &v, 0x0000L, 0xFFFFL);
16556           eth_ign_proto= htons((u16_t) v);
16557
16558           while (TRUE)
16559           {
16560                   if ((r= receive(ANY, &m)) != OK)
16561                           panic("dp8390: receive failed", r);
16562
16563                   switch (m.m_type)
16564                   {
16565                   case DL_WRITE:  do_vwrite(&m, FALSE, FALSE);    break;
16566                   case DL_WRITEV: do_vwrite(&m, FALSE, TRUE);     break;
16567                   case DL_READ:   do_vread(&m, FALSE);            break;
16568                   case DL_READV:  do_vread(&m, TRUE);             break;
16569                   case DL_INIT:   do_init(&m);                    break;
16570                   case DL_GETSTAT: do_getstat(&m);                break;
16571                   case DL_STOP:   do_stop(&m);                    break;
16572                   case HARD_INT:
16573                           for (i= 0, dep= &de_table[0]; i<DE_PORT_NR; i++, dep++)
16574                           {
16575                                   if (dep->de_mode != DEM_ENABLED)
16576                                           continue;
16577                                   assert(dep->de_flags & DEF_ENABLED);
16578                                   irq= dep->de_irq;
16579                                   assert(irq >= 0 && irq < NR_IRQ_VECTORS);
16580                                   if (int_pending[irq])
16581                                   {
16582                                           int_pending[irq]= 0;
16583                                           dp_check_ints(dep);
16584                                           do_int(dep);
16585                                   }
16586                           }
16587                           break;
16588                   default:
16589                           panic("dp8390: illegal message", m.m_type);
16590                   }
16591           }
16592   }


16595   /*==========================================================================*
16596    *                              dp_dump                                     *
16597    *==========================================================================*/
16598   void dp_dump()
16599   {
16600           dpeth_t *dep;
16601           int i, isr;
16602
16603           printf("\n");
16604           for (i= 0, dep = &de_table[0]; i<DE_PORT_NR; i++, dep++)
```

```
16605           {
16606                   if (dep->de_mode == DEM_DISABLED)
16607                           printf("dp8390 port %d is disabled\n", i);
16608                   else if (dep->de_mode == DEM_SINK)
16609                           printf("dp8390 port %d is in sink mode\n", i);
16610
16611                   if (dep->de_mode != DEM_ENABLED)
16612                           continue;
16613
16614                   printf("dp8390 statistics of port %d:\n", i);
16615
16616                   printf("recvErr    :%8ld\t", dep->de_stat.ets_recvErr);
16617                   printf("sendErr    :%8ld\t", dep->de_stat.ets_sendErr);
16618                   printf("OVW        :%8ld\n", dep->de_stat.ets_OVW);
16619
16620                   printf("CRCerr     :%8ld\t", dep->de_stat.ets_CRCerr);
16621                   printf("frameAll   :%8ld\t", dep->de_stat.ets_frameAll);
16622                   printf("missedP    :%8ld\n", dep->de_stat.ets_missedP);
16623
16624                   printf("packetR    :%8ld\t", dep->de_stat.ets_packetR);
16625                   printf("packetT    :%8ld\t", dep->de_stat.ets_packetT);
16626                   printf("transDef   :%8ld\n", dep->de_stat.ets_transDef);
16627
16628                   printf("collision  :%8ld\t", dep->de_stat.ets_collision);
16629                   printf("transAb    :%8ld\t", dep->de_stat.ets_transAb);
16630                   printf("carrSense  :%8ld\n", dep->de_stat.ets_carrSense);
16631
16632                   printf("fifoUnder  :%8ld\t", dep->de_stat.ets_fifoUnder);
16633                   printf("fifoOver   :%8ld\t", dep->de_stat.ets_fifoOver);
16634                   printf("CDheartbeat:%8ld\n", dep->de_stat.ets_CDheartbeat);
16635
16636                   printf("OWC        :%8ld\t", dep->de_stat.ets_OWC);
16637
16638                   isr= inb_reg0(dep, DP_ISR);
16639                   printf("dp_isr = 0x%x + 0x%x, de_flags = 0x%x\n", isr,
16640                                           inb_reg0(dep, DP_ISR), dep->de_flags);
16641           }
16642   }


16645   /*===========================================================================*
16646    *                              dp8390_stop                                  *
16647    *===========================================================================*/
16648   void dp8390_stop()
16649   {
16650           message mess;
16651           int i;
16652
16653           for (i= 0; i<DE_PORT_NR; i++)
16654           {
16655                   if (de_table[i].de_mode != DEM_ENABLED)
16656                           continue;
16657                   mess.m_type= DL_STOP;
16658                   mess.DL_PORT= i;
16659                   do_stop(&mess);
16660           }
16661   }


16664   /*===========================================================================*
```

```
16665    *                             do_vwrite                                        *
16666    *==============================================================================*/
16667   static void do_vwrite(mp, from_int, vectored)
16668   message *mp;
16669   int from_int;
16670   int vectored;
16671   {
16672           int port, count, size;
16673           int sendq_head;
16674           dpeth_t *dep;
16675
16676           port = mp->DL_PORT;
16677           count = mp->DL_COUNT;
16678           if (port < 0 || port >= DE_PORT_NR)
16679                   panic("dp8390: illegal port", port);
16680           dep= &de_table[port];
16681           dep->de_client= mp->DL_PROC;
16682
16683           if (dep->de_mode == DEM_SINK)
16684           {
16685                   assert(!from_int);
16686                   dep->de_flags |= DEF_PACK_SEND;
16687                   reply(dep, OK, FALSE);
16688                   return;
16689           }
16690           assert(dep->de_mode == DEM_ENABLED);
16691           assert(dep->de_flags & DEF_ENABLED);
16692           if (dep->de_flags & DEF_SEND_AVAIL)
16693                   panic("dp8390: send already in progress", NO_NUM);
16694
16695           sendq_head= dep->de_sendq_head;
16696           if (dep->de_sendq[sendq_head].sq_filled)
16697           {
16698                   if (from_int)
16699                           panic("dp8390: should not be sending\n", NO_NUM);
16700                   dep->de_sendmsg= *mp;
16701                   dep->de_flags |= DEF_SEND_AVAIL;
16702                   reply(dep, OK, FALSE);
16703                   return;
16704           }
16705           assert(!(dep->de_flags & DEF_PACK_SEND));
16706
16707           if (vectored)
16708           {
16709                   get_userdata(mp->DL_PROC, (vir_bytes) mp->DL_ADDR,
16710                           (count > IOVEC_NR ? IOVEC_NR : count) *
16711                           sizeof(iovec_t), dep->de_write_iovec.iod_iovec);
16712                   dep->de_write_iovec.iod_iovec_s = count;
16713                   dep->de_write_iovec.iod_proc_nr = mp->DL_PROC;
16714                   dep->de_write_iovec.iod_iovec_addr = (vir_bytes) mp->DL_ADDR;
16715
16716                   dep->de_tmp_iovec = dep->de_write_iovec;
16717                   size = calc_iovec_size(&dep->de_tmp_iovec);
16718           }
16719           else
16720           {
16721                   dep->de_write_iovec.iod_iovec[0].iov_addr =
16722                           (vir_bytes) mp->DL_ADDR;
16723                   dep->de_write_iovec.iod_iovec[0].iov_size =
16724                           mp->DL_COUNT;
```

```
16725                   dep->de_write_iovec.iod_iovec_s = 1;
16726                   dep->de_write_iovec.iod_proc_nr = mp->DL_PROC;
16727                   dep->de_write_iovec.iod_iovec_addr = 0;
16728                   size= mp->DL_COUNT;
16729           }
16730           if (size < ETH_MIN_PACK_SIZE || size > ETH_MAX_PACK_SIZE)
16731           {
16732                   panic("dp8390: invalid packet size", size);
16733           }
16734           (dep->de_user2nicf)(dep, &dep->de_write_iovec, 0,
16735                   dep->de_sendq[sendq_head].sq_sendpage * DP_PAGESIZE,
16736                   size);
16737           dep->de_sendq[sendq_head].sq_filled= TRUE;
16738           if (dep->de_sendq_tail == sendq_head)
16739           {
16740                   outb_reg0(dep, DP_TPSR, dep->de_sendq[sendq_head].sq_sendpage);
16741                   outb_reg0(dep, DP_TBCR1, size >> 8);
16742                   outb_reg0(dep, DP_TBCR0, size & 0xff);
16743                   outb_reg0(dep, DP_CR, CR_TXP);  /* there it goes.. */
16744           }
16745           else
16746                   dep->de_sendq[sendq_head].sq_size= size;
16747
16748           if (++sendq_head == dep->de_sendq_nr)
16749                   sendq_head= 0;
16750           assert(sendq_head < SENDQ_NR);
16751           dep->de_sendq_head= sendq_head;
16752
16753           dep->de_flags |= DEF_PACK_SEND;
16754
16755           /* If the interrupt handler called, don't send a reply. The reply
16756            * will be sent after all interrupts are handled.
16757            */
16758           if (from_int)
16759                   return;
16760           reply(dep, OK, FALSE);
16761
16762           assert(dep->de_mode == DEM_ENABLED);
16763           assert(dep->de_flags & DEF_ENABLED);
16764   }


16767   /*===========================================================================*
16768    *                               do_vread                                    *
16769    *===========================================================================*/
16770   static void do_vread(mp, vectored)
16771   message *mp;
16772   int vectored;
16773   {
16774           int port, count;
16775           int size;
16776           dpeth_t *dep;
16777
16778           port = mp->DL_PORT;
16779           count = mp->DL_COUNT;
16780           if (port < 0 || port >= DE_PORT_NR)
16781                   panic("dp8390: illegal port", port);
16782           dep= &de_table[port];
16783           dep->de_client= mp->DL_PROC;
16784           if (dep->de_mode == DEM_SINK)
```

```
16785           {
16786                   reply(dep, OK, FALSE);
16787                   return;
16788           }
16789           assert(dep->de_mode == DEM_ENABLED);
16790           assert(dep->de_flags & DEF_ENABLED);
16791
16792           if(dep->de_flags & DEF_READING)
16793                   panic("dp8390: read already in progress", NO_NUM);
16794
16795           if (vectored)
16796           {
16797                   get_userdata(mp->DL_PROC, (vir_bytes) mp->DL_ADDR,
16798                           (count > IOVEC_NR ? IOVEC_NR : count) *
16799                           sizeof(iovec_t), dep->de_read_iovec.iod_iovec);
16800                   dep->de_read_iovec.iod_iovec_s = count;
16801                   dep->de_read_iovec.iod_proc_nr = mp->DL_PROC;
16802                   dep->de_read_iovec.iod_iovec_addr = (vir_bytes) mp->DL_ADDR;
16803
16804                   dep->de_tmp_iovec = dep->de_read_iovec;
16805                   size= calc_iovec_size(&dep->de_tmp_iovec);
16806           }
16807           else
16808           {
16809                   dep->de_read_iovec.iod_iovec[0].iov_addr =
16810                           (vir_bytes) mp->DL_ADDR;
16811                   dep->de_read_iovec.iod_iovec[0].iov_size =
16812                           mp->DL_COUNT;
16813                   dep->de_read_iovec.iod_iovec_s = 1;
16814                   dep->de_read_iovec.iod_proc_nr = mp->DL_PROC;
16815                   dep->de_read_iovec.iod_iovec_addr = 0;
16816                   size= count;
16817           }
16818           if (size < ETH_MAX_PACK_SIZE)
16819                   panic("dp8390: wrong packet size", size);
16820           dep->de_flags |= DEF_READING;
16821
16822           dp_recv(dep);
16823
16824           if ((dep->de_flags & (DEF_READING|DEF_STOPPED)) ==
16825                   (DEF_READING|DEF_STOPPED))
16826           {
16827                   /* The chip is stopped, and all arrived packets are
16828                    * delivered.
16829                    */
16830                   dp_reset(dep);
16831           }
16832           reply(dep, OK, FALSE);
16833   }


16836   /*===========================================================================*
16837    *                              do_init                                      *
16838    *===========================================================================*/
16839   static void do_init(mp)
16840   message *mp;
16841   {
16842           int port;
16843           dpeth_t *dep;
16844           message reply_mess;
```

```
16845
16846           port = mp->DL_PORT;
16847           if (port < 0 || port >= DE_PORT_NR)
16848           {
16849                   reply_mess.m_type= DL_INIT_REPLY;
16850                   reply_mess.m3_i1= ENXIO;
16851                   mess_reply(mp, &reply_mess);
16852                   return;
16853           }
16854           dep= &de_table[port];
16855           if (dep->de_mode == DEM_DISABLED)
16856           {
16857                   /* This is the default, try to (re)locate the device. */
16858                   conf_hw(dep);
16859                   if (dep->de_mode == DEM_DISABLED)
16860                   {
16861                           /* Probe failed, or the device is configured off. */
16862                           reply_mess.m_type= DL_INIT_REPLY;
16863                           reply_mess.m3_i1= ENXIO;
16864                           mess_reply(mp, &reply_mess);
16865                           return;
16866                   }
16867                   if (dep->de_mode == DEM_ENABLED)
16868                           dp_init(dep);
16869           }
16870
16871           if (dep->de_mode == DEM_SINK)
16872           {
16873                   dep->de_address.ea_addr[0] =
16874                           dep->de_address.ea_addr[1] =
16875                           dep->de_address.ea_addr[2] =
16876                           dep->de_address.ea_addr[3] =
16877                           dep->de_address.ea_addr[4] =
16878                           dep->de_address.ea_addr[5] = 0;
16879                   dp_confaddr(dep);
16880                   reply_mess.m_type = DL_INIT_REPLY;
16881                   reply_mess.m3_i1 = mp->DL_PORT;
16882                   reply_mess.m3_i2 = DE_PORT_NR;
16883                   *(ether_addr_t *) reply_mess.m3_ca1 = dep->de_address;
16884                   mess_reply(mp, &reply_mess);
16885                   return;
16886           }
16887           assert(dep->de_mode == DEM_ENABLED);
16888           assert(dep->de_flags & DEF_ENABLED);
16889
16890           dep->de_flags &= ~(DEF_PROMISC | DEF_MULTI | DEF_BROAD);
16891
16892           if (mp->DL_MODE & DL_PROMISC_REQ)
16893                   dep->de_flags |= DEF_PROMISC | DEF_MULTI | DEF_BROAD;
16894           if (mp->DL_MODE & DL_MULTI_REQ)
16895                   dep->de_flags |= DEF_MULTI;
16896           if (mp->DL_MODE & DL_BROAD_REQ)
16897                   dep->de_flags |= DEF_BROAD;
16898
16899           dep->de_client = mp->m_source;
16900           dp_reinit(dep);
16901
16902           reply_mess.m_type = DL_INIT_REPLY;
16903           reply_mess.m3_i1 = mp->DL_PORT;
16904           reply_mess.m3_i2 = DE_PORT_NR;
```

```
16905           *(ether_addr_t *) reply_mess.m3_ca1 = dep->de_address;
16906
16907           mess_reply(mp, &reply_mess);
16908   }

16910   /*===========================================================================*
16911    *                              do_int                                       *
16912    *===========================================================================*/
16913   static void do_int(dep)
16914   dpeth_t *dep;
16915   {
16916           if (dep->de_flags & (DEF_PACK_SEND | DEF_PACK_RECV))
16917                   reply(dep, OK, TRUE);
16918   }


16921   /*===========================================================================*
16922    *                              do_getstat                                   *
16923    *===========================================================================*/
16924   static void do_getstat(mp)
16925   message *mp;
16926   {
16927           int port;
16928           dpeth_t *dep;
16929
16930           port = mp->DL_PORT;
16931           if (port < 0 || port >= DE_PORT_NR)
16932                   panic("dp8390: illegal port", port);
16933           dep= &de_table[port];
16934           dep->de_client= mp->DL_PROC;
16935           if (dep->de_mode == DEM_SINK)
16936           {
16937                   put_userdata(mp->DL_PROC, (vir_bytes) mp->DL_ADDR,
16938                           (vir_bytes) sizeof(dep->de_stat), &dep->de_stat);
16939                   reply(dep, OK, FALSE);
16940                   return;
16941           }
16942           assert(dep->de_mode == DEM_ENABLED);
16943           assert(dep->de_flags & DEF_ENABLED);
16944
16945           dep->de_stat.ets_CRCerr += inb_reg0(dep, DP_CNTR0);
16946           dep->de_stat.ets_frameAll += inb_reg0(dep, DP_CNTR1);
16947           dep->de_stat.ets_missedP += inb_reg0(dep, DP_CNTR2);
16948
16949           put_userdata(mp->DL_PROC, (vir_bytes) mp->DL_ADDR,
16950                   (vir_bytes) sizeof(dep->de_stat), &dep->de_stat);
16951           reply(dep, OK, FALSE);
16952   }


16955   /*===========================================================================*
16956    *                              do_stop                                      *
16957    *===========================================================================*/
16958   static void do_stop(mp)
16959   message *mp;
16960   {
16961           int port;
16962           dpeth_t *dep;
16963
16964           port = mp->DL_PORT;
```

```
16965
16966           if (port < 0 || port >= DE_PORT_NR)
16967                   panic("dp8390: illegal port", port);
16968           dep= &de_table[port];
16969           if (dep->de_mode == DEM_SINK)
16970                   return;
16971           assert(dep->de_mode == DEM_ENABLED);
16972
16973           if (!(dep->de_flags & DEF_ENABLED))
16974                   return;
16975
16976           outb_reg0(dep, DP_CR, CR_STP | CR_DM_ABORT);
16977           (dep->de_stopf)(dep);
16978
16979           dep->de_flags= DEF_EMPTY;
16980   }


16983   /*===========================================================================*
16984    *                              dp_init                                      *
16985    *===========================================================================*/
16986   static void dp_init(dep)
16987   dpeth_t *dep;
16988   {
16989           int dp_rcr_reg;
16990           int i;
16991
16992           /* General initialization */
16993           dep->de_flags = DEF_EMPTY;
16994           (*dep->de_initf)(dep);
16995
16996           dp_confaddr(dep);
16997
16998           /* Initialization of the dp8390 */
16999           outb_reg0(dep, DP_CR, CR_PS_P0 | CR_STP | CR_DM_ABORT);
17000           outb_reg0(dep, DP_IMR, 0);
17001           outb_reg0(dep, DP_PSTART, dep->de_startpage);
17002           outb_reg0(dep, DP_PSTOP, dep->de_stoppage);
17003           outb_reg0(dep, DP_BNRY, dep->de_startpage);
17004           outb_reg0(dep, DP_RCR, RCR_MON);
17005           outb_reg0(dep, DP_TCR, TCR_NORMAL);
17006           if (dep->de_16bit)
17007                   outb_reg0(dep, DP_DCR, DCR_WORDWIDE | DCR_8BYTES | DCR_BMS);
17008           else
17009                   outb_reg0(dep, DP_DCR, DCR_BYTEWIDE | DCR_8BYTES | DCR_BMS);
17010           outb_reg0(dep, DP_RBCR0, 0);
17011           outb_reg0(dep, DP_RBCR1, 0);
17012           outb_reg0(dep, DP_ISR, 0xFF);
17013           outb_reg0(dep, DP_CR, CR_PS_P1 | CR_DM_ABORT);
17014
17015           outb_reg1(dep, DP_PAR0, dep->de_address.ea_addr[0]);
17016           outb_reg1(dep, DP_PAR1, dep->de_address.ea_addr[1]);
17017           outb_reg1(dep, DP_PAR2, dep->de_address.ea_addr[2]);
17018           outb_reg1(dep, DP_PAR3, dep->de_address.ea_addr[3]);
17019           outb_reg1(dep, DP_PAR4, dep->de_address.ea_addr[4]);
17020           outb_reg1(dep, DP_PAR5, dep->de_address.ea_addr[5]);
17021
17022           outb_reg1(dep, DP_MAR0, 0xff);
17023           outb_reg1(dep, DP_MAR1, 0xff);
17024           outb_reg1(dep, DP_MAR2, 0xff);
```

```
17025           outb_reg1(dep, DP_MAR3, 0xff);
17026           outb_reg1(dep, DP_MAR4, 0xff);
17027           outb_reg1(dep, DP_MAR5, 0xff);
17028           outb_reg1(dep, DP_MAR6, 0xff);
17029           outb_reg1(dep, DP_MAR7, 0xff);
17030
17031           outb_reg1(dep, DP_CURR, dep->de_startpage + 1);
17032           outb_reg1(dep, DP_CR, CR_PS_P0 | CR_DM_ABORT);
17033
17034           dp_rcr_reg = 0;
17035           if (dep->de_flags & DEF_PROMISC)
17036                   dp_rcr_reg |= RCR_AB | RCR_PRO | RCR_AM;
17037           if (dep->de_flags & DEF_BROAD)
17038                   dp_rcr_reg |= RCR_AB;
17039           if (dep->de_flags & DEF_MULTI)
17040                   dp_rcr_reg |= RCR_AM;
17041           outb_reg0(dep, DP_RCR, dp_rcr_reg);
17042           inb_reg0(dep, DP_CNTR0);                /* reset counters by reading */
17043           inb_reg0(dep, DP_CNTR1);
17044           inb_reg0(dep, DP_CNTR2);
17045
17046           outb_reg0(dep, DP_IMR, IMR_PRXE | IMR_PTXE | IMR_RXEE | IMR_TXEE |
17047                   IMR_OVWE | IMR_CNTE);
17048           outb_reg0(dep, DP_CR, CR_STA | CR_DM_ABORT);
17049
17050           /* Finish the initialization. */
17051           dep->de_flags |= DEF_ENABLED;
17052           for (i= 0; i<dep->de_sendq_nr; i++)
17053                   dep->de_sendq[i].sq_filled= 0;
17054           dep->de_sendq_head= 0;
17055           dep->de_sendq_tail= 0;
17056           if (!dep->de_prog_IO)
17057           {
17058                   dep->de_user2nicf= dp_user2nic;
17059                   dep->de_nic2userf= dp_nic2user;
17060                   dep->de_getblockf= dp_getblock;
17061           }
17062           else if (dep->de_16bit)
17063           {
17064                   dep->de_user2nicf= dp_pio16_user2nic;
17065                   dep->de_nic2userf= dp_pio16_nic2user;
17066                   dep->de_getblockf= dp_pio16_getblock;
17067           }
17068           else
17069           {
17070                   dep->de_user2nicf= dp_pio8_user2nic;
17071                   dep->de_nic2userf= dp_pio8_nic2user;
17072                   dep->de_getblockf= dp_pio8_getblock;
17073           }
17074
17075           /* set the interrupt handler */
17076           put_irq_handler(dep->de_irq, dp_handler);
17077           enable_irq(dep->de_irq);
17078   }


17081   /*============================================================================*
17082    *                              dp_confaddr                                   *
17083    *============================================================================*/
17084   static void dp_confaddr(dep)
```

```
17085   dpeth_t *dep;
17086   {
17087           int i;
17088           char eakey[16];
17089           static char eafmt[]= "x:x:x:x:x:x";
17090           long v;
17091
17092           /* User defined ethernet address? */
17093           strcpy(eakey, dp_conf[dep-de_table].dpc_envvar);
17094           strcat(eakey, "_EA");
17095
17096           for (i= 0; i < 6; i++)
17097           {
17098                   v= dep->de_address.ea_addr[i];
17099                   if (env_parse(eakey, eafmt, i, &v, 0x00L, 0xFFL) != EP_SET)
17100                           break;
17101                   dep->de_address.ea_addr[i]= v;
17102           }
17103
17104           if (i != 0 && i != 6)
17105           {
17106                   /* It's all or nothing; force a panic. */
17107                   (void) env_parse(eakey, "?", 0, &v, 0L, 0L);
17108           }
17109   }


17112   /*===========================================================================*
17113    *                              dp_reinit                                    *
17114    *===========================================================================*/
17115   static void dp_reinit(dep)
17116   dpeth_t *dep;
17117   {
17118           int dp_rcr_reg;
17119
17120           outb_reg0(dep, DP_CR, CR_PS_P0);
17121
17122           dp_rcr_reg = 0;
17123           if (dep->de_flags & DEF_PROMISC)
17124                   dp_rcr_reg |= RCR_AB | RCR_PRO | RCR_AM;
17125           if (dep->de_flags & DEF_BROAD)
17126                   dp_rcr_reg |= RCR_AB;
17127           if (dep->de_flags & DEF_MULTI)
17128                   dp_rcr_reg |= RCR_AM;
17129           outb_reg0(dep, DP_RCR, dp_rcr_reg);
17130   }


17133   /*===========================================================================*
17134    *                              dp_reset                                     *
17135    *===========================================================================*/
17136   static void dp_reset(dep)
17137   dpeth_t *dep;
17138   {
17139           int i;
17140
17141           /* Stop chip */
17142           outb_reg0(dep, DP_CR, CR_STP | CR_DM_ABORT);
17143           outb_reg0(dep, DP_RBCR0, 0);
17144           outb_reg0(dep, DP_RBCR1, 0);
```

```
17145           for (i= 0; i < 0x1000 && ((inb_reg0(dep, DP_ISR) & ISR_RST) == 0); i++)
17146                   ; /* Do nothing */
17147           outb_reg0(dep, DP_TCR, TCR_1EXTERNAL|TCR_OFST);
17148           outb_reg0(dep, DP_CR, CR_STA|CR_DM_ABORT);
17149           outb_reg0(dep, DP_TCR, TCR_NORMAL|TCR_OFST);
17150
17151           /* Acknowledge the ISR_RDC (remote dma) interrupt. */
17152           for (i= 0; i < 0x1000 && ((inb_reg0(dep, DP_ISR) & ISR_RDC) == 0); i++)
17153                   ; /* Do nothing */
17154           outb_reg0(dep, DP_ISR, inb_reg0(dep, DP_ISR) & ~ISR_RDC);
17155
17156           /* Reset the transmit ring. If we were transmitting a packet, we
17157            * pretend that the packet is processed. Higher layers will
17158            * retransmit if the packet wasn't actually sent.
17159            */
17160           dep->de_sendq_head= dep->de_sendq_tail= 0;
17161           for (i= 0; i<dep->de_sendq_nr; i++)
17162                   dep->de_sendq[i].sq_filled= 0;
17163           dp_send(dep);
17164           dep->de_flags &= ~DEF_STOPPED;
17165   }


17168   /*============================================================================*
17169    *                              dp_check_ints                                 *
17170    *============================================================================*/
17171   static void dp_check_ints(dep)
17172   dpeth_t *dep;
17173   {
17174           int isr, tsr;
17175           int size, sendq_tail;
17176
17177           if (!(dep->de_flags & DEF_ENABLED))
17178                   panic("dp8390: got premature interrupt", NO_NUM);
17179
17180           for(;;)
17181           {
17182                   isr = inb_reg0(dep, DP_ISR);
17183                   if (!isr)
17184                           break;
17185                   outb_reg0(dep, DP_ISR, isr);
17186                   if (isr & (ISR_PTX|ISR_TXE))
17187                   {
17188                           if (isr & ISR_TXE)
17189                           {
17190   #if DEBUG
17191            { printf("dp8390: got send Error\n"); }
17192   #endif
17193                                   dep->de_stat.ets_sendErr++;
17194                           }
17195                           else
17196                           {
17197                                   tsr = inb_reg0(dep, DP_TSR);
17198
17199                                   if (tsr & TSR_PTX) dep->de_stat.ets_packetT++;
17200                                   if (tsr & TSR_DFR) dep->de_stat.ets_transDef++;
17201                                   if (tsr & TSR_COL) dep->de_stat.ets_collision++;
17202                                   if (tsr & TSR_ABT) dep->de_stat.ets_transAb++;
17203                                   if (tsr & TSR_CRS) dep->de_stat.ets_carrSense++;
17204                                   if (tsr & TSR_FU
```

```
17205                                           && ++dep->de_stat.ets_fifoUnder <= 10)
17206                                   {
17207                                           printf("dp8390: fifo underrun\n");
17208                                   }
17209                                   if (tsr & TSR_CDH
17210                                           && ++dep->de_stat.ets_CDheartbeat <= 10)
17211                                   {
17212                                           printf(
17213                                           "dp8390: CD heart beat failure\n");
17214                                   }
17215                                   if (tsr & TSR_OWC) dep->de_stat.ets_OWC++;
17216                           }
17217                           sendq_tail= dep->de_sendq_tail;
17218
17219                           if (!(dep->de_sendq[sendq_tail].sq_filled))
17220                           {
17221                                   /* Software bug? */
17222                                   assert(!debug);
17223
17224                                   /* Or hardware bug? */
17225                                   printf(
17226                           "dp8390: transmit interrupt, but not sending\n");
17227                                   continue;
17228                           }
17229                           dep->de_sendq[sendq_tail].sq_filled= 0;
17230                           if (++sendq_tail == dep->de_sendq_nr)
17231                                   sendq_tail= 0;
17232                           dep->de_sendq_tail= sendq_tail;
17233                           if (dep->de_sendq[sendq_tail].sq_filled)
17234                           {
17235                                   size= dep->de_sendq[sendq_tail].sq_size;
17236                                   outb_reg0(dep, DP_TPSR,
17237                                           dep->de_sendq[sendq_tail].sq_sendpage);
17238                                   outb_reg0(dep, DP_TBCR1, size >> 8);
17239                                   outb_reg0(dep, DP_TBCR0, size & 0xff);
17240                                   outb_reg0(dep, DP_CR, CR_TXP);  /* there is goes.. */
17241                           }
17242                           if (dep->de_flags & DEF_SEND_AVAIL)
17243                                   dp_send(dep);
17244                   }
17245
17246                   if (isr & ISR_PRX)
17247                           dp_recv(dep);
17248
17249                   if (isr & ISR_RXE) dep->de_stat.ets_recvErr++;
17250                   if (isr & ISR_CNT)
17251                   {
17252                           dep->de_stat.ets_CRCerr += inb_reg0(dep, DP_CNTR0);
17253                           dep->de_stat.ets_frameAll += inb_reg0(dep, DP_CNTR1);
17254                           dep->de_stat.ets_missedP += inb_reg0(dep, DP_CNTR2);
17255                   }
17256                   if (isr & ISR_OVW)
17257                   {
17258   #if DEBUG
17259    { printW(); printf("dp8390: got overwrite warning\n"); }
17260   #endif
17261                   }
17262                   if (isr & ISR_RDC)
17263                   {
17264                           /* Nothing to do */
```

```
17265                   }
17266                   if (isr & ISR_RST)
17267                   {
17268                           /* this means we got an interrupt but the ethernet
17269                            * chip is shutdown. We set the flag DEF_STOPPED,
17270                            * and continue processing arrived packets. When the
17271                            * receive buffer is empty, we reset the dp8390.
17272                            */
17273   #if DEBUG
17274    { printW(); printf("dp8390: NIC stopped\n"); }
17275   #endif
17276                           dep->de_flags |= DEF_STOPPED;
17277                           break;
17278                   }
17279           }
17280           if ((dep->de_flags & (DEF_READING|DEF_STOPPED)) ==
17281                                                   (DEF_READING|DEF_STOPPED))
17282           {
17283                   /* The chip is stopped, and all arrived packets are
17284                    * delivered.
17285                    */
17286                   dp_reset(dep);
17287           }
17288   }


17291   /*===========================================================================*
17292    *                                dp_recv                                    *
17293    *===========================================================================*/
17294   static void dp_recv(dep)
17295   dpeth_t *dep;
17296   {
17297           dp_rcvhdr_t header;
17298           unsigned pageno, curr, next;
17299           vir_bytes length;
17300           int packet_processed, r;
17301           u16_t eth_type;
17302
17303           packet_processed = FALSE;
17304           pageno = inb_reg0(dep, DP_BNRY) + 1;
17305           if (pageno == dep->de_stoppage) pageno = dep->de_startpage;
17306
17307           do
17308           {
17309                   outb_reg0(dep, DP_CR, CR_PS_P1);
17310                   curr = inb_reg1(dep, DP_CURR);
17311                   outb_reg0(dep, DP_CR, CR_PS_P0);
17312
17313                   if (curr == pageno) break;
17314
17315                   (dep->de_getblockf)(dep, pageno, (size_t)0, sizeof(header),
17316                           &header);
17317                   (dep->de_getblockf)(dep, pageno, sizeof(header) +
17318                           2*sizeof(ether_addr_t), sizeof(eth_type), &eth_type);
17319
17320                   length = (header.dr_rbcl | (header.dr_rbch << 8)) -
17321                           sizeof(dp_rcvhdr_t);
17322                   next = header.dr_next;
17323                   if (length < 60 || length > 1514)
17324                   {
```

```
17325                           printf(
17326                           "dp8390: packet with strange length arrived: %d\n",
17327                                   length);
17328                           next= curr;
17329                   }
17330                   else if (next < dep->de_startpage || next >= dep->de_stoppage)
17331                   {
17332                           printf("dp8390: strange next page\n");
17333                           next= curr;
17334                   }
17335                   else if (eth_type == eth_ign_proto)
17336                   {
17337                           /* hack: ignore packets of a given protocol, useful
17338                            * if you share a net with 80 computers sending
17339                            * Amoeba FLIP broadcasts.  (Protocol 0x8146.)
17340                            */
17341                           static int first= 1;
17342                           if (first)
17343                           {
17344                                   first= 0;
17345                                   printW();
17346                                   printf("dropping proto %04x packet\n",
17347                                           ntohs(eth_ign_proto));
17348                           }
17349                           dep->de_stat.ets_packetR++;
17350                           next = curr;
17351                   }
17352                   else if (header.dr_status & RSR_FO)
17353                   {
17354                           /* This is very serious, so we issue a warning and
17355                            * reset the buffers */
17356                           printf(
17357                           "dp8390: fifo overrun, resetting receive buffer\n");
17358                           dep->de_stat.ets_fifoOver++;
17359                           next = curr;
17360                   }
17361                   else if ((header.dr_status & RSR_PRX) &&
17362                                           (dep->de_flags & DEF_ENABLED))
17363                   {
17364                           r = dp_pkt2user(dep, pageno, length);
17365                           if (r != OK)
17366                                   return;
17367
17368                           packet_processed = TRUE;
17369                           dep->de_stat.ets_packetR++;
17370                   }
17371                   if (next == dep->de_startpage)
17372                           outb_reg0(dep, DP_BNRY, dep->de_stoppage - 1);
17373                   else
17374                           outb_reg0(dep, DP_BNRY, next - 1);
17375
17376                   pageno = next;
17377           }
17378           while (!packet_processed);
17379   }


17382   /*==========================================================================*
17383    *                              dp_send                                     *
17384    *==========================================================================*/
```

```
17385   static void dp_send(dep)
17386   dpeth_t *dep;
17387   {
17388           if (!(dep->de_flags & DEF_SEND_AVAIL))
17389                   return;
17390
17391           dep->de_flags &= ~DEF_SEND_AVAIL;
17392           switch(dep->de_sendmsg.m_type)
17393           {
17394           case DL_WRITE:  do_vwrite(&dep->de_sendmsg, TRUE, FALSE);       break;
17395           case DL_WRITEV: do_vwrite(&dep->de_sendmsg, TRUE, TRUE);        break;
17396           default:
17397                   panic("dp8390: wrong type:", dep->de_sendmsg.m_type);
17398                   break;
17399           }
17400   }


17403   /*============================================================================*
17404    *                               dp_getblock                                  *
17405    *============================================================================*/
17406   static void dp_getblock(dep, page, offset, size, dst)
17407   dpeth_t *dep;
17408   int page;
17409   size_t offset;
17410   size_t size;
17411   void *dst;
17412   {
17413           u16_t *ha;
17414           int i;
17415
17416           ha = (u16_t *) dst;
17417           offset = page * DP_PAGESIZE + offset;
17418           assert(!(size & 1));
17419           for (i= 0; i<size; i+=2)
17420           {
17421                   *ha = mem_rdw(dep->de_memsegm, offset+i);
17422                   ha++;
17423           }
17424   }


17427   /*============================================================================*
17428    *                               dp_pio8_getblock                             *
17429    *============================================================================*/
17430   static void dp_pio8_getblock(dep, page, offset, size, dst)
17431   dpeth_t *dep;
17432   int page;
17433   size_t offset;
17434   size_t size;
17435   void *dst;
17436   {
17437           u8_t *ha;
17438           int i;
17439
17440           ha = (u8_t *) dst;
17441           offset = page * DP_PAGESIZE + offset;
17442           outb_reg0(dep, DP_RBCR0, size & 0xFF);
17443           outb_reg0(dep, DP_RBCR1, size >> 8);
17444           outb_reg0(dep, DP_RSAR0, offset & 0xFF);
```

```
17445           outb_reg0(dep, DP_RSAR1, offset >> 8);
17446           outb_reg0(dep, DP_CR, CR_DM_RR | CR_PS_P0 | CR_STA);
17447
17448           for (i= 0; i<size; i++)
17449                   ha[i]= in_byte(dep->de_data_port);
17450   }


17453   /*===========================================================================*
17454    *                               dp_pio16_getblock                           *
17455    *===========================================================================*/
17456   static void dp_pio16_getblock(dep, page, offset, size, dst)
17457   dpeth_t *dep;
17458   int page;
17459   size_t offset;
17460   size_t size;
17461   void *dst;
17462   {
17463           u16_t *ha;
17464           int i;
17465
17466           ha = (u16_t *) dst;
17467           offset = page * DP_PAGESIZE + offset;
17468           outb_reg0(dep, DP_RBCR0, size & 0xFF);
17469           outb_reg0(dep, DP_RBCR1, size >> 8);
17470           outb_reg0(dep, DP_RSAR0, offset & 0xFF);
17471           outb_reg0(dep, DP_RSAR1, offset >> 8);
17472           outb_reg0(dep, DP_CR, CR_DM_RR | CR_PS_P0 | CR_STA);
17473
17474           assert (!(size & 1));
17475           size /= 2;
17476           for (i= 0; i<size; i++)
17477                   ha[i]= in_word(dep->de_data_port);
17478   }


17481   /*===========================================================================*
17482    *                               dp_pkt2user                                 *
17483    *===========================================================================*/
17484   static int dp_pkt2user(dep, page, length)
17485   dpeth_t *dep;
17486   int page, length;
17487   {
17488           int last, count;
17489
17490           if (!(dep->de_flags & DEF_READING))
17491                   return EGENERIC;
17492
17493           last = page + (length - 1) / DP_PAGESIZE;
17494           if (last >= dep->de_stoppage)
17495           {
17496                   count = (dep->de_stoppage - page) * DP_PAGESIZE -
17497                           sizeof(dp_rcvhdr_t);
17498
17499                   /* Save read_iovec since we need it twice. */
17500                   dep->de_tmp_iovec = dep->de_read_iovec;
17501                   (dep->de_nic2userf)(dep, page * DP_PAGESIZE +
17502                           sizeof(dp_rcvhdr_t), &dep->de_tmp_iovec, 0, count);
17503                   (dep->de_nic2userf)(dep, dep->de_startpage * DP_PAGESIZE,
17504                                   &dep->de_read_iovec, count, length - count);
```

```
17505           }
17506           else
17507           {
17508                   (dep->de_nic2userf)(dep, page * DP_PAGESIZE +
17509                           sizeof(dp_rcvhdr_t), &dep->de_read_iovec, 0, length);
17510           }
17511
17512           dep->de_read_s = length;
17513           dep->de_flags |= DEF_PACK_RECV;
17514           dep->de_flags &= ~DEF_READING;
17515
17516           return OK;
17517   }


17520   /*===========================================================================*
17521    *                              dp_user2nic                                  *
17522    *===========================================================================*/
17523   static void dp_user2nic(dep, iovp, offset, nic_addr, count)
17524   dpeth_t *dep;
17525   iovec_dat_t *iovp;
17526   vir_bytes offset;
17527   int nic_addr;
17528   vir_bytes count;
17529   {
17530           phys_bytes phys_hw, phys_user;
17531           int bytes, i;
17532
17533           phys_hw = dep->de_linmem + nic_addr;
17534
17535           i= 0;
17536           while (count > 0)
17537           {
17538                   if (i >= IOVEC_NR)
17539                   {
17540                           dp_next_iovec(iovp);
17541                           i= 0;
17542                           continue;
17543                   }
17544                   assert(i < iovp->iod_iovec_s);
17545                   if (offset >= iovp->iod_iovec[i].iov_size)
17546                   {
17547                           offset -= iovp->iod_iovec[i].iov_size;
17548                           i++;
17549                           continue;
17550                   }
17551                   bytes = iovp->iod_iovec[i].iov_size - offset;
17552                   if (bytes > count)
17553                           bytes = count;
17554
17555                   phys_user = numap(iovp->iod_proc_nr,
17556                           iovp->iod_iovec[i].iov_addr + offset, bytes);
17557                   if (!phys_user)
17558                           panic("dp8390: umap failed\n", NO_NUM);
17559                   phys_copy(phys_user, phys_hw, (phys_bytes) bytes);
17560                   count -= bytes;
17561                   phys_hw += bytes;
17562                   offset += bytes;
17563           }
17564           assert(count == 0);
```

```
17565   }


17568   /*===========================================================================*
17569    *                              dp_pio8_user2nic                             *
17570    *===========================================================================*/
17571   static void dp_pio8_user2nic(dep, iovp, offset, nic_addr, count)
17572   dpeth_t *dep;
17573   iovec_dat_t *iovp;
17574   vir_bytes offset;
17575   int nic_addr;
17576   vir_bytes count;
17577   {
17578           phys_bytes phys_user;
17579           int bytes, i;
17580
17581           outb_reg0(dep, DP_ISR, ISR_RDC);
17582
17583           outb_reg0(dep, DP_RBCR0, count & 0xFF);
17584           outb_reg0(dep, DP_RBCR1, count >> 8);
17585           outb_reg0(dep, DP_RSAR0, nic_addr & 0xFF);
17586           outb_reg0(dep, DP_RSAR1, nic_addr >> 8);
17587           outb_reg0(dep, DP_CR, CR_DM_RW | CR_PS_P0 | CR_STA);
17588
17589           i= 0;
17590           while (count > 0)
17591           {
17592                   if (i >= IOVEC_NR)
17593                   {
17594                           dp_next_iovec(iovp);
17595                           i= 0;
17596                           continue;
17597                   }
17598                   assert(i < iovp->iod_iovec_s);
17599                   if (offset >= iovp->iod_iovec[i].iov_size)
17600                   {
17601                           offset -= iovp->iod_iovec[i].iov_size;
17602                           i++;
17603                           continue;
17604                   }
17605                   bytes = iovp->iod_iovec[i].iov_size - offset;
17606                   if (bytes > count)
17607                           bytes = count;
17608
17609                   phys_user = numap(iovp->iod_proc_nr,
17610                           iovp->iod_iovec[i].iov_addr + offset, bytes);
17611                   if (!phys_user)
17612                           panic("dp8390: umap failed\n", NO_NUM);
17613                   port_write_byte(dep->de_data_port, phys_user, bytes);
17614                   count -= bytes;
17615                   offset += bytes;
17616           }
17617           assert(count == 0);
17618
17619           for (i= 0; i<100; i++)
17620           {
17621                   if (inb_reg0(dep, DP_ISR) & ISR_RDC)
17622                           break;
17623           }
17624           if (i == 100)
```

```
17625           {
17626                   panic("dp8390: remote dma failed to complete", NO_NUM);
17627           }
17628   }


17631   /*==========================================================================*
17632    *                              dp_pio16_user2nic                           *
17633    *==========================================================================*/
17634   static void dp_pio16_user2nic(dep, iovp, offset, nic_addr, count)
17635   dpeth_t *dep;
17636   iovec_dat_t *iovp;
17637   vir_bytes offset;
17638   int nic_addr;
17639   vir_bytes count;
17640   {
17641           phys_bytes phys_user;
17642           vir_bytes ecount;
17643           int bytes, i;
17644           u8_t two_bytes[2];
17645           phys_bytes phys_2bytes;
17646           int odd_byte;
17647
17648           ecount= (count+1) & ~1;
17649           phys_2bytes = vir2phys(two_bytes);
17650           odd_byte= 0;
17651
17652           outb_reg0(dep, DP_ISR, ISR_RDC);
17653           outb_reg0(dep, DP_RBCR0, ecount & 0xFF);
17654           outb_reg0(dep, DP_RBCR1, ecount >> 8);
17655           outb_reg0(dep, DP_RSAR0, nic_addr & 0xFF);
17656           outb_reg0(dep, DP_RSAR1, nic_addr >> 8);
17657           outb_reg0(dep, DP_CR, CR_DM_RW | CR_PS_P0 | CR_STA);
17658
17659           i= 0;
17660           while (count > 0)
17661           {
17662                   if (i >= IOVEC_NR)
17663                   {
17664                           dp_next_iovec(iovp);
17665                           i= 0;
17666                           continue;
17667                   }
17668                   assert(i < iovp->iod_iovec_s);
17669                   if (offset >= iovp->iod_iovec[i].iov_size)
17670                   {
17671                           offset -= iovp->iod_iovec[i].iov_size;
17672                           i++;
17673                           continue;
17674                   }
17675                   bytes = iovp->iod_iovec[i].iov_size - offset;
17676                   if (bytes > count)
17677                           bytes = count;
17678
17679                   phys_user = numap(iovp->iod_proc_nr,
17680                           iovp->iod_iovec[i].iov_addr + offset, bytes);
17681                   if (!phys_user)
17682                           panic("dp8390: umap failed\n", NO_NUM);
17683                   if (odd_byte)
17684                   {
```

```
17685                           phys_copy(phys_user, phys_2bytes+1,
17686                                   (phys_bytes) 1);
17687                           out_word(dep->de_data_port, *(u16_t *)two_bytes);
17688                           count--;
17689                           offset++;
17690                           bytes--;
17691                           phys_user++;
17692                           odd_byte= 0;
17693                           if (!bytes)
17694                                   continue;
17695                   }
17696                   ecount= bytes & ~1;
17697                   if (ecount != 0)
17698                   {
17699                           port_write(dep->de_data_port, phys_user, ecount);
17700                           count -= ecount;
17701                           offset += ecount;
17702                           bytes -= ecount;
17703                           phys_user += ecount;
17704                   }
17705                   if (bytes)
17706                   {
17707                           assert(bytes == 1);
17708                           phys_copy(phys_user, phys_2bytes, (phys_bytes) 1);
17709                           count--;
17710                           offset++;
17711                           bytes--;
17712                           phys_user++;
17713                           odd_byte= 1;
17714                   }
17715           }
17716           assert(count == 0);
17717
17718           if (odd_byte)
17719                   out_word(dep->de_data_port, *(u16_t *)two_bytes);
17720
17721           for (i= 0; i<100; i++)
17722           {
17723                   if (inb_reg0(dep, DP_ISR) & ISR_RDC)
17724                           break;
17725           }
17726           if (i == 100)
17727           {
17728                   panic("dp8390: remote dma failed to complete", NO_NUM);
17729           }
17730   }


17733   /*==========================================================================*
17734    *                              dp_nic2user                                 *
17735    *==========================================================================*/
17736   static void dp_nic2user(dep, nic_addr, iovp, offset, count)
17737   dpeth_t *dep;
17738   int nic_addr;
17739   iovec_dat_t *iovp;
17740   vir_bytes offset;
17741   vir_bytes count;
17742   {
17743           phys_bytes phys_hw, phys_user;
17744           int bytes, i;
```

```
17745
17746           phys_hw = dep->de_linmem + nic_addr;
17747
17748           i= 0;
17749           while (count > 0)
17750           {
17751                   if (i >= IOVEC_NR)
17752                   {
17753                           dp_next_iovec(iovp);
17754                           i= 0;
17755                           continue;
17756                   }
17757                   assert(i < iovp->iod_iovec_s);
17758                   if (offset >= iovp->iod_iovec[i].iov_size)
17759                   {
17760                           offset -= iovp->iod_iovec[i].iov_size;
17761                           i++;
17762                           continue;
17763                   }
17764                   bytes = iovp->iod_iovec[i].iov_size - offset;
17765                   if (bytes > count)
17766                           bytes = count;
17767
17768                   phys_user = numap(iovp->iod_proc_nr,
17769                           iovp->iod_iovec[i].iov_addr + offset, bytes);
17770                   if (!phys_user)
17771                           panic("dp8390: umap failed\n", NO_NUM);
17772                   phys_copy(phys_hw, phys_user, (phys_bytes) bytes);
17773                   count -= bytes;
17774                   phys_hw += bytes;
17775                   offset += bytes;
17776           }
17777           assert(count == 0);
17778   }


17781   /*==========================================================================*
17782    *                                dp_pio8_nic2user                          *
17783    *==========================================================================*/
17784   static void dp_pio8_nic2user(dep, nic_addr, iovp, offset, count)
17785   dpeth_t *dep;
17786   int nic_addr;
17787   iovec_dat_t *iovp;
17788   vir_bytes offset;
17789   vir_bytes count;
17790   {
17791           phys_bytes phys_user;
17792           int bytes, i;
17793
17794           outb_reg0(dep, DP_RBCR0, count & 0xFF);
17795           outb_reg0(dep, DP_RBCR1, count >> 8);
17796           outb_reg0(dep, DP_RSAR0, nic_addr & 0xFF);
17797           outb_reg0(dep, DP_RSAR1, nic_addr >> 8);
17798           outb_reg0(dep, DP_CR, CR_DM_RR | CR_PS_P0 | CR_STA);
17799
17800           i= 0;
17801           while (count > 0)
17802           {
17803                   if (i >= IOVEC_NR)
17804                   {
```

```
17805                           dp_next_iovec(iovp);
17806                           i= 0;
17807                           continue;
17808                   }
17809                   assert(i < iovp->iod_iovec_s);
17810                   if (offset >= iovp->iod_iovec[i].iov_size)
17811                   {
17812                           offset -= iovp->iod_iovec[i].iov_size;
17813                           i++;
17814                           continue;
17815                   }
17816                   bytes = iovp->iod_iovec[i].iov_size - offset;
17817                   if (bytes > count)
17818                           bytes = count;
17819
17820                   phys_user = numap(iovp->iod_proc_nr,
17821                           iovp->iod_iovec[i].iov_addr + offset, bytes);
17822                   if (!phys_user)
17823                           panic("dp8390: umap failed\n", NO_NUM);
17824                   port_read_byte(dep->de_data_port, phys_user, bytes);
17825                   count -= bytes;
17826                   offset += bytes;
17827           }
17828           assert(count == 0);
17829   }


17832   /*===========================================================================*
17833    *                               dp_pio16_nic2user                           *
17834    *===========================================================================*/
17835   static void dp_pio16_nic2user(dep, nic_addr, iovp, offset, count)
17836   dpeth_t *dep;
17837   int nic_addr;
17838   iovec_dat_t *iovp;
17839   vir_bytes offset;
17840   vir_bytes count;
17841   {
17842           phys_bytes phys_user;
17843           vir_bytes ecount;
17844           int bytes, i;
17845           u8_t two_bytes[2];
17846           phys_bytes phys_2bytes;
17847           int odd_byte;
17848
17849           ecount= (count+1) & ~1;
17850           phys_2bytes = vir2phys(two_bytes);
17851           odd_byte= 0;
17852
17853           outb_reg0(dep, DP_RBCR0, ecount & 0xFF);
17854           outb_reg0(dep, DP_RBCR1, ecount >> 8);
17855           outb_reg0(dep, DP_RSAR0, nic_addr & 0xFF);
17856           outb_reg0(dep, DP_RSAR1, nic_addr >> 8);
17857           outb_reg0(dep, DP_CR, CR_DM_RR | CR_PS_P0 | CR_STA);
17858
17859           i= 0;
17860           while (count > 0)
17861           {
17862                   if (i >= IOVEC_NR)
17863                   {
17864                           dp_next_iovec(iovp);
```

```
17865                           i= 0;
17866                           continue;
17867                   }
17868                   assert(i < iovp->iod_iovec_s);
17869                   if (offset >= iovp->iod_iovec[i].iov_size)
17870                   {
17871                           offset -= iovp->iod_iovec[i].iov_size;
17872                           i++;
17873                           continue;
17874                   }
17875                   bytes = iovp->iod_iovec[i].iov_size - offset;
17876                   if (bytes > count)
17877                           bytes = count;
17878
17879                   phys_user = numap(iovp->iod_proc_nr,
17880                           iovp->iod_iovec[i].iov_addr + offset, bytes);
17881                   if (!phys_user)
17882                           panic("dp8390: umap failed\n", NO_NUM);
17883                   if (odd_byte)
17884                   {
17885                           phys_copy(phys_2bytes+1, phys_user, (phys_bytes) 1);
17886                           count--;
17887                           offset++;
17888                           bytes--;
17889                           phys_user++;
17890                           odd_byte= 0;
17891                           if (!bytes)
17892                                   continue;
17893                   }
17894                   ecount= bytes & ~1;
17895                   if (ecount != 0)
17896                   {
17897                           port_read(dep->de_data_port, phys_user, ecount);
17898                           count -= ecount;
17899                           offset += ecount;
17900                           bytes -= ecount;
17901                           phys_user += ecount;
17902                   }
17903                   if (bytes)
17904                   {
17905                           assert(bytes == 1);
17906                           *(u16_t *)two_bytes= in_word(dep->de_data_port);
17907                           phys_copy(phys_2bytes, phys_user, (phys_bytes) 1);
17908                           count--;
17909                           offset++;
17910                           bytes--;
17911                           phys_user++;
17912                           odd_byte= 1;
17913                   }
17914           }
17915           assert(count == 0);
17916   }


17919   /*==========================================================================*
17920    *                              dp_next_iovec                               *
17921    *==========================================================================*/
17922   static void dp_next_iovec(iovp)
17923   iovec_dat_t *iovp;
17924   {
```

```
17925           assert(iovp->iod_iovec_s > IOVEC_NR);
17926
17927           iovp->iod_iovec_s -= IOVEC_NR;
17928
17929           iovp->iod_iovec_addr += IOVEC_NR * sizeof(iovec_t);
17930
17931           get_userdata(iovp->iod_proc_nr, iovp->iod_iovec_addr,
17932                   (iovp->iod_iovec_s > IOVEC_NR ? IOVEC_NR : iovp->iod_iovec_s) *
17933                   sizeof(iovec_t), iovp->iod_iovec);
17934   }


17937   /*===========================================================================*
17938    *                              dp_handler                                   *
17939    *===========================================================================*/
17940   static int dp_handler(irq)
17941   int irq;
17942   {
17943   /* DP8390 interrupt, send message and reenable interrupts. */
17944
17945           assert(irq >= 0 && irq < NR_IRQ_VECTORS);
17946           int_pending[irq]= 1;
17947           interrupt(dpeth_tasknr);
17948           return 1;
17949   }

17951   /*===========================================================================*
17952    *                              conf_hw                                      *
17953    *===========================================================================*/
17954   static void conf_hw(dep)
17955   dpeth_t *dep;
17956   {
17957           static eth_stat_t empty_stat = {0, 0, 0, 0, 0, 0        /* ,... */ };
17958
17959           int ifnr;
17960           dp_conf_t *dcp;
17961
17962           dep->de_mode= DEM_DISABLED;     /* Superfluous */
17963           ifnr= dep-de_table;
17964
17965           dcp= &dp_conf[ifnr];
17966           update_conf(dep, dcp);
17967           if (dep->de_mode != DEM_ENABLED)
17968                   return;
17969           if (!wdeth_probe(dep) && !ne_probe(dep))
17970           {
17971                   printf("dp8390: warning no ethernet card found at 0x%x\n",
17972                           dep->de_base_port);
17973                   dep->de_mode= DEM_DISABLED;
17974                   return;
17975           }
17976
17977           /* Allocate a memory segment, programmed I/O should set the
17978            * memory segment (linmem) to zero.
17979            */
17980           if (dep->de_linmem != 0)
17981           {
17982                   if (protected_mode)
17983                   {
17984                           init_dataseg(&gdt[dcp->dpc_prot_sel / DESC_SIZE],
```

```
17985                                   dep->de_linmem, dep->de_ramsize,
17986                                   TASK_PRIVILEGE);
17987                           dep->de_memsegm= dcp->dpc_prot_sel;
17988                   }
17989                   else
17990                   {
17991                           dep->de_memsegm= physb_to_hclick(dep->de_linmem);
17992                   }
17993           }
17994
17995   /* XXX */ if (dep->de_linmem == 0) dep->de_linmem= 0xFFFF0000;
17996
17997           dep->de_flags = DEF_EMPTY;
17998           dep->de_stat = empty_stat;
17999   }


18002   /*===========================================================================*
18003    *                              update_conf                                  *
18004    *===========================================================================*/
18005   static void update_conf(dep, dcp)
18006   dpeth_t *dep;
18007   dp_conf_t *dcp;
18008   {
18009           long v;
18010           static char dpc_fmt[] = "x:d:x";
18011
18012           /* Get the default settings and modify them from the environment. */
18013           dep->de_mode= DEM_SINK;
18014           v= dcp->dpc_port;
18015           switch (env_parse(dcp->dpc_envvar, dpc_fmt, 0, &v, 0x000L, 0x3FFL)) {
18016           case EP_OFF:
18017                   dep->de_mode= DEM_DISABLED;
18018                   break;
18019           case EP_ON:
18020           case EP_SET:
18021                   dep->de_mode= DEM_ENABLED;      /* Might become disabled if
18022                                                    * all probes fail */
18023                   break;
18024           }
18025           dep->de_base_port= v;
18026
18027           v= dcp->dpc_irq | DEI_DEFAULT;
18028           (void) env_parse(dcp->dpc_envvar, dpc_fmt, 1, &v, 0L,
18029                                                   (long) NR_IRQ_VECTORS - 1);
18030           dep->de_irq= v;
18031
18032           v= dcp->dpc_mem;
18033           (void) env_parse(dcp->dpc_envvar, dpc_fmt, 2, &v, 0L, LONG_MAX);
18034           dep->de_linmem= v;
18035   }


18038   /*===========================================================================*
18039    *                              calc_iovec_size                              *
18040    *===========================================================================*/
18041   static int calc_iovec_size(iovp)
18042   iovec_dat_t *iovp;
18043   {
18044           /* Calculate the size of a request. Note that the iovec_dat
```

```
18045            * structure will be unusable after calc_iovec_size.
18046            */
18047           int size;
18048           int i;
18049
18050           size= 0;
18051           i= 0;
18052           while (i < iovp->iod_iovec_s)
18053           {
18054                   if (i >= IOVEC_NR)
18055                   {
18056                           dp_next_iovec(iovp);
18057                           i= 0;
18058                           continue;
18059                   }
18060                   size += iovp->iod_iovec[i].iov_size;
18061                   i++;
18062           }
18063           return size;
18064   }


18067   /*==========================================================================*
18068    *                                reply                                     *
18069    *==========================================================================*/
18070   static void reply(dep, err, may_block)
18071   dpeth_t *dep;
18072   int err;
18073   int may_block;
18074   {
18075           message reply;
18076           int status;
18077           int r;
18078
18079           status = 0;
18080           if (dep->de_flags & DEF_PACK_SEND)
18081                   status |= DL_PACK_SEND;
18082           if (dep->de_flags & DEF_PACK_RECV)
18083                   status |= DL_PACK_RECV;
18084
18085           reply.m_type = DL_TASK_REPLY;
18086           reply.DL_PORT = dep - de_table;
18087           reply.DL_PROC = dep->de_client;
18088           reply.DL_STAT = status | ((u32_t) err << 16);
18089           reply.DL_COUNT = dep->de_read_s;
18090           reply.DL_CLCK = get_uptime();
18091           r= send(dep->de_client, &reply);
18092   #if 0
18093           /* XXX  What is the reason that this doesn't happen? */
18094           if (result == ELOCKED && may_block)
18095                   return;
18096   #endif
18097           if (r < 0)
18098                   panic("dp8390: send failed:", r);
18099
18100           dep->de_read_s = 0;
18101           dep->de_flags &= ~(DEF_PACK_SEND | DEF_PACK_RECV);
18102   }
```

```
18105   /*==========================================================================*
18106    *                              mess_reply                                  *
18107    *==========================================================================*/
18108   static void mess_reply(req, reply_mess)
18109   message *req;
18110   message *reply_mess;
18111   {
18112           if (send(req->m_source, reply_mess) != OK)
18113                   panic("dp8390: unable to mess_reply", NO_NUM);
18114   }


18117   /*==========================================================================*
18118    *                              get_userdata                                *
18119    *==========================================================================*/
18120   static void get_userdata(user_proc, user_addr, count, loc_addr)
18121   int user_proc;
18122   vir_bytes user_addr;
18123   vir_bytes count;
18124   void *loc_addr;
18125   {
18126           phys_bytes src;
18127
18128           src = numap(user_proc, user_addr, count);
18129           if (!src)
18130                   panic("dp8390: umap failed", NO_NUM);
18131
18132           phys_copy(src, vir2phys(loc_addr), (phys_bytes) count);
18133   }


18136   /*==========================================================================*
18137    *                              put_userdata                                *
18138    *==========================================================================*/
18139   static void put_userdata(user_proc, user_addr, count, loc_addr)
18140   int user_proc;
18141   vir_bytes user_addr;
18142   vir_bytes count;
18143   void *loc_addr;
18144   {
18145           phys_bytes dst;
18146
18147           dst = numap(user_proc, user_addr, count);
18148           if (!dst)
18149                   panic("dp8390: umap failed", NO_NUM);
18150
18151           phys_copy(vir2phys(loc_addr), dst, (phys_bytes) count);
18152   }

18154   #endif /* ENABLE_NETWORKING */
18155
18156   /*
18157    * $PchHeader: /mount/hd2/minix/sys/kernel/ibm/RCS/dp8390.c,v 1.4 1995/06/13 08:10:42 philip Exp $
18158    */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/driver.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

18200    /* This file contains device independent device driver interface.
18201     *                                                     Author: Kees J. Bot.
18202     *
18203     * The drivers support the following operations (using message format m2):
18204     *
18205     *    m_type      DEVICE    PROC_NR     COUNT    POSITION  ADRRESS
18206     * ----------------------------------------------------------------
18207     * |  DEV_OPEN  | device  | proc nr |         |         |         |
18208     * |------------+---------+---------+---------+---------+---------|
18209     * |  DEV_CLOSE | device  | proc nr |         |         |         |
18210     * |------------+---------+---------+---------+---------+---------|
18211     * |  DEV_READ  | device  | proc nr |  bytes  |  offset | buf ptr |
18212     * |------------+---------+---------+---------+---------+---------|
18213     * |  DEV_WRITE | device  | proc nr |  bytes  |  offset | buf ptr |
18214     * |------------+---------+---------+---------+---------+---------|
18215     * |SCATTERED_IO| device  | proc nr | requests|         | iov ptr |
18216     * ----------------------------------------------------------------
18217     * |  DEV_IOCTL | device  | proc nr |func code|         | buf ptr |
18218     * ----------------------------------------------------------------
18219     *
18220     * The file contains one entry point:
18221     *
18222     *   driver_task:       called by the device dependent task entry
18223     *
18224     *
18225     * Constructed 92/04/02 by Kees J. Bot from the old AT wini and floppy driver.
18226     */
18227
18228    #include "kernel.h"
18229    #include <sys/ioctl.h>
18230    #include "driver.h"
18231
18232    #if (CHIP == INTEL)
18233    #if ENABLE_ADAPTEC_SCSI && DMA_BUF_SIZE < 2048
18234    /* A bit extra scratch for the Adaptec driver. */
18235    #define BUF_EXTRA       (2048 - DMA_BUF_SIZE)
18236    #else
18237    #define BUF_EXTRA       0
18238    #endif
18239
18240    /* Claim space for variables. */
18241    PRIVATE u8_t buffer[(unsigned) 2 * DMA_BUF_SIZE + BUF_EXTRA];
18242    u8_t *tmp_buf;                  /* the DMA buffer eventually */
18243    phys_bytes tmp_phys;            /* phys address of DMA buffer */
18244
18245    #else /* CHIP != INTEL */
18246
18247    /* Claim space for variables. */
18248    u8_t tmp_buf[DMA_BUF_SIZE];     /* the DMA buffer */
18249    phys_bytes tmp_phys;            /* phys address of DMA buffer */
18250
18251    #endif /* CHIP != INTEL */
18252
18253    FORWARD _PROTOTYPE( void init_buffer, (void) );
18254
```

```
18255
18256   /*===========================================================================*
18257    *                              driver_task                                  *
18258    *===========================================================================*/
18259   PUBLIC void driver_task(dp)
18260   struct driver *dp;      /* Device dependent entry points. */
18261   {
18262   /* Main program of any device driver task. */
18263
18264     int r, caller, proc_nr;
18265     message mess;
18266
18267     init_buffer();        /* Get a DMA buffer. */
18268
18269
18270     /* Here is the main loop of the disk task.  It waits for a message, carries
18271      * it out, and sends a reply.
18272      */
18273
18274     while (TRUE) {
18275           /* First wait for a request to read or write a disk block. */
18276           receive(ANY, &mess);
18277
18278           caller = mess.m_source;
18279           proc_nr = mess.PROC_NR;
18280
18281           switch (caller) {
18282           case HARDWARE:
18283                   /* Leftover interrupt. */
18284                   continue;
18285           case FS_PROC_NR:
18286                   /* The only legitimate caller. */
18287                   break;
18288           default:
18289                   printf("%s: got message from %d\n", (*dp->dr_name)(), caller);
18290                   continue;
18291           }
18292
18293           /* Now carry out the work. */
18294           switch(mess.m_type) {
18295               case DEV_OPEN:      r = (*dp->dr_open)(dp, &mess);  break;
18296               case DEV_CLOSE:     r = (*dp->dr_close)(dp, &mess); break;
18297               case DEV_IOCTL:     r = (*dp->dr_ioctl)(dp, &mess); break;
18298
18299               case DEV_READ:
18300               case DEV_WRITE:     r = do_rdwt(dp, &mess);         break;
18301
18302               case SCATTERED_IO:  r = do_vrdwt(dp, &mess);        break;
18303               default:            r = EINVAL;                     break;
18304           }
18305
18306           /* Clean up leftover state. */
18307           (*dp->dr_cleanup)();
18308
18309           /* Finally, prepare and send the reply message. */
18310           mess.m_type = TASK_REPLY;
18311           mess.REP_PROC_NR = proc_nr;
18312
18313           mess.REP_STATUS = r;    /* # of bytes transferred or error code */
18314           send(caller, &mess);    /* send reply to caller */
```

```
18315       }
18316     }


18319     /*===========================================================================*
18320      *                              init_buffer                                  *
18321      *===========================================================================*/
18322     PRIVATE void init_buffer()
18323     {
18324     /* Select a buffer that can safely be used for dma transfers.  It may also
18325      * be used to read partition tables and such.  Its absolute address is
18326      * 'tmp_phys', the normal address is 'tmp_buf'.
18327      */
18328
18329     #if (CHIP == INTEL)
18330       tmp_buf = buffer;
18331       tmp_phys = vir2phys(buffer);
18332
18333       if (tmp_phys == 0) panic("no DMA buffer", NO_NUM);
18334
18335       if (dma_bytes_left(tmp_phys) < DMA_BUF_SIZE) {
18336             /* First half of buffer crosses a 64K boundary, can't DMA into that */
18337             tmp_buf += DMA_BUF_SIZE;
18338             tmp_phys += DMA_BUF_SIZE;
18339       }
18340     #else /* CHIP != INTEL */
18341       tmp_phys = vir2phys(tmp_buf);
18342     #endif /* CHIP != INTEL */
18343     }


18346     /*===========================================================================*
18347      *                              do_rdwt                                      *
18348      *===========================================================================*/
18349     PUBLIC int do_rdwt(dp, m_ptr)
18350     struct driver *dp;              /* device dependent entry points */
18351     message *m_ptr;                 /* pointer to read or write message */
18352     {
18353     /* Carry out a single read or write request. */
18354       struct iorequest_s ioreq;
18355       int r;
18356
18357       if (m_ptr->COUNT <= 0) return(EINVAL);
18358
18359       if ((*dp->dr_prepare)(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
18360
18361       ioreq.io_request = m_ptr->m_type;
18362       ioreq.io_buf = m_ptr->ADDRESS;
18363       ioreq.io_position = m_ptr->POSITION;
18364       ioreq.io_nbytes = m_ptr->COUNT;
18365
18366       r = (*dp->dr_schedule)(m_ptr->PROC_NR, &ioreq);
18367
18368       if (r == OK) (void) (*dp->dr_finish)();
18369
18370       r = ioreq.io_nbytes;
18371       return(r < 0 ? r : m_ptr->COUNT - r);
18372     }
```

```
18375   /*===========================================================================*
18376    *                                do_vrdwt                                   *
18377    *===========================================================================*/
18378   PUBLIC int do_vrdwt(dp, m_ptr)
18379   struct driver *dp;      /* device dependent entry points */
18380   message *m_ptr;         /* pointer to read or write message */
18381   {
18382   /* Fetch a vector of i/o requests.  Handle requests one at a time.  Return
18383    * status in the vector.
18384    */
18385
18386     struct iorequest_s *iop;
18387     static struct iorequest_s iovec[NR_IOREQS];
18388     phys_bytes iovec_phys;
18389     unsigned nr_requests;
18390     int request;
18391     int r;
18392     phys_bytes user_iovec_phys;
18393
18394     nr_requests = m_ptr->COUNT;
18395
18396     if (nr_requests > sizeof iovec / sizeof iovec[0])
18397           panic("FS passed too big an I/O vector", nr_requests);
18398
18399     iovec_phys = vir2phys(iovec);
18400     user_iovec_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
18401                             (vir_bytes) (nr_requests * sizeof iovec[0]));
18402
18403     if (user_iovec_phys == 0)
18404           panic("FS passed a bad I/O vector", (int) m_ptr->ADDRESS);
18405
18406     phys_copy(user_iovec_phys, iovec_phys,
18407                               (phys_bytes) nr_requests * sizeof iovec[0]);
18408
18409     if ((*dp->dr_prepare)(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
18410
18411     for (request = 0, iop = iovec; request < nr_requests; request++, iop++) {
18412           if ((r = (*dp->dr_schedule)(m_ptr->PROC_NR, iop)) != OK) break;
18413     }
18414
18415     if (r == OK) (void) (*dp->dr_finish)();
18416
18417     phys_copy(iovec_phys, user_iovec_phys,
18418                               (phys_bytes) nr_requests * sizeof iovec[0]);
18419     return(OK);
18420   }


18423   /*===========================================================================*
18424    *                                no_name                                    *
18425    *===========================================================================*/
18426   PUBLIC char *no_name()
18427   {
18428   /* If no specific name for the device. */
18429
18430     return(tasktab[proc_number(proc_ptr) + NR_TASKS].name);
18431   }


18434   /*===========================================================================*
```

```
18435    *                              do_nop                                   *
18436    *=======================================================================*/
18437   PUBLIC int do_nop(dp, m_ptr)
18438   struct driver *dp;
18439   message *m_ptr;
18440   {
18441   /* Nothing there, or nothing to do. */
18442
18443     switch (m_ptr->m_type) {
18444     case DEV_OPEN:        return(ENODEV);
18445     case DEV_CLOSE:       return(OK);
18446     case DEV_IOCTL:       return(ENOTTY);
18447     default:              return(EIO);
18448     }
18449   }


18452   /*=======================================================================*
18453    *                              nop_finish                               *
18454    *=======================================================================*/
18455   PUBLIC int nop_finish()
18456   {
18457   /* Nothing to finish, all the work has been done by dp->dr_schedule. */
18458     return(OK);
18459   }


18462   /*=======================================================================*
18463    *                              nop_cleanup                              *
18464    *=======================================================================*/
18465   PUBLIC void nop_cleanup()
18466   {
18467   /* Nothing to clean up. */
18468   }


18471   /*=======================================================================*
18472    *                              clock_mess                               *
18473    *=======================================================================*/
18474   PUBLIC void clock_mess(ticks, func)
18475   int ticks;                    /* how many clock ticks to wait */
18476   watchdog_t func;              /* function to call upon time out */
18477   {
18478   /* Send the clock task a message. */
18479
18480     message mess;
18481
18482     mess.m_type = SET_ALARM;
18483     mess.CLOCK_PROC_NR = proc_number(proc_ptr);
18484     mess.DELTA_TICKS = (long) ticks;
18485     mess.FUNC_TO_CALL = (sighandler_t) func;
18486     sendrec(CLOCK, &mess);
18487   }


18490   /*=======================================================================*
18491    *                              do_diocntl                               *
18492    *=======================================================================*/
18493   PUBLIC int do_diocntl(dp, m_ptr)
18494   struct driver *dp;
```

```
18495   message *m_ptr;                 /* pointer to ioctl request */
18496   {
18497   /* Carry out a partition setting/getting request. */
18498     struct device *dv;
18499     phys_bytes user_phys, entry_phys;
18500     struct partition entry;
18501
18502     if (m_ptr->REQUEST != DIOCSETP && m_ptr->REQUEST != DIOCGETP) return(ENOTTY);
18503
18504     /* Decode the message parameters. */
18505     if ((dv = (*dp->dr_prepare)(m_ptr->DEVICE)) == NIL_DEV) return(ENXIO);
18506
18507     user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, sizeof(entry));
18508     if (user_phys == 0) return(EFAULT);
18509
18510     entry_phys = vir2phys(&entry);
18511
18512     if (m_ptr->REQUEST == DIOCSETP) {
18513           /* Copy just this one partition table entry. */
18514           phys_copy(user_phys, entry_phys, (phys_bytes) sizeof(entry));
18515           dv->dv_base = entry.base;
18516           dv->dv_size = entry.size;
18517     } else {
18518           /* Return a partition table entry and the geometry of the drive. */
18519           entry.base = dv->dv_base;
18520           entry.size = dv->dv_size;
18521           (*dp->dr_geometry)(&entry);
18522           phys_copy(entry_phys, user_phys, (phys_bytes) sizeof(entry));
18523     }
18524     return(OK);
18525   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/drvlib.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

18600   /* IBM device driver utility functions.                Author: Kees J. Bot
18601    *                                                             7 Dec 1995
18602    * Entry point:
18603    *   partition: partition a disk to the partition table(s) on it.
18604    */
18605
18606   #include "kernel.h"
18607   #include "driver.h"
18608   #include "drvlib.h"
18609
18610
18611   FORWARD _PROTOTYPE( void extpartition, (struct driver *dp, int extdev,
18612                                                 unsigned long extbase) );
18613   FORWARD _PROTOTYPE( int get_part_table, (struct driver *dp, int device,
18614                           unsigned long offset, struct part_entry *table) );
18615   FORWARD _PROTOTYPE( void sort, (struct part_entry *table) );
18616
18617
18618   /*============================================================================*
18619    *                              partition                                     *
```

```
18620  *===========================================================================*/
18621 PUBLIC void partition(dp, device, style)
18622 struct driver *dp;      /* device dependent entry points */
18623 int device;             /* device to partition */
18624 int style;              /* partitioning style: floppy, primary, sub. */
18625 {
18626 /* This routine is called on first open to initialize the partition tables
18627  * of a device.  It makes sure that each partition falls safely within the
18628  * device's limits.  Depending on the partition style we are either making
18629  * floppy partitions, primary partitions or subpartitions.  Only primary
18630  * partitions are sorted, because they are shared with other operating
18631  * systems that expect this.
18632  */
18633   struct part_entry table[NR_PARTITIONS], *pe;
18634   int disk, par;
18635   struct device *dv;
18636   unsigned long base, limit, part_limit;
18637
18638   /* Get the geometry of the device to partition */
18639   if ((dv = (*dp->dr_prepare)(device)) == NIL_DEV || dv->dv_size == 0) return;
18640   base = dv->dv_base >> SECTOR_SHIFT;
18641   limit = base + (dv->dv_size >> SECTOR_SHIFT);
18642
18643   /* Read the partition table for the device. */
18644   if (!get_part_table(dp, device, 0L, table)) return;
18645
18646   /* Compute the device number of the first partition. */
18647   switch (style) {
18648   case P_FLOPPY:
18649         device += MINOR_fd0a;
18650         break;
18651   case P_PRIMARY:
18652         sort(table);            /* sort a primary partition table */
18653         device += 1;
18654         break;
18655   case P_SUB:
18656         disk = device / DEV_PER_DRIVE;
18657         par = device % DEV_PER_DRIVE - 1;
18658         device = MINOR_hd1a + (disk * NR_PARTITIONS + par) * NR_PARTITIONS;
18659   }
18660
18661   /* Find an array of devices. */
18662   if ((dv = (*dp->dr_prepare)(device)) == NIL_DEV) return;
18663
18664   /* Set the geometry of the partitions from the partition table. */
18665   for (par = 0; par < NR_PARTITIONS; par++, dv++) {
18666         /* Shrink the partition to fit within the device. */
18667         pe = &table[par];
18668         part_limit = pe->lowsec + pe->size;
18669         if (part_limit < pe->lowsec) part_limit = limit;
18670         if (part_limit > limit) part_limit = limit;
18671         if (pe->lowsec < base) pe->lowsec = base;
18672         if (part_limit < pe->lowsec) part_limit = pe->lowsec;
18673
18674         dv->dv_base = pe->lowsec << SECTOR_SHIFT;
18675         dv->dv_size = (part_limit - pe->lowsec) << SECTOR_SHIFT;
18676
18677         if (style == P_PRIMARY) {
18678                 /* Each Minix primary partition can be subpartitioned. */
18679                 if (pe->sysind == MINIX_PART)
```

```
18680                           partition(dp, device + par, P_SUB);
18681
18682                   /* An extended partition has logical partitions. */
18683                   if (pe->sysind == EXT_PART)
18684                           extpartition(dp, device + par, pe->lowsec);
18685           }
18686     }
18687   }


18690   /*============================================================================*
18691    *                               extpartition                                 *
18692    *============================================================================*/
18693   PRIVATE void extpartition(dp, extdev, extbase)
18694   struct driver *dp;      /* device dependent entry points */
18695   int extdev;             /* extended partition to scan */
18696   unsigned long extbase;  /* sector offset of the base extended partition */
18697   {
18698   /* Extended partitions cannot be ignored alas, because people like to move
18699    * files to and from DOS partitions.  Avoid reading this code, it's no fun.
18700    */
18701     struct part_entry table[NR_PARTITIONS], *pe;
18702     int subdev, disk, par;
18703     struct device *dv;
18704     unsigned long offset, nextoffset;
18705
18706     disk = extdev / DEV_PER_DRIVE;
18707     par = extdev % DEV_PER_DRIVE - 1;
18708     subdev = MINOR_hd1a + (disk * NR_PARTITIONS + par) * NR_PARTITIONS;
18709
18710     offset = 0;
18711     do {
18712           if (!get_part_table(dp, extdev, offset, table)) return;
18713           sort(table);
18714
18715           /* The table should contain one logical partition and optionally
18716            * another extended partition.  (It's a linked list.)
18717            */
18718           nextoffset = 0;
18719           for (par = 0; par < NR_PARTITIONS; par++) {
18720                   pe = &table[par];
18721                   if (pe->sysind == EXT_PART) {
18722                           nextoffset = pe->lowsec;
18723                   } else
18724                   if (pe->sysind != NO_PART) {
18725                           if ((dv = (*dp->dr_prepare)(subdev)) == NIL_DEV) return;
18726
18727                           dv->dv_base = (extbase + offset
18728                                           + pe->lowsec) << SECTOR_SHIFT;
18729                           dv->dv_size = pe->size << SECTOR_SHIFT;
18730
18731                           /* Out of devices? */
18732                           if (++subdev % NR_PARTITIONS == 0) return;
18733                   }
18734           }
18735     } while ((offset = nextoffset) != 0);
18736   }


18739   /*============================================================================*
```

```
18740    *                             get_part_table                                *
18741    *===========================================================================*/
18742   PRIVATE int get_part_table(dp, device, offset, table)
18743   struct driver *dp;
18744   int device;
18745   unsigned long offset;           /* sector offset to the table */
18746   struct part_entry *table;       /* four entries */
18747   {
18748   /* Read the partition table for the device, return true iff there were no
18749    * errors.
18750    */
18751     message mess;
18752
18753     mess.DEVICE = device;
18754     mess.POSITION = offset << SECTOR_SHIFT;
18755     mess.COUNT = SECTOR_SIZE;
18756     mess.ADDRESS = (char *) tmp_buf;
18757     mess.PROC_NR = proc_number(proc_ptr);
18758     mess.m_type = DEV_READ;
18759
18760     if (do_rdwt(dp, &mess) != SECTOR_SIZE) {
18761           printf("%s: can't read partition table\n", (*dp->dr_name)());
18762           return 0;
18763     }
18764     if (tmp_buf[510] != 0x55 || tmp_buf[511] != 0xAA) {
18765           /* Invalid partition table. */
18766           return 0;
18767     }
18768     memcpy(table, (tmp_buf + PART_TABLE_OFF), NR_PARTITIONS * sizeof(table[0]));
18769     return 1;
18770   }


18773   /*===========================================================================*
18774    *                             sort                                          *
18775    *===========================================================================*/
18776   PRIVATE void sort(table)
18777   struct part_entry *table;
18778   {
18779   /* Sort a partition table. */
18780     struct part_entry *pe, tmp;
18781     int n = NR_PARTITIONS;
18782
18783     do {
18784           for (pe = table; pe < table + NR_PARTITIONS-1; pe++) {
18785                 if (pe[0].sysind == NO_PART
18786                         || (pe[0].lowsec > pe[1].lowsec
18787                                       && pe[1].sysind != NO_PART)) {
18788                       tmp = pe[0]; pe[0] = pe[1]; pe[1] = tmp;
18789                 }
18790           }
18791     } while (--n > 0);
18792   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/esdi_wini.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

18800    /*     device dependent part of a hard disk driver for ibm ps/2 esdi adapter
18801     *
18802     *     written by doug burks, based on other minix wini drivers.
18803     *     some additions by art roberts
18804     *
18805     *     references:
18806     *        ibm personal system/2 hardware technical reference  (1988)
18807     *        ibm 60/120mb fixed disk drive technical reference  (1988)
18808     *
18809     *     caveats:
18810     *       * this driver has been reported to work on ibm ps/2 models 50 and
18811     *         70 with ibm's 60/120mb hard disks.
18812     *       * for a true esdi adapter, changes will have to be made, but this
18813     *         certainly serves as a good start.
18814     *       * no timeouts are implemented, so this driver could hang under
18815     *         adverse conditions.
18816     *       * the error processing has not been tested.  my disk works too well.
18817     *
18818     * The file contains one entry point:
18819     *
18820     *   esdi_winchester_task:      main entry when system is brought up
18821     *
18822     *
18823     * Changes:
18824     *       3 May 1992 by Kees J. Bot: device dependent/independent split.
18825     */
18826    #include "kernel.h"
18827    #include "driver.h"
18828    #include "drvlib.h"
18829
18830    #if ENABLE_ESDI_WINI
18831
18832    /* If the DMA buffer is large enough then use it always. */
18833    #define USE_BUF         (DMA_BUF_SIZE > BLOCK_SIZE)
18834
18835
18836    /*****  esdi i/o adapter ports  */
18837
18838    #define  CMD_REG   0x3510       /* command interface register           */
18839    #define  STAT_REG  0x3510       /* status interface register            */
18840    #define  BCTL_REG  0x3512       /* basic control register               */
18841    #define  BST_REG   0x3512       /* basic status register                */
18842    #define  ATT_REG   0x3513       /* attention register                   */
18843    #define  INT_REG   0x3513       /* interrupt status register            */
18844
18845
18846    /*****  basic status register bits  */
18847
18848    #define  DMA_ENA   0x80         /* DMA enabled?                         */
18849    #define  INT_PND   0x40         /* interrupt pending?                   */
18850    #define  CMD_PRG   0x20         /* command in progress?                 */
18851    #define  BUSY      0x10         /* is adapter busy?                     */
18852    #define  STR_FUL   0x08         /* status interface register set?       */
18853    #define  CMD_FUL   0x04         /* command interface register full?     */
18854    #define  XFR_REQ   0x02         /* data transfer operation ready?       */
```

```
18855   #define  INT_SET   0x01         /* adapter sending interrupt?           */
18856
18857
18858   /*****  attention register commands  */
18859
18860   #define  ATT_CMD   0x01         /* command request                      */
18861   #define  ATT_EOI   0x02         /* end of interrupt processing          */
18862   #define  ATT_ABT   0x03         /* abort the current command            */
18863   #define  ATT_RST   0xE4         /* reset the esdi adapter               */
18864
18865
18866   /*****  dma register addresses  */
18867
18868   #define  DMA_EXTCMD  0x18       /* extended function register           */
18869   #define  DMA_EXEC    0x1A       /* extended function execute            */
18870
18871
18872   /*****  miscellaneous  */
18873
18874   #define  ERR            (-1)    /* general error code                   */
18875   #define  ERR_BAD_SECTOR (-2)    /* block marked bad detected            */
18876   #define  MAX_ERRORS     4       /* maximum number of read/write retries  */
18877   #define  MAX_DRIVES     2       /* maximum number of physical drives     */
18878   #define  NR_DEVICES     (MAX_DRIVES*DEV_PER_DRIVE)
18879                                  /* Maximum number of logical devices     */
18880   #define  SUB_PER_DRIVE  (NR_PARTITIONS * NR_PARTITIONS)
18881   #define  NR_SUBDEVS     (MAX_DRIVES * SUB_PER_DRIVE)
18882
18883   #define  SYS_PORTA      0x92    /* system control port a                */
18884   #define  LIGHT_ON       0xC0    /* fixed-disk activity light reg. mask   */
18885
18886
18887   /*****  variables  */
18888
18889   PRIVATE struct wini {           /* disk/partition information           */
18890     unsigned open_ct;                     /* in-use count                 */
18891     struct device part[DEV_PER_DRIVE];    /* primary partitions: hd[0-4]  */
18892     struct device subpart[SUB_PER_DRIVE]; /* subpartitions: hd[1-4][a-d] */
18893   } wini[MAX_DRIVES], *w_wn;
18894
18895   PRIVATE struct trans {
18896     struct iorequest_s *iop;      /* belongs to this I/O request          */
18897     unsigned long block;          /* first sector to transfer             */
18898     unsigned count;               /* byte count                           */
18899     phys_bytes phys;              /* user physical address                */
18900     phys_bytes dma;               /* DMA physical address                 */
18901   } wtrans[NR_IOREQS];
18902
18903   PRIVATE int nr_drives;          /* actual number of physical disk drive  */
18904   PRIVATE int command[4];         /* controller command buffer            */
18905   PRIVATE unsigned int status_block[9];   /* status block output from a command */
18906   PRIVATE int dma_channel;        /* fixed disk dma channel number        */
18907   PRIVATE struct trans *w_tp;     /* to add transfer requests             */
18908   PRIVATE unsigned w_count;       /* number of bytes to transfer          */
18909   PRIVATE unsigned long w_nextblock;      /* next block on disk to transfer */
18910   PRIVATE int w_opcode;           /* DEV_READ or DEV_WRITE                */
18911   PRIVATE int w_drive;            /* selected drive                       */
18912   PRIVATE int w_istat;            /* interrupt status of last command     */
18913   PRIVATE struct device *w_dv;    /* device's base and size               */
18914
```

```
18915
18916   /*****  functions  */
18917
18918   FORWARD _PROTOTYPE( struct device *w_prepare, (int device) );
18919   FORWARD _PROTOTYPE( char *w_name, (void) );
18920   FORWARD _PROTOTYPE( int w_do_open, (struct driver *dp, message *m_ptr) );
18921   FORWARD _PROTOTYPE( int w_do_close, (struct driver *dp, message *m_ptr) );
18922   FORWARD _PROTOTYPE( void w_init, (void) );
18923   FORWARD _PROTOTYPE( int w_command, (int device, int cmd, int num_words) );
18924   FORWARD _PROTOTYPE( int w_schedule, (int proc_nr, struct iorequest_s *iop) );
18925   FORWARD _PROTOTYPE( int w_finish, (void) );
18926   FORWARD _PROTOTYPE( int w_transfer, (struct trans *tp, unsigned count) );
18927   FORWARD _PROTOTYPE( int w_att_write, (int value) );
18928   FORWARD _PROTOTYPE( void w_interrupt, (int dma) );
18929   FORWARD _PROTOTYPE( int w_handler, (int irq) );
18930   FORWARD _PROTOTYPE( void w_dma_setup, (struct trans *tp, unsigned count) );
18931   FORWARD _PROTOTYPE( void w_geometry, (struct partition *entry));
18932
18933
18934   /* Entry points to this driver. */
18935   PRIVATE struct driver w_dtab = {
18936     w_name,       /* current device's name */
18937     w_do_open,    /* open or mount request, initialize device */
18938     w_do_close,   /* release device */
18939     do_diocntl,   /* get or set a partition's geometry */
18940     w_prepare,    /* prepare for I/O on a given minor device */
18941     w_schedule,   /* precompute cylinder, head, sector, etc. */
18942     w_finish,     /* do the I/O */
18943     nop_cleanup,  /* no cleanup needed */
18944     w_geometry    /* tell the geometry of the disk */
18945   };
18946
18947
18948   /*===========================================================================*
18949    *                               esdi_winchester_task                        *
18950    *===========================================================================*/
18951   PUBLIC void esdi_winchester_task()
18952   {
18953     driver_task(&w_dtab);
18954   }


18957   /*===========================================================================*
18958    *                               w_prepare                                   *
18959    *===========================================================================*/
18960   PRIVATE struct device *w_prepare(device)
18961   int device;
18962   {
18963   /* Prepare for I/O on a device. */
18964
18965     /* Nothing to transfer as yet. */
18966     w_count = 0;
18967
18968     if (device < NR_DEVICES) {                    /* hd0, hd1, ... */
18969           w_drive = device / DEV_PER_DRIVE;       /* save drive number */
18970           w_wn = &wini[w_drive];
18971           w_dv = &w_wn->part[device % DEV_PER_DRIVE];
18972     } else
18973     if ((unsigned) (device -= MINOR_hd1a) < NR_SUBDEVS) { /* hd1a, hd1b, ... */
18974           w_drive = device / SUB_PER_DRIVE;
```

```
18975           w_wn = &wini[w_drive];
18976           w_dv = &w_wn->subpart[device % SUB_PER_DRIVE];
18977     } else {
18978           return(NIL_DEV);
18979     }
18980     return(w_drive < nr_drives ? w_dv : NIL_DEV);
18981   }


18984   /*===========================================================================*
18985    *                              w_name                                       *
18986    *===========================================================================*/
18987   PRIVATE char *w_name()
18988   {
18989   /* Return a name for the current device. */
18990     static char name[] = "esdi-hd5";
18991
18992     name[7] = '0' + w_drive * DEV_PER_DRIVE;
18993     return name;
18994   }


18997   /*===========================================================================*
18998    *                              w_do_open                                    *
18999    *===========================================================================*/
19000   PRIVATE int w_do_open(dp, m_ptr)
19001   struct driver *dp;
19002   message *m_ptr;
19003   {
19004   /* Device open: Initialize the controller and read the partition table. */
19005
19006     static int init_done = FALSE;
19007
19008     if (!init_done) { w_init(); init_done = TRUE; }
19009
19010     if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
19011
19012     if (w_wn->open_ct++ == 0) {
19013           /* partition the disk */
19014           partition(&w_dtab, w_drive * DEV_PER_DRIVE, P_PRIMARY);
19015     }
19016     return(OK);
19017   }


19020   /*===========================================================================*
19021    *                              w_do_close                                   *
19022    *===========================================================================*/
19023   PRIVATE int w_do_close(dp, m_ptr)
19024   struct driver *dp;
19025   message *m_ptr;
19026   {
19027   /* Device close: Release a device. */
19028
19029     if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
19030     w_wn->open_ct--;
19031     return(OK);
19032   }
```

```
19035   /*==========================================================================*
19036    *                              w_init                                      *
19037    *==========================================================================*/
19038   PRIVATE void w_init()
19039   {
19040   /*     initializes everything needed to run the hard disk
19041    *
19042    *     the following items are initialized:
19043    *       -- hard disk attributes stored in bios
19044    *       -- dma transfer channel, read from system register
19045    *       -- dma transfer and interrupts [disabled]
19046    *
19047    *     the hard disk adapter is initialized when the ibm ps/2 is turned on,
19048    *     using the programmable option select registers.  thus the only
19049    *     additional initialization is making sure the dma transfer and interrupts
19050    *     are disabled.  other initialization problems could be checked for, such
19051    *     as an operation underway.  the paranoid could add a check for adapter
19052    *     activity and abort the operations.  the truly paranoid can reset the
19053    *     adapter.  until such worries are proven, why bother?
19054    */
19055     unsigned int drive;           /* hard disk drive number               */
19056     unsigned long size;           /* hard disk size                       */
19057
19058     /* get the number of drives from the bios */
19059     phys_copy(0x475L, tmp_phys, 1L);
19060     nr_drives = tmp_buf[0];
19061     if (nr_drives > MAX_DRIVES) nr_drives = MAX_DRIVES;
19062
19063     put_irq_handler(AT_WINI_IRQ, w_handler);
19064     enable_irq(AT_WINI_IRQ);      /* ready for winchester interrupts */
19065
19066     for (drive = 0; drive < nr_drives; ++drive) {
19067           (void) w_prepare(drive * DEV_PER_DRIVE);
19068           if (w_command(drive, 0x0609, 6) != OK) {
19069                   printf("%s: unable to get parameters\n", w_name());
19070                   nr_drives = drive;
19071                   break;
19072           }
19073           /* size of the drive */
19074           size =  ((unsigned long) status_block[2] <<  0) |
19075                   ((unsigned long) status_block[3] << 16);
19076           if (drive == 0) {
19077                   if (w_command(7, 0x060A, 5) != OK) {
19078                           printf("%s: unable to get dma channel\n", w_name());
19079                           nr_drives = 0;
19080                           return;
19081                   }
19082                   dma_channel = (status_block[2] & 0x3C00) >> 10;
19083           }
19084           printf("%s: %lu sectors\n", w_name(), size);
19085           w_wn->part[0].dv_size = size << SECTOR_SHIFT;
19086     }
19087   }


19090   /*==========================================================================*
19091    *                            w_command                                     *
19092    *==========================================================================*/
19093   PRIVATE int w_command(device, cmd, num_words)
19094   int device;                   /* i device to operate on               */
```

```
19095                                   /*   1-2   physical disk drive number    */
19096                                   /*   7     hard disk controller          */
19097   int cmd;                        /* i command to execute                  */
19098   int num_words;                  /* i expected size of status block       */
19099   {
19100   /*     executes a command for a particular device
19101    *
19102    *     the operation is conducted as follows:
19103    *       -- create the command block
19104    *       -- initialize for command reading by the controller
19105    *       -- write the command block to the controller, making sure the
19106    *          controller has digested the previous command word, before shoving
19107    *          the next down its throat
19108    *       -- wait for an interrupt
19109    *       -- read expected number of words of command status information
19110    *       -- return the command status block
19111    *
19112    *     reading and writing registers is accompanied by enabling and disabling
19113    *     interrupts to ensure that the status register contents still apply when
19114    *     the desired register is read/written.
19115    */
19116     register int ki;              /* -- scratch --                          */
19117     int status;                   /* disk adapter status register value    */
19118
19119     device <<= 5;                 /* adjust device for our use             */
19120     command[0] = cmd | device;    /* build command block                   */
19121     command[1] = 0;
19122
19123     w_att_write(device | ATT_CMD);
19124
19125     for (ki = 0; ki < 2; ++ki) {
19126           out_word(CMD_REG, command[ki]);
19127           unlock();
19128           while (TRUE) {
19129                   lock();
19130                   status = in_byte(BST_REG);
19131                   if (!(status & CMD_FUL)) break;
19132                   unlock();
19133           }
19134     }
19135     unlock();
19136
19137     w_interrupt(0);
19138     if (w_istat != (device | 0x01)) {
19139           w_att_write(device | ATT_ABT);
19140           w_interrupt(0);
19141           return(ERR);
19142     }
19143     for (ki = 0; ki < num_words; ++ki) {
19144           while (TRUE) {
19145                   lock();
19146                   status = in_byte(BST_REG);
19147                   if (status & STR_FUL) break;
19148                   unlock();
19149           }
19150           status_block[ki] = in_word(STAT_REG);
19151           unlock();
19152     }
19153     w_att_write(device | ATT_EOI);
19154
```

```
19155       return(OK);
19156     }


19159     /*============================================================================*
19160      *                              w_schedule                                    *
19161      *============================================================================*/
19162     PRIVATE int w_schedule(proc_nr, iop)
19163     int proc_nr;                    /* process doing the request */
19164     struct iorequest_s *iop;        /* pointer to read or write request */
19165     {
19166     /* Gather I/O requests on consecutive blocks so they may be read/written
19167      * in one command if using a buffer.  Check and gather all the requests
19168      * and try to finish them as fast as possible if unbuffered.
19169      */
19170       int r, opcode;
19171       unsigned long pos;
19172       unsigned nbytes, count;
19173       unsigned long block;
19174       phys_bytes user_phys, dma_phys;
19175
19176       /* This many bytes to read/write */
19177       nbytes = iop->io_nbytes;
19178       if ((nbytes & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);
19179
19180       /* From/to this position on the device */
19181       pos = iop->io_position;
19182       if ((pos & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);
19183
19184       /* To/from this user address */
19185       user_phys = numap(proc_nr, (vir_bytes) iop->io_buf, nbytes);
19186       if (user_phys == 0) return(iop->io_nbytes = EINVAL);
19187
19188       /* Read or write? */
19189       opcode = iop->io_request & ~OPTIONAL_IO;
19190
19191       /* Which block on disk and how close to EOF? */
19192       if (pos >= w_dv->dv_size) return(OK);         /* At EOF */
19193       if (pos + nbytes > w_dv->dv_size) nbytes = w_dv->dv_size - pos;
19194       block = (w_dv->dv_base + pos) >> SECTOR_SHIFT;
19195
19196       if (USE_BUF && w_count > 0 && block != w_nextblock) {
19197             /* This new request can't be chained to the job being built */
19198             if ((r = w_finish()) != OK) return(r);
19199       }
19200
19201       /* The next consecutive block */
19202       if (USE_BUF) w_nextblock = block + (nbytes >> SECTOR_SHIFT);
19203
19204       /* While there are "unscheduled" bytes in the request: */
19205       do {
19206             count = nbytes;
19207
19208             if (USE_BUF) {
19209                     if (w_count == DMA_BUF_SIZE) {
19210                             /* Can't transfer more than the buffer allows. */
19211                             if ((r = w_finish()) != OK) return(r);
19212                     }
19213
19214                     if (w_count + count > DMA_BUF_SIZE)
```

```
19215                           count = DMA_BUF_SIZE - w_count;
19216           } else {
19217                   if (w_tp == wtrans + NR_IOREQS) {
19218                           /* All transfer slots in use. */
19219                           if ((r = w_finish()) != OK) return(r);
19220                   }
19221           }
19222
19223           if (w_count == 0) {
19224                   /* The first request in a row, initialize. */
19225                   w_opcode = opcode;
19226                   w_tp = wtrans;
19227           }
19228
19229           if (USE_BUF) {
19230                   dma_phys = tmp_phys + w_count;
19231           } else {
19232                   /* Note: No 64K boundary problem, the better PS/2's have a
19233                    * working DMA chip.
19234                    */
19235                   dma_phys = user_phys;
19236           }
19237
19238           /* Store I/O parameters */
19239           w_tp->iop = iop;
19240           w_tp->block = block;
19241           w_tp->count = count;
19242           w_tp->phys = user_phys;
19243           w_tp->dma = dma_phys;
19244
19245           /* Update counters */
19246           w_tp++;
19247           w_count += count;
19248           block += count >> SECTOR_SHIFT;
19249           user_phys += count;
19250           nbytes -= count;
19251     } while (nbytes > 0);
19252
19253     return(OK);
19254   }


19257   /*============================================================================*
19258    *                              w_finish                                      *
19259    *============================================================================*/
19260   PRIVATE int w_finish()
19261   {
19262   /*     carries out the I/O requests gathered in wtrans[]
19263    *
19264    *     fills the disk information structure for one block at a time or many
19265    *     in a row before calling 'w_transfer' to do the dirty work.  while
19266    *     unsuccessful operations are re-tried, this may be superfluous, since
19267    *     the controller does the same on its own.  turns on the fixed disk
19268    *     activity light, while busy.  computers need blinking lights, right?
19269    */
19270
19271     struct trans *tp = wtrans, *tp2;
19272     unsigned count;
19273     int r, errors = 0, many = USE_BUF;
19274
```

```
19275     if (w_count == 0) return(OK); /* Spurious finish. */
19276
19277     do {
19278           if (w_opcode == DEV_WRITE) {
19279                   tp2 = tp;
19280                   count = 0;
19281                   do {
19282                           if (USE_BUF || tp2->dma == tmp_phys) {
19283                                   phys_copy(tp2->phys, tp2->dma,
19284                                                   (phys_bytes) tp2->count);
19285                           }
19286                           count += tp2->count;
19287                           tp2++;
19288                   } while (many && count < w_count);
19289           } else {
19290                   count = many ? w_count : tp->count;
19291           }
19292
19293           /* Turn on the disk activity light. */
19294           out_byte(SYS_PORTA, in_byte(SYS_PORTA) | LIGHT_ON);
19295
19296           /* Perform the transfer. */
19297           r = w_transfer(tp, count);
19298
19299           /* Turn off the disk activity light. */
19300           out_byte(SYS_PORTA, in_byte(SYS_PORTA) & ~LIGHT_ON);
19301
19302           if (r != OK) {
19303                   /* An error occurred, try again block by block unless */
19304                   if (r == ERR_BAD_SECTOR || ++errors == MAX_ERRORS)
19305                           return(tp->iop->io_nbytes = EIO);
19306
19307                   many = 0;
19308                   continue;
19309           }
19310           errors = 0;
19311
19312           w_count -= count;
19313
19314           do {
19315                   if (w_opcode == DEV_READ) {
19316                           if (USE_BUF || tp->dma == tmp_phys) {
19317                                   phys_copy(tp->dma, tp->phys,
19318                                                   (phys_bytes) tp->count);
19319                           }
19320                   }
19321                   tp->iop->io_nbytes -= tp->count;
19322                   count -= tp->count;
19323                   tp++;
19324           } while (count > 0);
19325     } while (w_count > 0);
19326
19327     return(OK);
19328   }


19331   /*===========================================================================*
19332    *                              w_transfer                                   *
19333    *===========================================================================*/
19334   PRIVATE int w_transfer(tp, count)
```

```
19335   struct trans *tp;               /* pointer to the transfer struct */
19336   unsigned int count;             /* bytes to transfer */
19337   {
19338   /*     reads/writes a single block of data from/to the hard disk
19339    *
19340    *     the read/write operation performs the following steps:
19341    *       -- create the command block
19342    *       -- initialize the command reading by the controller
19343    *       -- write the command block to the controller, making sure the
19344    *            controller has digested the previous command word, before
19345    *            shoving the next down its throat.
19346    *       -- wait for an interrupt, which must return a 'data transfer ready'
19347    *            status.  abort the command if it doesn't.
19348    *       -- set up and start up the direct memory transfer
19349    *       -- wait for an interrupt, signalling the end of the transfer
19350    */
19351     int device;                   /* device mask for the command register  */
19352     int ki;                       /* -- scratch --                          */
19353     int status;                   /* basic status register value           */
19354
19355     device = w_drive << 5;
19356     command[0] = (w_opcode == DEV_WRITE ? 0x4602 : 0x4601) | device;
19357     command[1] = count >> SECTOR_SHIFT;
19358     command[2] = (int) (tp->block & 0xFFFF);
19359     command[3] = (int) (tp->block >> 16);
19360
19361     w_att_write(device | ATT_CMD);
19362
19363     for (ki = 0; ki < 4; ++ki) {
19364           out_word(CMD_REG, command[ki]);
19365           unlock();
19366           while (TRUE) {
19367                   lock();
19368                   status = in_byte(BST_REG);
19369                   if (!(status & CMD_FUL)) break;
19370                   unlock();
19371           }
19372     }
19373     unlock();
19374
19375     w_interrupt(0);
19376     if (w_istat != (device | 0x0B)) {
19377           w_att_write(device | ATT_ABT);
19378           w_interrupt(0);
19379           return(ERR);
19380     }
19381     w_dma_setup(tp, count);
19382
19383     w_interrupt(1);
19384
19385     w_att_write(device | ATT_EOI);
19386
19387     if ((w_istat & 0x0F) > 8) return(ERR);
19388     return(OK);
19389   }



19393   /*============================================================================*
19394    *                           w_att_write                                      *
```

```
19395     *===========================================================================*/
19396    PRIVATE int w_att_write(value)
19397    register int value;
19398    {
19399    /*     writes a command to the esdi attention register
19400     *
19401     *     waits for the controller to finish its business before sending the
19402     *     command to the controller.  note that the interrupts must be off to read
19403     *     the basic status register and, if the controller is ready, must not be
19404     *     turned back on until the attention register command is sent.
19405     */
19406      int status;                   /* basic status register value           */
19407
19408      while (TRUE) {
19409            lock();
19410            status = in_byte(BST_REG);
19411            if (!(status & (INT_PND | BUSY))) break;
19412            unlock();
19413      }
19414      out_byte(ATT_REG, value);
19415      unlock();
19416
19417      return(OK);
19418    }



19422    /*===========================================================================*
19423     *                           w_interrupt                                     *
19424     *===========================================================================*/
19425    PRIVATE void w_interrupt(dma)
19426    int dma;                      /* i dma transfer is underway            */
19427    {
19428    /*     waits for an interrupt from the hard disk controller
19429     *
19430     *     enable interrupts on the hard disk and interrupt controllers (and dma if
19431     *     necessary).  wait for an interrupt.  when received, return the interrupt
19432     *     status register value.
19433     *
19434     *     an interrupt can be detected either from the basic status register or
19435     *     through a system interrupt handler.  the handler is used for all
19436     *     interrupts, due to the expected long times to process reads and writes
19437     *     and to avoid busy waits.
19438     */
19439      message dummy;                /* -- scratch --                         */
19440
19441      out_byte(BCTL_REG, dma ? 0x03 : 0x01);
19442
19443      receive(HARDWARE, &dummy);
19444
19445      out_byte(BCTL_REG, 0);
19446      if (dma) out_byte(DMA_EXTCMD, 0x90 + dma_channel);
19447    }



19451    /*===========================================================================*
19452     *                           w_handler                                       *
19453     *===========================================================================*/
19454    PRIVATE int w_handler(irq)
```

```
19455   int irq;
19456   {
19457   /* Disk interrupt, send message to winchester task and reenable interrupts. */
19458
19459     w_istat = in_byte(INT_REG);
19460     interrupt(WINCHESTER);
19461     return 1;
19462   }



19466   /*============================================================================*
19467    *                       w_dma_setup                                          *
19468    *============================================================================*/
19469   PRIVATE void w_dma_setup(tp, count)
19470   struct trans *tp;
19471   unsigned int count;
19472   {
19473   /*     programs the dma controller to move data to and from the hard disk.
19474    *
19475    *     uses the extended mode operation of the ibm ps/2 interrupt controller
19476    *     chip, rather than the intel 8237 (pc/at) compatible mode.
19477    */
19478
19479     lock();
19480     out_byte(DMA_EXTCMD, 0x90 + dma_channel);
19481     /* Disable access to dma channel 5     */
19482     out_byte(DMA_EXTCMD, 0x20 + dma_channel);
19483     /* Clear the address byte pointer      */
19484     out_byte(DMA_EXEC, (int)  tp->dma >>  0);     /* address bits 0..7   */
19485     out_byte(DMA_EXEC, (int)  tp->dma >>  8);     /* address bits 8..15  */
19486     out_byte(DMA_EXEC, (int) (tp->dma >> 16));    /* address bits 16..19 */
19487     out_byte(DMA_EXTCMD, 0x40 + dma_channel);
19488     /* Clear the count byte pointer        */
19489     out_byte(DMA_EXEC, (count - 1) >> 0);         /* count bits 0..7     */
19490     out_byte(DMA_EXEC, (count - 1) >> 8);         /* count bits 8..15    */
19491     out_byte(DMA_EXTCMD, 0x70 + dma_channel);
19492     /* Set the transfer mode               */
19493     out_byte(DMA_EXEC, w_opcode == DEV_WRITE ? 0x44 : 0x4C);
19494     out_byte(DMA_EXTCMD, 0xA0 + dma_channel);
19495     /* Enable access to dma channel 5      */
19496     unlock();
19497   }


19500   /*============================================================================*
19501    *                              w_geometry                                    *
19502    *============================================================================*/
19503   PRIVATE void w_geometry(entry)
19504   struct partition *entry;
19505   {
19506     entry->cylinders = (w_wn->part[0].dv_size >> SECTOR_SHIFT) / (64 * 32);
19507     entry->heads = 64;
19508     entry->sectors = 32;
19509   }
19510   #endif /* ENABLE_ESDI_WINI */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/exception.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

19600    /* This file contains a simple exception handler.  Exceptions in user
19601     * processes are converted to signals.  Exceptions in the kernel, MM and
19602     * FS cause a panic.
19603     */
19604
19605    #include "kernel.h"
19606    #include <signal.h>
19607    #include "proc.h"
19608
19609    /*===========================================================================*
19610     *                                exception                                  *
19611     *===========================================================================*/
19612    PUBLIC void exception(vec_nr)
19613    unsigned vec_nr;
19614    {
19615    /* An exception or unexpected interrupt has occurred. */
19616
19617      struct ex_s {
19618            char *msg;
19619            int signum;
19620            int minprocessor;
19621      };
19622      static struct ex_s ex_data[] = {
19623            "Divide error", SIGFPE, 86,
19624            "Debug exception", SIGTRAP, 86,
19625            "Nonmaskable interrupt", SIGBUS, 86,
19626            "Breakpoint", SIGEMT, 86,
19627            "Overflow", SIGFPE, 86,
19628            "Bounds check", SIGFPE, 186,
19629            "Invalid opcode", SIGILL, 186,
19630            "Coprocessor not available", SIGFPE, 186,
19631            "Double fault", SIGBUS, 286,
19632            "Copressor segment overrun", SIGSEGV, 286,
19633            "Invalid TSS", SIGSEGV, 286,
19634            "Segment not present", SIGSEGV, 286,
19635            "Stack exception", SIGSEGV, 286,        /* STACK_FAULT already used */
19636            "General protection", SIGSEGV, 286,
19637            "Page fault", SIGSEGV, 386,             /* not close */
19638            NIL_PTR, SIGILL, 0,                     /* probably software trap */
19639            "Coprocessor error", SIGFPE, 386,
19640      };
19641      register struct ex_s *ep;
19642      struct proc *saved_proc;
19643
19644      saved_proc= proc_ptr; /* Save proc_ptr, because it may be changed by debug
19645                             * statements.
19646                             */
19647
19648      ep = &ex_data[vec_nr];
19649
19650      if (vec_nr == 2) {            /* spurious NMI on some machines */
19651            printf("got spurious NMI\n");
19652            return;
19653      }
19654
```

```
19655      if (k_reenter == 0 && isuserp(saved_proc)) {
19656            unlock();               /* this is protected like sys_call() */
19657            cause_sig(proc_number(saved_proc), ep->signum);
19658            return;
19659      }
19660
19661      /* This is not supposed to happen. */
19662      if (ep->msg == NIL_PTR || processor < ep->minprocessor)
19663            printf("\nIntel-reserved exception %d\n", vec_nr);
19664      else
19665            printf("\n%s\n", ep->msg);
19666      printf("process number %d, pc = 0x%04x:0x%08x\n",
19667            proc_number(saved_proc),
19668            (unsigned) saved_proc->p_reg.cs,
19669            (unsigned) saved_proc->p_reg.pc);
19670      panic("exception in system code", NO_NUM);
19671    }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/floppy.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
19700    /* This file contains the device dependent part of the driver for the Floppy
19701     * Disk Controller (FDC) using the NEC PD765 chip.
19702     *
19703     * The file contains one entry point:
19704     *
19705     *   floppy_task:       main entry when system is brought up
19706     *   floppy_stop:       stop all activity
19707     *
19708     *  Changes:
19709     *      27 Oct. 1986 by Jakob Schripsema: fdc_results fixed for 8 MHz
19710     *      28 Nov. 1986 by Peter Kay: better resetting for 386
19711     *      06 Jan. 1988 by Al Crew: allow 1.44 MB diskettes
19712     *              1989 by Bruce Evans: I/O vector to keep up with 1-1 interleave
19713     *      13 May  1991 by Don Chapman: renovated the errors loop.
19714     *              1991 by Bruce Evans: len[] / motors / reset / step rate / ...
19715     *      14 Feb  1992 by Andy Tanenbaum: check drive density on opens only
19716     *      27 Mar  1992 by Kees J. Bot: last details on density checking
19717     *      04 Apr  1992 by Kees J. Bot: device dependent/independent split
19718     */
19719
19720    #include "kernel.h"
19721    #include "driver.h"
19722    #include "drvlib.h"
19723    #include <ibm/diskparm.h>
19724
19725    /* I/O Ports used by floppy disk task. */
19726    #define DOR             0x3F2    /* motor drive control bits */
19727    #define FDC_STATUS      0x3F4    /* floppy disk controller status register */
19728    #define FDC_DATA        0x3F5    /* floppy disk controller data register */
19729    #define FDC_RATE        0x3F7    /* transfer rate register */
19730    #define DMA_ADDR        0x004    /* port for low 16 bits of DMA address */
19731    #define DMA_TOP         0x081    /* port for top 4 bits of 20-bit DMA addr */
19732    #define DMA_COUNT       0x005    /* port for DMA count (count =  bytes - 1) */
19733    #define DMA_FLIPFLOP    0x00C    /* DMA byte pointer flip-flop */
19734    #define DMA_MODE        0x00B    /* DMA mode port */
```

```
19735    #define DMA_INIT       0x00A    /* DMA init port */
19736    #define DMA_RESET_VAL   0x06
19737
19738    /* Status registers returned as result of operation. */
19739    #define ST0             0x00    /* status register 0 */
19740    #define ST1             0x01    /* status register 1 */
19741    #define ST2             0x02    /* status register 2 */
19742    #define ST3             0x00    /* status register 3 (return by DRIVE_SENSE) */
19743    #define ST_CYL          0x03    /* slot where controller reports cylinder */
19744    #define ST_HEAD         0x04    /* slot where controller reports head */
19745    #define ST_SEC          0x05    /* slot where controller reports sector */
19746    #define ST_PCN          0x01    /* slot where controller reports present cyl */
19747
19748    /* Fields within the I/O ports. */
19749    /* Main status register. */
19750    #define CTL_BUSY        0x10    /* bit is set when read or write in progress */
19751    #define DIRECTION       0x40    /* bit is set when reading data reg is valid */
19752    #define MASTER          0x80    /* bit is set when data reg can be accessed */
19753
19754    /* Digital output port (DOR). */
19755    #define MOTOR_SHIFT        4    /* high 4 bits control the motors in DOR */
19756    #define ENABLE_INT      0x0C    /* used for setting DOR port */
19757
19758    /* ST0. */
19759    #define ST0_BITS        0xF8    /* check top 5 bits of seek status */
19760    #define TRANS_ST0       0x00    /* top 5 bits of ST0 for READ/WRITE */
19761    #define SEEK_ST0        0x20    /* top 5 bits of ST0 for SEEK */
19762
19763    /* ST1. */
19764    #define BAD_SECTOR      0x05    /* if these bits are set in ST1, recalibrate */
19765    #define WRITE_PROTECT   0x02    /* bit is set if diskette is write protected */
19766
19767    /* ST2. */
19768    #define BAD_CYL         0x1F    /* if any of these bits are set, recalibrate */
19769
19770    /* ST3 (not used). */
19771    #define ST3_FAULT       0x80    /* if this bit is set, drive is sick */
19772    #define ST3_WR_PROTECT  0x40    /* set when diskette is write protected */
19773    #define ST3_READY       0x20    /* set when drive is ready */
19774
19775    /* Floppy disk controller command bytes. */
19776    #define FDC_SEEK        0x0F    /* command the drive to seek */
19777    #define FDC_READ        0xE6    /* command the drive to read */
19778    #define FDC_WRITE       0xC5    /* command the drive to write */
19779    #define FDC_SENSE       0x08    /* command the controller to tell its status */
19780    #define FDC_RECALIBRATE 0x07    /* command the drive to go to cyl 0 */
19781    #define FDC_SPECIFY     0x03    /* command the drive to accept params */
19782    #define FDC_READ_ID     0x4A    /* command the drive to read sector identity */
19783    #define FDC_FORMAT      0x4D    /* command the drive to format a track */
19784
19785    /* DMA channel commands. */
19786    #define DMA_READ        0x46    /* DMA read opcode */
19787    #define DMA_WRITE       0x4A    /* DMA write opcode */
19788
19789    /* Parameters for the disk drive. */
19790    #define HC_SIZE         2880    /* # sectors on largest legal disk (1.44MB) */
19791    #define NR_HEADS        0x02    /* two heads (i.e., two tracks/cylinder) */
19792    #define MAX_SECTORS       18    /* largest # sectors per track */
19793    #define DTL             0xFF    /* determines data length (sector size) */
19794    #define SPEC2           0x02    /* second parameter to SPECIFY */
```

```
19795   #define MOTOR_OFF       3*HZ    /* how long to wait before stopping motor */
19796   #define WAKEUP          2*HZ    /* timeout on I/O, FDC won't quit. */
19797
19798   /* Error codes */
19799   #define ERR_SEEK         (-1)   /* bad seek */
19800   #define ERR_TRANSFER     (-2)   /* bad transfer */
19801   #define ERR_STATUS       (-3)   /* something wrong when getting status */
19802   #define ERR_READ_ID      (-4)   /* bad read id */
19803   #define ERR_RECALIBRATE  (-5)   /* recalibrate didn't work properly */
19804   #define ERR_DRIVE        (-6)   /* something wrong with a drive */
19805   #define ERR_WR_PROTECT   (-7)   /* diskette is write protected */
19806   #define ERR_TIMEOUT      (-8)   /* interrupt timeout */
19807
19808   /* No retries on some errors. */
19809   #define err_no_retry(err)       ((err) <= ERR_WR_PROTECT)
19810
19811   /* Encoding of drive type in minor device number. */
19812   #define DEV_TYPE_BITS   0x7C    /* drive type + 1, if nonzero */
19813   #define DEV_TYPE_SHIFT     2    /* right shift to normalize type bits */
19814   #define FORMAT_DEV_BIT  0x80    /* bit in minor to turn write into format */
19815
19816   /* Miscellaneous. */
19817   #define MAX_ERRORS         6    /* how often to try rd/wt before quitting */
19818   #define MAX_RESULTS        7    /* max number of bytes controller returns */
19819   #define NR_DRIVES          2    /* maximum number of drives */
19820   #define DIVISOR          128    /* used for sector size encoding */
19821   #define SECTOR_SIZE_CODE   2    /* code to say "512" to the controller */
19822   #define TIMEOUT          500    /* milliseconds waiting for FDC */
19823   #define NT                 7    /* number of diskette/drive combinations */
19824   #define UNCALIBRATED       0    /* drive needs to be calibrated at next use */
19825   #define CALIBRATED         1    /* no calibration needed */
19826   #define BASE_SECTOR        1    /* sectors are numbered starting at 1 */
19827   #define NO_SECTOR          0    /* current sector unknown */
19828   #define NO_CYL           (-1)   /* current cylinder unknown, must seek */
19829   #define NO_DENS          100    /* current media unknown */
19830   #define BSY_IDLE           0    /* busy doing nothing */
19831   #define BSY_IO             1    /* doing I/O */
19832   #define BSY_WAKEN          2    /* got a wakeup call */
19833
19834   /* Variables. */
19835   PRIVATE struct floppy {         /* main drive struct, one entry per drive */
19836     int fl_curcyl;                /* current cylinder */
19837     int fl_hardcyl;               /* hardware cylinder, as opposed to: */
19838     int fl_cylinder;              /* cylinder number addressed */
19839     int fl_sector;                /* sector addressed */
19840     int fl_head;                  /* head number addressed */
19841     char fl_calibration;          /* CALIBRATED or UNCALIBRATED */
19842     char fl_density;              /* NO_DENS = ?, 0 = 360K; 1 = 360K/1.2M; etc.*/
19843     char fl_class;                /* bitmap for possible densities */
19844     struct device fl_geom;        /* Geometry of the drive */
19845     struct device fl_part[NR_PARTITIONS];  /* partition's base & size */
19846   } floppy[NR_DRIVES], *f_fp;
19847
19848   /* Gather transfer data for each sector. */
19849   PRIVATE struct trans {          /* precomputed transfer params */
19850     unsigned tr_count;            /* byte count */
19851     struct iorequest_s *tr_iop;   /* belongs to this I/O request */
19852     phys_bytes tr_phys;           /* user physical address */
19853     phys_bytes tr_dma;            /* DMA physical address */
19854   } ftrans[MAX_SECTORS];
```

```
19855
19856   PRIVATE unsigned f_count;       /* this many bytes to transfer */
19857   PRIVATE unsigned f_nexttrack;   /* don't do blocks above this */
19858   PRIVATE int motor_status;       /* bitmap of current motor status */
19859   PRIVATE int motor_goal;         /* bitmap of desired motor status */
19860   PRIVATE int need_reset;         /* set to 1 when controller must be reset */
19861   PRIVATE int d;                  /* diskette/drive combination */
19862   PRIVATE int f_drive;            /* selected drive */
19863   PRIVATE int f_device;           /* selected minor device */
19864   PRIVATE int f_opcode;           /* DEV_READ or DEV_WRITE */
19865   PRIVATE int f_sectors;          /* sectors per track of the floppy */
19866   PRIVATE int f_must;             /* must do part of the next track? */
19867   PRIVATE int f_busy;             /* BSY_IDLE, BSY_IO, BSY_WAKEN */
19868   PRIVATE int current_spec1;      /* latest spec1 sent to the controller */
19869   PRIVATE struct device *f_dv;    /* device's base and size */
19870   PRIVATE struct disk_parameter_s fmt_param; /* parameters for format */
19871   PRIVATE char f_results[MAX_RESULTS];/* the controller can give lots of output */
19872
19873
19874   /* Seven combinations of diskette/drive are supported.
19875    *
19876    * # Drive  diskette  Sectors  Tracks  Rotation Data-rate  Comment
19877    * 0  360K    360K      9       40     300 RPM  250 kbps   Standard PC DSDD
19878    * 1  1.2M    1.2M     15       80     360 RPM  500 kbps   AT disk in AT drive
19879    * 2  720K    360K      9       40     300 RPM  250 kbps   Quad density PC
19880    * 3  720K    720K      9       80     300 RPM  250 kbps   Toshiba, et al.
19881    * 4  1.2M    360K      9       40     360 RPM  300 kbps   PC disk in AT drive
19882    * 5  1.2M    720K      9       80     360 RPM  300 kbps   Toshiba in AT drive
19883    * 6  1.44M   1.44M    18       80     300 RPM  500 kbps   PS/2, et al.
19884    *
19885    * In addition, 720K diskettes can be read in 1.44MB drives, but that does
19886    * not need a different set of parameters.  This combination uses
19887    *
19888    * X  1.44M   720K      9       80     300 RPM  250 kbps   PS/2, et al.
19889    */
19890   PRIVATE char gap[NT] =
19891           {0x2A, 0x1B, 0x2A, 0x2A, 0x23, 0x23, 0x1B}; /* gap size */
19892   PRIVATE char rate[NT] =
19893           {0x02, 0x00, 0x02, 0x02, 0x01, 0x01, 0x00}; /* 2=250,1=300,0=500 kbps*/
19894   PRIVATE char nr_sectors[NT] =
19895           {9,    15,   9,    9,    9,    9,    18};   /* sectors/track */
19896   PRIVATE int nr_blocks[NT] =
19897           {720,  2400, 720,  1440, 720,  1440, 2880}; /* sectors/diskette*/
19898   PRIVATE char steps_per_cyl[NT] =
19899           {1,    1,    2,    1,    2,    1,     1};   /* 2 = dbl step */
19900   PRIVATE char mtr_setup[NT] =
19901           {1*HZ/4,3*HZ/4,1*HZ/4,4*HZ/4,3*HZ/4,3*HZ/4,4*HZ/4}; /* in ticks */
19902   PRIVATE char spec1[NT] =
19903           {0xDF, 0xDF, 0xDF, 0xDF, 0xDF, 0xDF, 0xDF}; /* step rate, etc. */
19904   PRIVATE char test_sector[NT] =
19905           {4*9,  14,   2*9,  4*9,  2*9,  4*9,  17};   /* to recognize it */
19906
19907   #define b(d)    (1 << (d))      /* bit for density d. */
19908
19909   /* The following table is used with the test_sector array to recognize a
19910    * drive/floppy combination.  The sector to test has been determined by
19911    * looking at the differences in gap size, sectors/track, and double stepping.
19912    * This means that types 0 and 3 can't be told apart, only the motor start
19913    * time differs.  If a read test succeeds then the drive is limited to the
19914    * set of densities it can support to avoid unnecessary tests in the future.
```

```
19915     */
19916
19917    PRIVATE struct test_order {
19918            char    t_density;      /* floppy/drive type */
19919            char    t_class;        /* limit drive to this class of densities */
19920    } test_order[NT-1] = {
19921            { 6,  b(3) | b(6) },            /* 1.44M  {720K, 1.44M} */
19922            { 1,  b(1) | b(4) | b(5) },     /* 1.2M   {1.2M, 360K, 720K} */
19923            { 3,  b(2) | b(3) | b(6) },     /* 720K   {360K, 720K, 1.44M} */
19924            { 4,  b(1) | b(4) | b(5) },     /* 360K   {1.2M, 360K, 720K} */
19925            { 5,  b(1) | b(4) | b(5) },     /* 720K   {1.2M, 360K, 720K} */
19926            { 2,  b(2) | b(3) },            /* 360K   {360K, 720K} */
19927            /* Note that type 0 is missing, type 3 can read/write it too (alas). */
19928    };
19929
19930    FORWARD _PROTOTYPE( struct device *f_prepare, (int device) );
19931    FORWARD _PROTOTYPE( char *f_name, (void) );
19932    FORWARD _PROTOTYPE( void f_cleanup, (void) );
19933    FORWARD _PROTOTYPE( int f_schedule, (int proc_nr, struct iorequest_s *iop) );
19934    FORWARD _PROTOTYPE( int f_finish, (void) );
19935    FORWARD _PROTOTYPE( void defuse, (void) );
19936    FORWARD _PROTOTYPE( void dma_setup, (struct trans *tp) );
19937    FORWARD _PROTOTYPE( void start_motor, (void) );
19938    FORWARD _PROTOTYPE( void stop_motor, (void) );
19939    FORWARD _PROTOTYPE( int seek, (struct floppy *fp) );
19940    FORWARD _PROTOTYPE( int f_transfer, (struct floppy *fp, struct trans *tp) );
19941    FORWARD _PROTOTYPE( int fdc_results, (void) );
19942    FORWARD _PROTOTYPE( int f_handler, (int irq) );
19943    FORWARD _PROTOTYPE( void fdc_out, (int val) );
19944    FORWARD _PROTOTYPE( int recalibrate, (struct floppy *fp) );
19945    FORWARD _PROTOTYPE( void f_reset, (void) );
19946    FORWARD _PROTOTYPE( void send_mess, (void) );
19947    FORWARD _PROTOTYPE( int f_intr_wait, (void) );
19948    FORWARD _PROTOTYPE( void f_timeout, (void) );
19949    FORWARD _PROTOTYPE( int read_id, (struct floppy *fp) );
19950    FORWARD _PROTOTYPE( int f_do_open, (struct driver *dp, message *m_ptr) );
19951    FORWARD _PROTOTYPE( int test_read, (int density) );
19952    FORWARD _PROTOTYPE( void f_geometry, (struct partition *entry));
19953
19954
19955    /* Entry points to this driver. */
19956    PRIVATE struct driver f_dtab = {
19957      f_name,       /* current device's name */
19958      f_do_open,    /* open or mount request, sense type of diskette */
19959      do_nop,       /* nothing on a close */
19960      do_diocntl,   /* get or set a partitions geometry */
19961      f_prepare,    /* prepare for I/O on a given minor device */
19962      f_schedule,   /* precompute cylinder, head, sector, etc. */
19963      f_finish,     /* do the I/O */
19964      f_cleanup,    /* cleanup before sending reply to user process */
19965      f_geometry    /* tell the geometry of the diskette */
19966    };
19967
19968
19969    /*===========================================================================*
19970     *                               floppy_task                                 *
19971     *===========================================================================*/
19972    PUBLIC void floppy_task()
19973    {
19974    /* Initialize the floppy structure. */
```

```
19975
19976     struct floppy *fp;
19977
19978     for (fp = &floppy[0]; fp < &floppy[NR_DRIVES]; fp++) {
19979           fp->fl_curcyl = NO_CYL;
19980           fp->fl_density = NO_DENS;
19981           fp->fl_class = ~0;
19982     }
19983
19984     put_irq_handler(FLOPPY_IRQ, f_handler);
19985     enable_irq(FLOPPY_IRQ);               /* ready for floppy interrupts */
19986
19987     driver_task(&f_dtab);
19988   }


19991   /*===========================================================================*
19992    *                              f_prepare                                    *
19993    *===========================================================================*/
19994   PRIVATE struct device *f_prepare(device)
19995   int device;
19996   {
19997   /* Prepare for I/O on a device. */
19998
19999     /* Leftover jobs after an I/O error must be removed */
20000     if (f_count > 0) defuse();
20001
20002     f_device = device;
20003     f_drive = device & ~(DEV_TYPE_BITS | FORMAT_DEV_BIT);
20004     if (f_drive < 0 || f_drive >= NR_DRIVES) return(NIL_DEV);
20005
20006     f_fp = &floppy[f_drive];
20007     f_dv = &f_fp->fl_geom;
20008     d = f_fp->fl_density;
20009     f_sectors = nr_sectors[d];
20010
20011     f_must = TRUE;        /* the first transfers must be done */
20012
20013     /* A partition? */
20014     if ((device &= DEV_TYPE_BITS) >= MINOR_fd0a)
20015           f_dv = &f_fp->fl_part[(device - MINOR_fd0a) >> DEV_TYPE_SHIFT];
20016
20017     return f_dv;
20018   }


20021   /*===========================================================================*
20022    *                              f_name                                       *
20023    *===========================================================================*/
20024   PRIVATE char *f_name()
20025   {
20026   /* Return a name for the current device. */
20027     static char name[] = "fd3";
20028
20029     name[2] = '0' + f_drive;
20030     return name;
20031   }


20034   /*===========================================================================*
```

```
20035    *                              f_cleanup                                *
20036    *===========================================================================*/
20037   PRIVATE void f_cleanup()
20038   {
20039     /* Start watchdog timer to turn all motors off in a few seconds.
20040      * There is a race here.  An old watchdog might bite before the
20041      * new delay is installed, and turn of the motors prematurely.
20042      * This cannot be solved simply by resetting motor_goal after
20043      * sending the message, because the new watchdog might bite
20044      * before motor_goal is reset.  Then the motors would stay on
20045      * until after the next floppy access.  This could be fixed with
20046      * extra code (call the clock task twice in some cases).  Or
20047      * stop_motor() could be replaced by send_mess(), and send a
20048      * STOP_MOTOR message to be accepted by the clock task.  This
20049      * would be slower but have the advantage that this comment could
20050      * be deleted!
20051      *
20052      * Since it is not likely and not serious for an old watchdog to
20053      * bite, accept that possibility for now.  A full solution to the
20054      * motor madness requires a lots of extra work anyway, such as
20055      * a separate timer for each motor, and smaller delays for motors
20056      * that have just been turned off or start faster than the spec.
20057      * (is there a motor-ready bit?).
20058      */
20059     motor_goal = 0;
20060     clock_mess(MOTOR_OFF, stop_motor);
20061   }


20064   /*===========================================================================*
20065    *                              f_schedule                                   *
20066    *===========================================================================*/
20067   PRIVATE int f_schedule(proc_nr, iop)
20068   int proc_nr;                  /* process doing the request */
20069   struct iorequest_s *iop;      /* pointer to read or write request */
20070   {
20071     int r, opcode, spanning;
20072     unsigned long pos;
20073     unsigned block;       /* Seen any 32M floppies lately? */
20074     unsigned nbytes, count, dma_count;
20075     phys_bytes user_phys, dma_phys;
20076     struct trans *tp, *tp0;
20077
20078     /* Ignore any alarm to turn motor off, now there is work to do. */
20079     motor_goal = motor_status;
20080
20081     /* This many bytes to read/write */
20082     nbytes = iop->io_nbytes;
20083     if ((nbytes & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);
20084
20085     /* From/to this position on disk */
20086     pos = iop->io_position;
20087     if ((pos & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);
20088
20089     /* To/from this user address */
20090     user_phys = numap(proc_nr, (vir_bytes) iop->io_buf, nbytes);
20091     if (user_phys == 0) return(iop->io_nbytes = EINVAL);
20092
20093     /* Read, write or format? */
20094     opcode = iop->io_request & ~OPTIONAL_IO;
```

```
20095     if (f_device & FORMAT_DEV_BIT) {
20096           if (opcode != DEV_WRITE) return(iop->io_nbytes = EIO);
20097           if (nbytes != BLOCK_SIZE) return(iop->io_nbytes = EINVAL);
20098
20099           phys_copy(user_phys + SECTOR_SIZE, vir2phys(&fmt_param),
20100                                                 (phys_bytes) sizeof fmt_param);
20101
20102           /* Check that the number of sectors in the data is reasonable, to
20103            * avoid division by 0.  Leave checking of other data to the FDC.
20104            */
20105           if (fmt_param.sectors_per_cylinder == 0)
20106                   return(iop->io_nbytes = EIO);
20107
20108           /* Only the first sector of the parameters now needed. */
20109           iop->io_nbytes = nbytes = SECTOR_SIZE;
20110     }
20111
20112     /* Which block on disk and how close to EOF? */
20113     if (pos >= f_dv->dv_size) return(OK);         /* At EOF */
20114     if (pos + nbytes > f_dv->dv_size) nbytes = f_dv->dv_size - pos;
20115     block = (f_dv->dv_base + pos) >> SECTOR_SHIFT;
20116
20117     spanning = FALSE;     /* set if the block spans a track */
20118
20119     /* While there are "unscheduled" bytes in the request: */
20120     do {
20121           count = nbytes;
20122
20123           if (f_count > 0 && block >= f_nexttrack) {
20124                   /* The new job leaves the track, finish all gathered jobs */
20125                   if ((r = f_finish()) != OK) return(r);
20126                   f_must = spanning;
20127           }
20128
20129           if (f_count == 0) {
20130                   /* This is the first job, compute cylinder and head */
20131                   f_opcode = opcode;
20132                   f_fp->fl_cylinder = block / (NR_HEADS * f_sectors);
20133                   f_fp->fl_hardcyl = f_fp->fl_cylinder * steps_per_cyl[d];
20134                   f_fp->fl_head = (block % (NR_HEADS * f_sectors)) / f_sectors;
20135
20136                   /* See where the next track starts, one is trouble enough */
20137                   f_nexttrack = (f_fp->fl_cylinder * NR_HEADS
20138                                           + f_fp->fl_head + 1) * f_sectors;
20139           }
20140
20141           /* Don't do track spanning I/O. */
20142           if (block + (count >> SECTOR_SHIFT) > f_nexttrack)
20143                   count = (f_nexttrack - block) << SECTOR_SHIFT;
20144
20145           /* Memory chunk to DMA. */
20146           dma_phys = user_phys;
20147           dma_count = dma_bytes_left(dma_phys);
20148
20149   #if _WORD_SIZE > 2
20150           /* The DMA chip uses a 24 bit address, so don't DMA above 16MB. */
20151           if (dma_phys >= 0x1000000) dma_count = 0;
20152   #endif
20153           if (dma_count < count) {
20154                   /* Nearing a 64K boundary. */
```

```
20155                   if (dma_count >= SECTOR_SIZE) {
20156                           /* Can read a few sectors before hitting the
20157                            * boundary.
20158                            */
20159                           count = dma_count & ~SECTOR_MASK;
20160                   } else {
20161                           /* Must use the special buffer for this. */
20162                           count = SECTOR_SIZE;
20163                           dma_phys = tmp_phys;
20164                   }
20165           }
20166
20167           /* Store the I/O parameters in the ftrans slots for the sectors to
20168            * read.  The first slot specifies all sectors, the ones following
20169            * it each specify one sector less.  This allows I/O to be started
20170            * in the middle of a block.
20171            */
20172           tp = tp0 = &ftrans[block % f_sectors];
20173
20174           block += count >> SECTOR_SHIFT;
20175           nbytes -= count;
20176           f_count += count;
20177           if (!(iop->io_request & OPTIONAL_IO)) f_must = TRUE;
20178
20179           do {
20180                   tp->tr_count = count;
20181                   tp->tr_iop = iop;
20182                   tp->tr_phys = user_phys;
20183                   tp->tr_dma = dma_phys;
20184                   tp++;
20185
20186                   user_phys += SECTOR_SIZE;
20187                   dma_phys += SECTOR_SIZE;
20188                   count -= SECTOR_SIZE;
20189           } while (count > 0);
20190
20191           spanning = TRUE;        /* the rest of the block may span a track */
20192     } while (nbytes > 0);
20193
20194     return(OK);
20195   }


20198   /*==========================================================================*
20199    *                                f_finish                                  *
20200    *==========================================================================*/
20201   PRIVATE int f_finish()
20202   {
20203   /* Carry out the I/O requests gathered in ftrans[].  */
20204
20205     struct floppy *fp = f_fp;
20206     struct trans *tp;
20207     int r, errors;
20208
20209     if (f_count == 0) return(OK); /* Spurious finish. */
20210
20211     /* If all the requests are optional then don't read from the next track.
20212      * (There may be enough buffers to read the next track, but doing so is
20213      * unwise.  It's no good to be greedy on a slow device.)
20214      */
```

```
20215     if (!f_must) {
20216           defuse();
20217           return(EAGAIN);
20218     }
20219
20220     /* See if motor is running; if not, turn it on and wait */
20221     start_motor();
20222
20223     /* Let read_id find out the next sector to read/write if it pays to do so.
20224      * Note that no read_id is done while formatting if there is one format
20225      * request per track as there should be.
20226      */
20227     fp->fl_sector = f_count >= (6 * SECTOR_SIZE) ? 0 : BASE_SECTOR;
20228
20229     do {
20230           /* This loop allows a failed operation to be repeated. */
20231           errors = 0;
20232           for (;;) {
20233                   /* First check to see if a reset is needed. */
20234                   if (need_reset) f_reset();
20235
20236                   /* Set the stepping rate */
20237                   if (current_spec1 != spec1[d]) {
20238                           fdc_out(FDC_SPECIFY);
20239                           current_spec1 = spec1[d];
20240                           fdc_out(current_spec1);
20241                           fdc_out(SPEC2);
20242                   }
20243
20244                   /* Set the data rate */
20245                   if (pc_at) out_byte(FDC_RATE, rate[d]);
20246
20247                   /* If we are going to a new cylinder, perform a seek. */
20248                   r = seek(fp);
20249
20250                   if (fp->fl_sector == NO_SECTOR) {
20251                           /* Don't retry read_id too often, we need tp soon */
20252                           if (errors > 0) fp->fl_sector = BASE_SECTOR;
20253
20254                           /* Find out what the current sector is */
20255                           if (r == OK) r = read_id(fp);
20256                   }
20257
20258                   /* Look for the next job in ftrans[] */
20259                   if (fp->fl_sector != NO_SECTOR) {
20260                           for (;;) {
20261                                   if (fp->fl_sector >= BASE_SECTOR + f_sectors)
20262                                           fp->fl_sector = BASE_SECTOR;
20263
20264                                   tp = &ftrans[fp->fl_sector - BASE_SECTOR];
20265                                   if (tp->tr_count > 0) break;
20266                                   fp->fl_sector++;
20267                           }
20268                           /* Do not transfer more than f_count bytes. */
20269                           if (tp->tr_count > f_count) tp->tr_count = f_count;
20270                   }
20271
20272                   if (r == OK && tp->tr_dma == tmp_phys
20273                                           && f_opcode == DEV_WRITE) {
20274                           /* Copy the bad user buffer to the DMA buffer. */
```

```
20275                           phys_copy(tp->tr_phys, tp->tr_dma,
20276                                                   (phys_bytes) tp->tr_count);
20277                   }
20278
20279                   /* Set up the DMA chip and perform the transfer. */
20280                   if (r == OK) {
20281                           dma_setup(tp);
20282                           r = f_transfer(fp, tp);
20283                   }
20284
20285                   if (r == OK && tp->tr_dma == tmp_phys
20286                                                   && f_opcode == DEV_READ) {
20287                           /* Copy the DMA buffer to the bad user buffer. */
20288                           phys_copy(tp->tr_dma, tp->tr_phys,
20289                                                   (phys_bytes) tp->tr_count);
20290                   }
20291
20292                   if (r == OK) break;     /* if successful, exit loop */
20293
20294                   /* Don't retry if write protected or too many errors. */
20295                   if (err_no_retry(r) || ++errors == MAX_ERRORS) {
20296                           if (fp->fl_sector != 0) tp->tr_iop->io_nbytes = EIO;
20297                           return(EIO);
20298                   }
20299
20300                   /* Recalibrate if halfway, but bail out if optional I/O. */
20301                   if (errors == MAX_ERRORS / 2) {
20302                           fp->fl_calibration = UNCALIBRATED;
20303                           if (tp->tr_iop->io_request & OPTIONAL_IO)
20304                                   return(tp->tr_iop->io_nbytes = EIO);
20305                   }
20306           }
20307           f_count -= tp->tr_count;
20308           tp->tr_iop->io_nbytes -= tp->tr_count;
20309     } while (f_count > 0);
20310
20311     /* Defuse the leftover partial jobs. */
20312     defuse();
20313
20314     return(OK);
20315   }


20318   /*==========================================================================*
20319    *                              defuse                                      *
20320    *==========================================================================*/
20321   PRIVATE void defuse()
20322   {
20323   /* Invalidate leftover requests in the transfer array. */
20324
20325     struct trans *tp;
20326
20327     for (tp = ftrans; tp < ftrans + MAX_SECTORS; tp++) tp->tr_count = 0;
20328     f_count = 0;
20329   }


20332   /*==========================================================================*
20333    *                              dma_setup                                   *
20334    *==========================================================================*/
```

```
20335   PRIVATE void dma_setup(tp)
20336   struct trans *tp;               /* pointer to the transfer struct */
20337   {
20338   /* The IBM PC can perform DMA operations by using the DMA chip.  To use it,
20339    * the DMA (Direct Memory Access) chip is loaded with the 20-bit memory address
20340    * to be read from or written to, the byte count minus 1, and a read or write
20341    * opcode.  This routine sets up the DMA chip.  Note that the chip is not
20342    * capable of doing a DMA across a 64K boundary (e.g., you can't read a
20343    * 512-byte block starting at physical address 65520).
20344    */
20345
20346     /* Set up the DMA registers.  (The comment on the reset is a bit strong,
20347      * it probably only resets the floppy channel.)
20348      */
20349     out_byte(DMA_INIT, DMA_RESET_VAL);    /* reset the dma controller */
20350     out_byte(DMA_FLIPFLOP, 0);            /* write anything to reset it */
20351     out_byte(DMA_MODE, f_opcode == DEV_WRITE ? DMA_WRITE : DMA_READ);
20352     out_byte(DMA_ADDR, (int) tp->tr_dma >>  0);
20353     out_byte(DMA_ADDR, (int) tp->tr_dma >>  8);
20354     out_byte(DMA_TOP, (int) (tp->tr_dma >> 16));
20355     out_byte(DMA_COUNT, (tp->tr_count - 1) >> 0);
20356     out_byte(DMA_COUNT, (tp->tr_count - 1) >> 8);
20357     out_byte(DMA_INIT, 2);        /* some sort of enable */
20358   }


20361   /*==========================================================================*
20362    *                              start_motor                                 *
20363    *==========================================================================*/
20364   PRIVATE void start_motor()
20365   {
20366   /* Control of the floppy disk motors is a big pain.  If a motor is off, you
20367    * have to turn it on first, which takes 1/2 second.  You can't leave it on
20368    * all the time, since that would wear out the diskette.  However, if you turn
20369    * the motor off after each operation, the system performance will be awful.
20370    * The compromise used here is to leave it on for a few seconds after each
20371    * operation.  If a new operation is started in that interval, it need not be
20372    * turned on again.  If no new operation is started, a timer goes off and the
20373    * motor is turned off.  I/O port DOR has bits to control each of 4 drives.
20374    * The timer cannot go off while we are changing with the bits, since the
20375    * clock task cannot run while another (this) task is active, so there is no
20376    * need to lock().
20377    */
20378
20379     int motor_bit, running;
20380     message mess;
20381
20382     motor_bit = 1 << f_drive;             /* bit mask for this drive */
20383     running = motor_status & motor_bit;   /* nonzero if this motor is running */
20384     motor_goal = motor_status | motor_bit;/* want this drive running too */
20385
20386     out_byte(DOR, (motor_goal << MOTOR_SHIFT) | ENABLE_INT | f_drive);
20387     motor_status = motor_goal;
20388
20389     /* If the motor was already running, we don't have to wait for it. */
20390     if (running) return;                  /* motor was already running */
20391     clock_mess(mtr_setup[d], send_mess);  /* motor was not running */
20392     receive(CLOCK, &mess);                /* wait for clock interrupt */
20393   }
```

```
20396   /*===========================================================================*
20397    *                              stop_motor                                   *
20398    *===========================================================================*/
20399   PRIVATE void stop_motor()
20400   {
20401   /* This routine is called by the clock interrupt after several seconds have
20402    * elapsed with no floppy disk activity.  It checks to see if any drives are
20403    * supposed to be turned off, and if so, turns them off.
20404    */
20405
20406     if (motor_goal != motor_status) {
20407           out_byte(DOR, (motor_goal << MOTOR_SHIFT) | ENABLE_INT);
20408           motor_status = motor_goal;
20409     }
20410   }


20413   /*===========================================================================*
20414    *                              floppy_stop                                  *
20415    *===========================================================================*/
20416   PUBLIC void floppy_stop()
20417   {
20418   /* Stop all activity. */
20419
20420     motor_goal = 0;
20421     stop_motor();
20422   }


20425   /*===========================================================================*
20426    *                              seek                                         *
20427    *===========================================================================*/
20428   PRIVATE int seek(fp)
20429   struct floppy *fp;              /* pointer to the drive struct */
20430   {
20431   /* Issue a SEEK command on the indicated drive unless the arm is already
20432    * positioned on the correct cylinder.
20433    */
20434
20435     int r;
20436     message mess;
20437
20438     /* Are we already on the correct cylinder? */
20439     if (fp->fl_calibration == UNCALIBRATED)
20440           if (recalibrate(fp) != OK) return(ERR_SEEK);
20441     if (fp->fl_curcyl == fp->fl_hardcyl) return(OK);
20442
20443     /* No.  Wrong cylinder.  Issue a SEEK and wait for interrupt. */
20444     fdc_out(FDC_SEEK);
20445     fdc_out((fp->fl_head << 2) | f_drive);
20446     fdc_out(fp->fl_hardcyl);
20447     if (need_reset) return(ERR_SEEK);     /* if controller is sick, abort seek */
20448     if (f_intr_wait() != OK) return(ERR_TIMEOUT);
20449
20450     /* Interrupt has been received.  Check drive status. */
20451     fdc_out(FDC_SENSE);           /* probe FDC to make it return status */
20452     r = fdc_results();            /* get controller status bytes */
20453     if (r != OK || (f_results[ST0] & ST0_BITS) != SEEK_ST0
20454                                   || f_results[ST1] != fp->fl_hardcyl) {
```

```
20455           /* seek failed, may need a recalibrate */
20456           return(ERR_SEEK);
20457       }
20458       /* give head time to settle on a format, no retrying here! */
20459       if (f_device & FORMAT_DEV_BIT) {
20460           clock_mess(2, send_mess);
20461           receive(CLOCK, &mess);
20462       }
20463       fp->fl_curcyl = fp->fl_hardcyl;
20464       return(OK);
20465   }


20468   /*===========================================================================*
20469    *                              f_transfer                                   *
20470    *===========================================================================*/
20471   PRIVATE int f_transfer(fp, tp)
20472   struct floppy *fp;              /* pointer to the drive struct */
20473   struct trans *tp;               /* pointer to the transfer struct */
20474   {
20475   /* The drive is now on the proper cylinder.  Read, write or format 1 block. */
20476
20477       int r, s;
20478
20479       /* Never attempt a transfer if the drive is uncalibrated or motor is off. */
20480       if (fp->fl_calibration == UNCALIBRATED) return(ERR_TRANSFER);
20481       if ((motor_status & (1 << f_drive)) == 0) return(ERR_TRANSFER);
20482
20483       /* The command is issued by outputting several bytes to the controller chip.
20484        */
20485       if (f_device & FORMAT_DEV_BIT) {
20486           fdc_out(FDC_FORMAT);
20487           fdc_out((fp->fl_head << 2) | f_drive);
20488           fdc_out(fmt_param.sector_size_code);
20489           fdc_out(fmt_param.sectors_per_cylinder);
20490           fdc_out(fmt_param.gap_length_for_format);
20491           fdc_out(fmt_param.fill_byte_for_format);
20492       } else {
20493           fdc_out(f_opcode == DEV_WRITE ? FDC_WRITE : FDC_READ);
20494           fdc_out((fp->fl_head << 2) | f_drive);
20495           fdc_out(fp->fl_cylinder);
20496           fdc_out(fp->fl_head);
20497           fdc_out(fp->fl_sector);
20498           fdc_out(SECTOR_SIZE_CODE);
20499           fdc_out(f_sectors);
20500           fdc_out(gap[d]);        /* sector gap */
20501           fdc_out(DTL);           /* data length */
20502       }
20503
20504       /* Block, waiting for disk interrupt. */
20505       if (need_reset) return(ERR_TRANSFER); /* if controller is sick, abort op */
20506
20507       if (f_intr_wait() != OK) return(ERR_TIMEOUT);
20508
20509       /* Get controller status and check for errors. */
20510       r = fdc_results();
20511       if (r != OK) return(r);
20512
20513       if (f_results[ST1] & WRITE_PROTECT) {
20514           printf("%s: diskette is write protected.\n", f_name());
```

```
20515           return(ERR_WR_PROTECT);
20516     }
20517
20518     if ((f_results[ST0] & ST0_BITS) != TRANS_ST0) return(ERR_TRANSFER);
20519     if (f_results[ST1] | f_results[ST2]) return(ERR_TRANSFER);
20520
20521     if (f_device & FORMAT_DEV_BIT) return(OK);
20522
20523     /* Compare actual numbers of sectors transferred with expected number. */
20524     s =  (f_results[ST_CYL] - fp->fl_cylinder) * NR_HEADS * f_sectors;
20525     s += (f_results[ST_HEAD] - fp->fl_head) * f_sectors;
20526     s += (f_results[ST_SEC] - fp->fl_sector);
20527     if ((s << SECTOR_SHIFT) != tp->tr_count) return(ERR_TRANSFER);
20528
20529     /* This sector is next for I/O: */
20530     fp->fl_sector = f_results[ST_SEC];
20531     return(OK);
20532   }


20535   /*===========================================================================*
20536    *                              fdc_results                                  *
20537    *===========================================================================*/
20538   PRIVATE int fdc_results()
20539   {
20540   /* Extract results from the controller after an operation, then allow floppy
20541    * interrupts again.
20542    */
20543
20544     int result_nr, status;
20545     struct milli_state ms;
20546
20547     /* Extract bytes from FDC until it says it has no more.  The loop is
20548      * really an outer loop on result_nr and an inner loop on status.
20549      */
20550     result_nr = 0;
20551     milli_start(&ms);
20552     do {
20553           /* Reading one byte is almost a mirror of fdc_out() - the DIRECTION
20554            * bit must be set instead of clear, but the CTL_BUSY bit destroys
20555            * the perfection of the mirror.
20556            */
20557           status = in_byte(FDC_STATUS) & (MASTER | DIRECTION | CTL_BUSY);
20558           if (status == (MASTER | DIRECTION | CTL_BUSY)) {
20559                   if (result_nr >= MAX_RESULTS) break;    /* too many results */
20560                   f_results[result_nr++] = in_byte(FDC_DATA);
20561                   continue;
20562           }
20563           if (status == MASTER) { /* all read */
20564                   enable_irq(FLOPPY_IRQ);
20565                   return(OK);     /* only good exit */
20566           }
20567     } while (milli_elapsed(&ms) < TIMEOUT);
20568     need_reset = TRUE;            /* controller chip must be reset */
20569     enable_irq(FLOPPY_IRQ);
20570     return(ERR_STATUS);
20571   }


20574   /*===========================================================================*
```

```
20575     *                               f_handler                                *
20576     *========================================================================*/
20577    PRIVATE int f_handler(irq)
20578    int irq;
20579    {
20580    /* FDC interrupt, send message to floppy task. */
20581
20582      interrupt(FLOPPY);
20583      return 0;
20584    }


20587    /*========================================================================*
20588     *                               fdc_out                                  *
20589     *========================================================================*/
20590    PRIVATE void fdc_out(val)
20591    int val;                /* write this byte to floppy disk controller */
20592    {
20593    /* Output a byte to the controller.  This is not entirely trivial, since you
20594     * can only write to it when it is listening, and it decides when to listen.
20595     * If the controller refuses to listen, the FDC chip is given a hard reset.
20596     */
20597
20598      struct milli_state ms;
20599
20600      if (need_reset) return;       /* if controller is not listening, return */
20601
20602      /* It may take several tries to get the FDC to accept a command. */
20603      milli_start(&ms);
20604      while ((in_byte(FDC_STATUS) & (MASTER | DIRECTION)) != (MASTER | 0)) {
20605            if (milli_elapsed(&ms) >= TIMEOUT) {
20606                    /* Controller is not listening.  Hit it over the head. */
20607                    need_reset = TRUE;
20608                    return;
20609            }
20610      }
20611      out_byte(FDC_DATA, val);
20612    }


20615    /*========================================================================*
20616     *                               recalibrate                              *
20617     *========================================================================*/
20618    PRIVATE int recalibrate(fp)
20619    struct floppy *fp;      /* pointer tot he drive struct */
20620    {
20621    /* The floppy disk controller has no way of determining its absolute arm
20622     * position (cylinder).  Instead, it steps the arm a cylinder at a time and
20623     * keeps track of where it thinks it is (in software).  However, after a
20624     * SEEK, the hardware reads information from the diskette telling where the
20625     * arm actually is.  If the arm is in the wrong place, a recalibration is done,
20626     * which forces the arm to cylinder 0.  This way the controller can get back
20627     * into sync with reality.
20628     */
20629
20630      int r;
20631
20632      /* Issue the RECALIBRATE command and wait for the interrupt. */
20633      start_motor();                /* can't recalibrate with motor off */
20634      fdc_out(FDC_RECALIBRATE);     /* tell drive to recalibrate itself */
```

```
20635     fdc_out(f_drive);             /* specify drive */
20636     if (need_reset) return(ERR_SEEK);     /* don't wait if controller is sick */
20637     if (f_intr_wait() != OK) return(ERR_TIMEOUT);
20638
20639     /* Determine if the recalibration succeeded. */
20640     fdc_out(FDC_SENSE);           /* issue SENSE command to request results */
20641     r = fdc_results();            /* get results of the FDC_RECALIBRATE command*/
20642     fp->fl_curcyl = NO_CYL;       /* force a SEEK next time */
20643     if (r != OK ||                /* controller would not respond */
20644        (f_results[ST0] & ST0_BITS) != SEEK_ST0 || f_results[ST_PCN] != 0) {
20645           /* Recalibration failed.  FDC must be reset. */
20646           need_reset = TRUE;
20647           return(ERR_RECALIBRATE);
20648     } else {
20649           /* Recalibration succeeded. */
20650           fp->fl_calibration = CALIBRATED;
20651           return(OK);
20652     }
20653   }


20656   /*===========================================================================*
20657    *                              f_reset                                      *
20658    *===========================================================================*/
20659   PRIVATE void f_reset()
20660   {
20661   /* Issue a reset to the controller.  This is done after any catastrophe,
20662    * like the controller refusing to respond.
20663    */
20664
20665     int i;
20666     message mess;
20667
20668     /* Disable interrupts and strobe reset bit low. */
20669     need_reset = FALSE;
20670
20671     /* It is not clear why the next lock is needed.  Writing 0 to DOR causes
20672      * interrupt, while the PC documentation says turning bit 8 off disables
20673      * interrupts.  Without the lock:
20674      *   1) the interrupt handler sets the floppy mask bit in the 8259.
20675      *   2) writing ENABLE_INT to DOR causes the FDC to assert the interrupt
20676      *      line again, but the mask stops the cpu being interrupted.
20677      *   3) the sense interrupt clears the interrupt (not clear which one).
20678      * and for some reason the reset does not work.
20679      */
20680     lock();
20681     motor_status = 0;
20682     motor_goal = 0;
20683     out_byte(DOR, 0);             /* strobe reset bit low */
20684     out_byte(DOR, ENABLE_INT);    /* strobe it high again */
20685     unlock();
20686     receive(HARDWARE, &mess);     /* collect the RESET interrupt */
20687
20688     /* The controller supports 4 drives and returns a result for each of them.
20689      * Collect all the results now.  The old version only collected the first
20690      * result.  This happens to work for 2 drives, but it doesn't work for 3
20691      * or more drives, at least with only drives 0 and 2 actually connected
20692      * (the controller generates an extra interrupt for the middle drive when
20693      * drive 2 is accessed and the driver panics).
20694      *
```

```
20695       * It would be better to keep collecting results until there are no more.
20696       * For this, fdc_results needs to return the number of results (instead
20697       * of OK) when it succeeds.
20698       */
20699      for (i = 0; i < 4; i++) {
20700            fdc_out(FDC_SENSE);     /* probe FDC to make it return status */
20701            (void) fdc_results();   /* flush controller */
20702      }
20703      for (i = 0; i < NR_DRIVES; i++)       /* clear each drive */
20704            floppy[i].fl_calibration = UNCALIBRATED;
20705
20706      /* The current timing parameters must be specified again. */
20707      current_spec1 = 0;
20708    }


20711    /*===========================================================================*
20712     *                               send_mess                                   *
20713     *===========================================================================*/
20714    PRIVATE void send_mess()
20715    {
20716    /* This routine is called when the clock task has timed out on motor startup.*/
20717
20718      message mess;
20719
20720      send(FLOPPY, &mess);
20721    }


20724    /*===========================================================================*
20725     *                               f_intr_wait                                 *
20726     *===========================================================================*/
20727    PRIVATE int f_intr_wait()
20728    {
20729    /* Wait for an interrupt, but not forever.  The FDC may have all the time of
20730     * the world, but we humans do not.
20731     */
20732      message mess;
20733
20734      f_busy = BSY_IO;
20735      clock_mess(WAKEUP, f_timeout);
20736      receive(HARDWARE, &mess);
20737
20738      if (f_busy == BSY_WAKEN) {
20739            /* No interrupt from the FDC, this means that there is probably no
20740             * floppy in the drive.  Get the FDC down to earth and return error.
20741             */
20742            f_reset();
20743            return(ERR_TIMEOUT);
20744      }
20745      f_busy = BSY_IDLE;
20746      return(OK);
20747    }


20750    /*===========================================================================*
20751     *                               f_timeout                                   *
20752     *===========================================================================*/
20753    PRIVATE void f_timeout()
20754    {
```

```
20755   /* When it takes too long for the FDC to get an interrupt (no floppy in the
20756    * drive), this routine is called.  It sets a flag and fakes a hardware
20757    * interrupt.
20758    */
20759     if (f_busy == BSY_IO) {
20760           f_busy = BSY_WAKEN;
20761           interrupt(FLOPPY);
20762     }
20763   }


20766   /*===========================================================================*
20767    *                               read_id                                     *
20768    *===========================================================================*/
20769   PRIVATE int read_id(fp)
20770   struct floppy *fp;      /* pointer to the drive struct */
20771   {
20772   /* Determine current cylinder and sector. */
20773
20774     int result;
20775
20776     /* Never attempt a read id if the drive is uncalibrated or motor is off. */
20777     if (fp->fl_calibration == UNCALIBRATED) return(ERR_READ_ID);
20778     if ((motor_status & (1 << f_drive)) == 0) return(ERR_READ_ID);
20779
20780     /* The command is issued by outputting 2 bytes to the controller chip. */
20781     fdc_out(FDC_READ_ID);         /* issue the read id command */
20782     fdc_out( (f_fp->fl_head << 2) | f_drive);
20783
20784     /* Block, waiting for disk interrupt. */
20785     if (need_reset) return(ERR_READ_ID);  /* if controller is sick, abort op */
20786
20787     if (f_intr_wait() != OK) return(ERR_TIMEOUT);
20788
20789     /* Get controller status and check for errors. */
20790     result = fdc_results();
20791     if (result != OK) return(result);
20792
20793     if ((f_results[ST0] & ST0_BITS) != TRANS_ST0) return(ERR_READ_ID);
20794     if (f_results[ST1] | f_results[ST2]) return(ERR_READ_ID);
20795
20796     /* The next sector is next for I/O: */
20797     f_fp->fl_sector = f_results[ST_SEC] + 1;
20798     return(OK);
20799   }


20802   /*===========================================================================*
20803    *                               f_do_open                                   *
20804    *===========================================================================*/
20805   PRIVATE int f_do_open(dp, m_ptr)
20806   struct driver *dp;
20807   message *m_ptr;                /* pointer to open message */
20808   {
20809   /* Handle an open on a floppy.  Determine diskette type if need be. */
20810
20811     int dtype;
20812     struct test_order *top;
20813
20814     /* Decode the message parameters. */
```

```
20815     if (f_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
20816
20817     dtype = f_device & DEV_TYPE_BITS;     /* get density from minor dev */
20818     if (dtype >= MINOR_fd0a) dtype = 0;
20819     if (dtype != 0) {
20820           /* All types except 0 indicate a specific drive/medium combination.*/
20821           dtype = (dtype >> DEV_TYPE_SHIFT) - 1;
20822           if (dtype >= NT) return(ENXIO);
20823           f_fp->fl_density = dtype;
20824           f_fp->fl_geom.dv_size = (long) nr_blocks[dtype] << SECTOR_SHIFT;
20825           return(OK);
20826     }
20827     if (f_device & FORMAT_DEV_BIT) return(EIO);   /* Can't format /dev/fdx */
20828
20829     /* No need to test if the motor is still running. */
20830     if (motor_status & (1 << f_drive)) return(OK);
20831
20832     /* The device opened is /dev/fdx.  Experimentally determine drive/medium.
20833      * First check fl_density.  If it is not NO_DENS, the drive has been used
20834      * before and the value of fl_density tells what was found last time. Try
20835      * that first.
20836      */
20837     if (f_fp->fl_density != NO_DENS && test_read(f_fp->fl_density) == OK)
20838           return(OK);
20839
20840     /* Either drive type is unknown or a different diskette is now present.
20841      * Use test_order to try them one by one.
20842      */
20843     for (top = &test_order[0]; top < &test_order[NT-1]; top++) {
20844           dtype = top->t_density;
20845
20846           /* Skip densities that have been proven to be impossible */
20847           if (!(f_fp->fl_class & (1 << dtype))) continue;
20848
20849           if (test_read(dtype) == OK) {
20850                   /* The test succeeded, use this knowledge to limit the
20851                    * drive class to match the density just read.
20852                    */
20853                   f_fp->fl_class &= top->t_class;
20854                   return(OK);
20855           }
20856           /* Test failed, wrong density or did it time out? */
20857           if (f_busy == BSY_WAKEN) break;
20858     }
20859     f_fp->fl_density = NO_DENS;
20860     return(EIO);                  /* nothing worked */
20861   }


20864   /*============================================================================*
20865    *                              test_read                                     *
20866    *============================================================================*/
20867   PRIVATE int test_read(density)
20868   int density;
20869   {
20870   /* Try to read the highest numbered sector on cylinder 2.  Not all floppy
20871    * types have as many sectors per track, and trying cylinder 2 finds the
20872    * ones that need double stepping.
20873    */
20874
```

```
20875       message m;
20876       int r, device;
20877
20878       f_fp->fl_density = density;
20879       device = ((density + 1) << DEV_TYPE_SHIFT) + f_drive;
20880       f_fp->fl_geom.dv_size = (long) nr_blocks[density] << SECTOR_SHIFT;
20881       m.m_type = DEV_READ;
20882       m.DEVICE = device;
20883       m.PROC_NR = FLOPPY;
20884       m.COUNT = SECTOR_SIZE;
20885       m.POSITION = (long) test_sector[density] * SECTOR_SIZE;
20886       m.ADDRESS = (char *) tmp_buf;
20887       r = do_rdwt(&f_dtab, &m);
20888       if (r != SECTOR_SIZE) return(EIO);
20889
20890       partition(&f_dtab, f_drive, P_FLOPPY);
20891       return(OK);
20892     }


20895     /*==========================================================================*
20896      *                                f_geometry                                *
20897      *==========================================================================*/
20898     PRIVATE void f_geometry(entry)
20899     struct partition *entry;
20900     {
20901       entry->cylinders = nr_blocks[d] / (NR_HEADS * f_sectors);
20902       entry->heads = NR_HEADS;
20903       entry->sectors = f_sectors;
20904     }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/i8259.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

21000     /* This file contains routines for initializing the 8259 interrupt controller:
21001      *      get_irq_handler: address of handler for a given interrupt
21002      *      put_irq_handler: register an interrupt handler
21003      *      intr_init:      initialize the interrupt controller(s)
21004      */
21005
21006     #include "kernel.h"
21007
21008     #define ICW1_AT         0x11    /* edge triggered, cascade, need ICW4 */
21009     #define ICW1_PC         0x13    /* edge triggered, no cascade, need ICW4 */
21010     #define ICW1_PS         0x19    /* level triggered, cascade, need ICW4 */
21011     #define ICW4_AT         0x01    /* not SFNM, not buffered, normal EOI, 8086 */
21012     #define ICW4_PC         0x09    /* not SFNM, buffered, normal EOI, 8086 */
21013
21014     FORWARD _PROTOTYPE( int spurious_irq, (int irq) );
21015
21016     #if _WORD_SIZE == 2
21017     typedef _PROTOTYPE( void (*vecaddr_t), (void) );
21018
21019     FORWARD _PROTOTYPE( void set_vec, (int vec_nr, vecaddr_t addr) );
```

```
21020
21021   PRIVATE vecaddr_t int_vec[] = {
21022     int00, int01, int02, int03, int04, int05, int06, int07,
21023   };
21024
21025   PRIVATE vecaddr_t irq_vec[] = {
21026     hwint00, hwint01, hwint02, hwint03, hwint04, hwint05, hwint06, hwint07,
21027     hwint08, hwint09, hwint10, hwint11, hwint12, hwint13, hwint14, hwint15,
21028   };
21029   #else
21030   #define set_vec(nr, addr)       ((void)0)
21031   #endif
21032
21033
21034   /*===========================================================================*
21035    *                              intr_init                                    *
21036    *===========================================================================*/
21037   PUBLIC void intr_init(mine)
21038   int mine;
21039   {
21040   /* Initialize the 8259s, finishing with all interrupts disabled.  This is
21041    * only done in protected mode, in real mode we don't touch the 8259s, but
21042    * use the BIOS locations instead.  The flag "mine" is set if the 8259s are
21043    * to be programmed for Minix, or to be reset to what the BIOS expects.
21044    */
21045
21046     int i;
21047
21048     lock();
21049     if (protected_mode) {
21050           /* The AT and newer PS/2 have two interrupt controllers, one master,
21051            * one slaved at IRQ 2.  (We don't have to deal with the PC that
21052            * has just one controller, because it must run in real mode.)
21053            */
21054           out_byte(INT_CTL, ps_mca ? ICW1_PS : ICW1_AT);
21055           out_byte(INT_CTLMASK, mine ? IRQ0_VECTOR : BIOS_IRQ0_VEC);
21056                                                       /* ICW2 for master */
21057           out_byte(INT_CTLMASK, (1 << CASCADE_IRQ));      /* ICW3 tells slaves */
21058           out_byte(INT_CTLMASK, ICW4_AT);
21059           out_byte(INT_CTLMASK, ~(1 << CASCADE_IRQ));     /* IRQ 0-7 mask */
21060           out_byte(INT2_CTL, ps_mca ? ICW1_PS : ICW1_AT);
21061           out_byte(INT2_CTLMASK, mine ? IRQ8_VECTOR : BIOS_IRQ8_VEC);
21062                                                       /* ICW2 for slave */
21063           out_byte(INT2_CTLMASK, CASCADE_IRQ);            /* ICW3 is slave nr */
21064           out_byte(INT2_CTLMASK, ICW4_AT);
21065           out_byte(INT2_CTLMASK, ~0);                     /* IRQ 8-15 mask */
21066     } else {
21067           /* Use the BIOS interrupt vectors in real mode.  We only reprogram the
21068            * exceptions here, the interrupt vectors are reprogrammed on demand.
21069            * SYS_VECTOR is the Minix system call for message passing.
21070            */
21071           for (i = 0; i < 8; i++) set_vec(i, int_vec[i]);
21072           set_vec(SYS_VECTOR, s_call);
21073     }
21074
21075     /* Initialize the table of interrupt handlers. */
21076     for (i = 0; i < NR_IRQ_VECTORS; i++) irq_table[i] = spurious_irq;
21077   }

21079   /*===========================================================================*
```

```
21080     *                             spurious_irq                              *
21081     *=======================================================================*/
21082    PRIVATE int spurious_irq(irq)
21083    int irq;
21084    {
21085    /* Default interrupt handler.  It complains a lot. */
21086
21087      if (irq < 0 || irq >= NR_IRQ_VECTORS)
21088            panic("invalid call to spurious_irq", irq);
21089
21090      printf("spurious irq %d\n", irq);
21091
21092      return 1;     /* Reenable interrupt */
21093    }

21095    /*=======================================================================*
21096     *                             put_irq_handler                           *
21097     *=======================================================================*/
21098    PUBLIC void put_irq_handler(irq, handler)
21099    int irq;
21100    irq_handler_t handler;
21101    {
21102    /* Register an interrupt handler. */
21103
21104      if (irq < 0 || irq >= NR_IRQ_VECTORS)
21105            panic("invalid call to put_irq_handler", irq);
21106
21107      if (irq_table[irq] == handler)
21108            return;         /* extra initialization */
21109
21110      if (irq_table[irq] != spurious_irq)
21111            panic("attempt to register second irq handler for irq", irq);
21112
21113      disable_irq(irq);
21114      if (!protected_mode) set_vec(BIOS_VECTOR(irq), irq_vec[irq]);
21115      irq_table[irq]= handler;
21116      irq_use |= 1 << irq;
21117    }


21120    #if _WORD_SIZE == 2
21121    /*=======================================================================*
21122     *                                  set_vec                              *
21123     *=======================================================================*/
21124    PRIVATE void set_vec(vec_nr, addr)
21125    int vec_nr;                     /* which vector */
21126    vecaddr_t addr;                 /* where to start */
21127    {
21128    /* Set up a real mode interrupt vector. */
21129
21130      u16_t vec[2];
21131
21132      /* Build the vector in the array 'vec'. */
21133      vec[0] = (u16_t) addr;
21134      vec[1] = (u16_t) physb_to_hclick(code_base);
21135
21136      /* Copy the vector into place. */
21137      phys_copy(vir2phys(vec), vec_nr * 4L, 4L);
21138    }
21139    #endif /* _WORD_SIZE == 2 */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/keyboard.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

21200    /* Keyboard driver for PC's and AT's.
21201     *
21202     * Changed by Marcus Hampel     (04/02/1994)
21203     *  - Loadable keymaps
21204     */
21205
21206    #include "kernel.h"
21207    #include <termios.h>
21208    #include <signal.h>
21209    #include <unistd.h>
21210    #include <minix/callnr.h>
21211    #include <minix/com.h>
21212    #include <minix/keymap.h>
21213    #include "tty.h"
21214    #include "keymaps/us-std.src"
21215
21216    /* Standard and AT keyboard.  (PS/2 MCA implies AT throughout.) */
21217    #define KEYBD           0x60    /* I/O port for keyboard data */
21218
21219    /* AT keyboard. */
21220    #define KB_COMMAND      0x64    /* I/O port for commands on AT */
21221    #define KB_GATE_A20     0x02    /* bit in output port to enable A20 line */
21222    #define KB_PULSE_OUTPUT 0xF0    /* base for commands to pulse output port */
21223    #define KB_RESET        0x01    /* bit in output port to reset CPU */
21224    #define KB_STATUS       0x64    /* I/O port for status on AT */
21225    #define KB_ACK          0xFA    /* keyboard ack response */
21226    #define KB_BUSY         0x02    /* status bit set when KEYBD port ready */
21227    #define LED_CODE        0xED    /* command to keyboard to set LEDs */
21228    #define MAX_KB_ACK_RETRIES 0x1000       /* max #times to wait for kb ack */
21229    #define MAX_KB_BUSY_RETRIES 0x1000      /* max #times to loop while kb busy */
21230    #define KBIT            0x80    /* bit used to ack characters to keyboard */
21231
21232    /* Miscellaneous. */
21233    #define ESC_SCAN           1    /* Reboot key when panicking */
21234    #define SLASH_SCAN        53    /* to recognize numeric slash */
21235    #define HOME_SCAN         71    /* first key on the numeric keypad */
21236    #define DEL_SCAN          83    /* DEL for use in CTRL-ALT-DEL reboot */
21237    #define CONSOLE            0    /* line number for console */
21238    #define MEMCHECK_ADR   0x472    /* address to stop memory check after reboot */
21239    #define MEMCHECK_MAG  0x1234    /* magic number to stop memory check */
21240
21241    #define kb_addr()       (&kb_lines[0])  /* there is only one keyboard */
21242    #define KB_IN_BYTES       32    /* size of keyboard input buffer */
21243
21244    PRIVATE int alt1;               /* left alt key state */
21245    PRIVATE int alt2;               /* right alt key state */
21246    PRIVATE int capslock;           /* caps lock key state */
21247    PRIVATE int esc;                /* escape scan code detected? */
21248    PRIVATE int control;            /* control key state */
21249    PRIVATE int caps_off;           /* 1 = normal position, 0 = depressed */
21250    PRIVATE int numlock;            /* number lock key state */
21251    PRIVATE int num_off;            /* 1 = normal position, 0 = depressed */
21252    PRIVATE int slock;              /* scroll lock key state */
21253    PRIVATE int slock_off;          /* 1 = normal position, 0 = depressed */
21254    PRIVATE int shift;              /* shift key state */
```

```
21255
21256   PRIVATE char numpad_map[] =
21257                   {'H', 'Y', 'A', 'B', 'D', 'C', 'V', 'U', 'G', 'S', 'T', '@'};
21258
21259   /* Keyboard structure, 1 per console. */
21260   struct kb_s {
21261     char *ihead;                  /* next free spot in input buffer */
21262     char *itail;                  /* scan code to return to TTY */
21263     int icount;                   /* # codes in buffer */
21264     char ibuf[KB_IN_BYTES];       /* input buffer */
21265   };
21266
21267   PRIVATE struct kb_s kb_lines[NR_CONS];
21268
21269   FORWARD _PROTOTYPE( int kb_ack, (void) );
21270   FORWARD _PROTOTYPE( int kb_wait, (void) );
21271   FORWARD _PROTOTYPE( int func_key, (int scode) );
21272   FORWARD _PROTOTYPE( int scan_keyboard, (void) );
21273   FORWARD _PROTOTYPE( unsigned make_break, (int scode) );
21274   FORWARD _PROTOTYPE( void set_leds, (void) );
21275   FORWARD _PROTOTYPE( int kbd_hw_int, (int irq) );
21276   FORWARD _PROTOTYPE( void kb_read, (struct tty *tp) );
21277   FORWARD _PROTOTYPE( unsigned map_key, (int scode) );
21278
21279
21280   /*==========================================================================*
21281    *                               map_key0                                   *
21282    *==========================================================================*/
21283   /* Map a scan code to an ASCII code ignoring modifiers. */
21284   #define map_key0(scode)  \
21285           ((unsigned) keymap[(scode) * MAP_COLS])
21286
21287
21288   /*==========================================================================*
21289    *                               map_key                                    *
21290    *==========================================================================*/
21291   PRIVATE unsigned map_key(scode)
21292   int scode;
21293   {
21294   /* Map a scan code to an ASCII code. */
21295
21296     int caps, column;
21297     u16_t *keyrow;
21298
21299     if (scode == SLASH_SCAN && esc) return '/';   /* don't map numeric slash */
21300
21301     keyrow = &keymap[scode * MAP_COLS];
21302
21303     caps = shift;
21304     if (numlock && HOME_SCAN <= scode && scode <= DEL_SCAN) caps = !caps;
21305     if (capslock && (keyrow[0] & HASCAPS)) caps = !caps;
21306
21307     if (alt1 || alt2) {
21308           column = 2;
21309           if (control || alt2) column = 3;        /* Ctrl + Alt1 == Alt2 */
21310           if (caps) column = 4;
21311     } else {
21312           column = 0;
21313           if (caps) column = 1;
21314           if (control) column = 5;
```

```
21315       }
21316       return keyrow[column] & ~HASCAPS;
21317     }


21320     /*==========================================================================*
21321      *                              kbd_hw_int                                  *
21322      *==========================================================================*/
21323     PRIVATE int kbd_hw_int(irq)
21324     int irq;
21325     {
21326     /* A keyboard interrupt has occurred.  Process it. */
21327
21328       int code;
21329       unsigned km;
21330       register struct kb_s *kb;
21331
21332       /* Fetch the character from the keyboard hardware and acknowledge it. */
21333       code = scan_keyboard();
21334
21335       /* The IBM keyboard interrupts twice per key, once when depressed, once when
21336        * released.  Filter out the latter, ignoring all but the shift-type keys.
21337        * The shift-type keys 29, 42, 54, 56, 58, and 69 must be processed normally.
21338        */
21339
21340       if (code & 0200) {
21341             /* A key has been released (high bit is set). */
21342             km = map_key0(code & 0177);
21343             if (km != CTRL && km != SHIFT && km != ALT && km != CALOCK
21344                             && km != NLOCK && km != SLOCK && km != EXTKEY)
21345                   return 1;
21346       }
21347
21348       /* Store the character in memory so the task can get at it later. */
21349       kb = kb_addr();
21350       if (kb->icount < KB_IN_BYTES) {
21351             *kb->ihead++ = code;
21352             if (kb->ihead == kb->ibuf + KB_IN_BYTES) kb->ihead = kb->ibuf;
21353             kb->icount++;
21354             tty_table[current].tty_events = 1;
21355             force_timeout();
21356       }
21357       /* Else it doesn't fit - discard it. */
21358       return 1;     /* Reenable keyboard interrupt */
21359     }


21362     /*==========================================================================*
21363      *                              kb_read                                     *
21364      *==========================================================================*/
21365     PRIVATE void kb_read(tp)
21366     tty_t *tp;
21367     {
21368     /* Process characters from the circular keyboard buffer. */
21369
21370       struct kb_s *kb;
21371       char buf[3];
21372       int scode;
21373       unsigned ch;
21374
```

```
21375     kb = kb_addr();
21376     tp = &tty_table[current];             /* always use the current console */
21377
21378     while (kb->icount > 0) {
21379           scode = *kb->itail++;                   /* take one key scan code */
21380           if (kb->itail == kb->ibuf + KB_IN_BYTES) kb->itail = kb->ibuf;
21381           lock();
21382           kb->icount--;
21383           unlock();
21384
21385           /* Function keys are being used for debug dumps. */
21386           if (func_key(scode)) continue;
21387
21388           /* Perform make/break processing. */
21389           ch = make_break(scode);
21390
21391           if (ch <= 0xFF) {
21392                   /* A normal character. */
21393                   buf[0] = ch;
21394                   (void) in_process(tp, buf, 1);
21395           } else
21396           if (HOME <= ch && ch <= INSRT) {
21397                   /* An ASCII escape sequence generated by the numeric pad. */
21398                   buf[0] = ESC;
21399                   buf[1] = '[';
21400                   buf[2] = numpad_map[ch - HOME];
21401                   (void) in_process(tp, buf, 3);
21402           } else
21403           if (ch == ALEFT) {
21404                   /* Choose lower numbered console as current console. */
21405                   select_console(current - 1);
21406           } else
21407           if (ch == ARIGHT) {
21408                   /* Choose higher numbered console as current console. */
21409                   select_console(current + 1);
21410           } else
21411           if (AF1 <= ch && ch <= AF12) {
21412                   /* Alt-F1 is console, Alt-F2 is ttyc1, etc. */
21413                   select_console(ch - AF1);
21414           }
21415     }
21416   }


21419   /*===========================================================================*
21420    *                               make_break                                  *
21421    *===========================================================================*/
21422   PRIVATE unsigned make_break(scode)
21423   int scode;                      /* scan code of key just struck or released */
21424   {
21425   /* This routine can handle keyboards that interrupt only on key depression,
21426    * as well as keyboards that interrupt on key depression and key release.
21427    * For efficiency, the interrupt routine filters out most key releases.
21428    */
21429     int ch, make;
21430     static int CAD_count = 0;
21431
21432     /* Check for CTRL-ALT-DEL, and if found, halt the computer. This would
21433      * be better done in keyboard() in case TTY is hung, except control and
21434      * alt are set in the high level code.
```

```
21435       */
21436      if (control && (alt1 || alt2) && scode == DEL_SCAN)
21437      {
21438            if (++CAD_count == 3) wreboot(RBT_HALT);
21439            cause_sig(INIT_PROC_NR, SIGABRT);
21440            return -1;
21441      }
21442
21443      /* High-order bit set on key release. */
21444      make = (scode & 0200 ? 0 : 1);        /* 0 = release, 1 = press */
21445
21446      ch = map_key(scode & 0177);           /* map to ASCII */
21447
21448      switch (ch) {
21449            case CTRL:
21450                  control = make;
21451                  ch = -1;
21452                  break;
21453            case SHIFT:
21454                  shift = make;
21455                  ch = -1;
21456                  break;
21457            case ALT:
21458                  if (make) {
21459                        if (esc) alt2 = 1; else alt1 = 1;
21460                  } else {
21461                        alt1 = alt2 = 0;
21462                  }
21463                  ch = -1;
21464                  break;
21465            case CALOCK:
21466                  if (make && caps_off) {
21467                        capslock = 1 - capslock;
21468                        set_leds();
21469                  }
21470                  caps_off = 1 - make;
21471                  ch = -1;
21472                  break;
21473            case NLOCK:
21474                  if (make && num_off) {
21475                        numlock = 1 - numlock;
21476                        set_leds();
21477                  }
21478                  num_off = 1 - make;
21479                  ch = -1;
21480                  break;
21481            case SLOCK:
21482                  if (make & slock_off) {
21483                        slock = 1 - slock;
21484                        set_leds();
21485                  }
21486                  slock_off = 1 - make;
21487                  ch = -1;
21488                  break;
21489            case EXTKEY:
21490                  esc = 1;
21491                  return(-1);
21492            default:
21493                  if (!make) ch = -1;
21494      }
```

```
21495     esc = 0;
21496     return(ch);
21497   }


21500   /*==========================================================================*
21501    *                              set_leds                                    *
21502    *==========================================================================*/
21503   PRIVATE void set_leds()
21504   {
21505   /* Set the LEDs on the caps lock and num lock keys */
21506
21507     unsigned leds;
21508
21509     if (!pc_at) return;   /* PC/XT doesn't have LEDs */
21510
21511     /* encode LED bits */
21512     leds = (slock << 0) | (numlock << 1) | (capslock << 2);
21513
21514     kb_wait();                  /* wait for buffer empty  */
21515     out_byte(KEYBD, LED_CODE);  /* prepare keyboard to accept LED values */
21516     kb_ack();                   /* wait for ack response  */
21517
21518     kb_wait();                  /* wait for buffer empty  */
21519     out_byte(KEYBD, leds);      /* give keyboard LED values */
21520     kb_ack();                   /* wait for ack response  */
21521   }


21524   /*==========================================================================*
21525    *                              kb_wait                                     *
21526    *==========================================================================*/
21527   PRIVATE int kb_wait()
21528   {
21529   /* Wait until the controller is ready; return zero if this times out. */
21530
21531     int retries;
21532
21533     retries = MAX_KB_BUSY_RETRIES + 1;
21534     while (--retries != 0 && in_byte(KB_STATUS) & KB_BUSY)
21535           ;                     /* wait until not busy */
21536     return(retries);            /* nonzero if ready */
21537   }


21540   /*==========================================================================*
21541    *                              kb_ack                                      *
21542    *==========================================================================*/
21543   PRIVATE int kb_ack()
21544   {
21545   /* Wait until kbd acknowledges last command; return zero if this times out. */
21546
21547     int retries;
21548
21549     retries = MAX_KB_ACK_RETRIES + 1;
21550     while (--retries != 0 && in_byte(KEYBD) != KB_ACK)
21551           ;                     /* wait for ack */
21552     return(retries);            /* nonzero if ack received */
21553   }
```

```
21555   /*===========================================================================*
21556    *                              kb_init                                      *
21557    *===========================================================================*/
21558   PUBLIC void kb_init(tp)
21559   tty_t *tp;
21560   {
21561   /* Initialize the keyboard driver. */
21562
21563     register struct kb_s *kb;
21564
21565     /* Input function. */
21566     tp->tty_devread = kb_read;
21567
21568     kb = kb_addr();
21569
21570     /* Set up input queue. */
21571     kb->ihead = kb->itail = kb->ibuf;
21572
21573     /* Set initial values. */
21574     caps_off = 1;
21575     num_off = 1;
21576     slock_off = 1;
21577     esc = 0;
21578
21579     set_leds();                 /* turn off numlock led */
21580
21581     scan_keyboard();            /* stop lockup from leftover keystroke */
21582
21583     put_irq_handler(KEYBOARD_IRQ, kbd_hw_int);    /* set the interrupt handler */
21584     enable_irq(KEYBOARD_IRQ);     /* safe now everything initialised! */
21585   }


21588   /*===========================================================================*
21589    *                              kbd_loadmap                                  *
21590    *===========================================================================*/
21591   PUBLIC int kbd_loadmap(user_phys)
21592   phys_bytes user_phys;
21593   {
21594   /* Load a new keymap. */
21595
21596     phys_copy(user_phys, vir2phys(keymap), (phys_bytes) sizeof(keymap));
21597     return(OK);
21598   }


21601   /*===========================================================================*
21602    *                              func_key                                     *
21603    *===========================================================================*/
21604   PRIVATE int func_key(scode)
21605   int scode;                    /* scan code for a function key */
21606   {
21607   /* This procedure traps function keys for debugging and control purposes. */
21608
21609     unsigned code;
21610
21611     code = map_key0(scode);                     /* first ignore modifiers */
21612     if (code < F1 || code > F12) return(FALSE);   /* not our job */
21613
21614     switch (map_key(scode)) {                   /* include modifiers */
```

```
21615
21616     case F1:      p_dmp(); break;         /* print process table */
21617     case F2:      map_dmp(); break;       /* print memory map */
21618     case F3:      toggle_scroll(); break; /* hardware vs. software scrolling */
21619
21620   #if ENABLE_NETWORKING
21621     case F5:      dp_dump(); break;              /* network statistics */
21622   #endif
21623     case CF7:     sigchar(&tty_table[CONSOLE], SIGQUIT); break;
21624     case CF8:     sigchar(&tty_table[CONSOLE], SIGINT); break;
21625     case CF9:     sigchar(&tty_table[CONSOLE], SIGKILL); break;
21626     default:      return(FALSE);
21627     }
21628     return(TRUE);
21629   }


21632   /*==========================================================================*
21633    *                              scan_keyboard                               *
21634    *==========================================================================*/
21635   PRIVATE int scan_keyboard()
21636   {
21637   /* Fetch the character from the keyboard hardware and acknowledge it. */
21638
21639     int code;
21640     int val;
21641
21642     code = in_byte(KEYBD);        /* get the scan code for the key struck */
21643     val = in_byte(PORT_B);        /* strobe the keyboard to ack the char */
21644     out_byte(PORT_B, val | KBIT); /* strobe the bit high */
21645     out_byte(PORT_B, val);        /* now strobe it low */
21646     return code;
21647   }


21650   /*==========================================================================*
21651    *                              wreboot                                     *
21652    *==========================================================================*/
21653   PUBLIC void wreboot(how)
21654   int how;                /* 0 = halt, 1 = reboot, 2 = panic!, ... */
21655   {
21656   /* Wait for keystrokes for printing debugging info and reboot. */
21657
21658     int quiet, code;
21659     static u16_t magic = MEMCHECK_MAG;
21660     struct tasktab *ttp;
21661
21662     /* Mask all interrupts. */
21663     out_byte(INT_CTLMASK, ~0);
21664
21665     /* Tell several tasks to stop. */
21666     cons_stop();
21667   #if ENABLE_NETWORKING
21668     dp8390_stop();
21669   #endif
21670     floppy_stop();
21671     clock_stop();
21672
21673     if (how == RBT_HALT) {
21674           printf("System Halted\n");
```

```
21675           if (!mon_return) how = RBT_PANIC;
21676       }
21677
21678       if (how == RBT_PANIC) {
21679           /* A panic! */
21680           printf("Hit ESC to reboot, F-keys for debug dumps\n");
21681
21682           (void) scan_keyboard(); /* ack any old input */
21683           quiet = scan_keyboard();/* quiescent value (0 on PC, last code on AT)*/
21684           for (;;) {
21685                   milli_delay(100);       /* pause for a decisecond */
21686                   code = scan_keyboard();
21687                   if (code != quiet) {
21688                           /* A key has been pressed. */
21689                           if (code == ESC_SCAN) break; /* reboot if ESC typed */
21690                           (void) func_key(code);       /* process function key */
21691                           quiet = scan_keyboard();
21692                   }
21693           }
21694           how = RBT_REBOOT;
21695       }
21696
21697       if (how == RBT_REBOOT) printf("Rebooting\n");
21698
21699       if (mon_return && how != RBT_RESET) {
21700           /* Reinitialize the interrupt controllers to the BIOS defaults. */
21701           intr_init(0);
21702           out_byte(INT_CTLMASK, 0);
21703           out_byte(INT2_CTLMASK, 0);
21704
21705           /* Return to the boot monitor. */
21706           if (how == RBT_HALT) {
21707                   reboot_code = vir2phys("");
21708           } else
21709           if (how == RBT_REBOOT) {
21710                   reboot_code = vir2phys("delay;boot");
21711           }
21712           level0(monitor);
21713       }
21714
21715       /* Stop BIOS memory test. */
21716       phys_copy(vir2phys(&magic), (phys_bytes) MEMCHECK_ADR,
21717                                                   (phys_bytes) sizeof(magic));
21718
21719       if (protected_mode) {
21720           /* Use the AT keyboard controller to reset the processor.
21721            * The A20 line is kept enabled in case this code is ever
21722            * run from extended memory, and because some machines
21723            * appear to drive the fake A20 high instead of low just
21724            * after reset, leading to an illegal opode trap.  This bug
21725            * is more of a problem if the fake A20 is in use, as it
21726            * would be if the keyboard reset were used for real mode.
21727            */
21728           kb_wait();
21729           out_byte(KB_COMMAND,
21730                    KB_PULSE_OUTPUT | (0x0F & ~(KB_GATE_A20 | KB_RESET)));
21731           milli_delay(10);
21732
21733           /* If the nice method fails then do a reset.  In protected
21734            * mode this means a processor shutdown.
```

```
21735             */
21736            printf("Hard reset...\n");
21737            milli_delay(250);
21738      }
21739      /* In real mode, jumping to the reset address is good enough. */
21740      level0(reset);
21741    }
```

++
src/kernel/main.c
++

```
21800    /* This file contains the main program of MINIX.  The routine main()
21801     * initializes the system and starts the ball rolling by setting up the proc
21802     * table, interrupt vectors, and scheduling each task to run to initialize
21803     * itself.
21804     *
21805     * The entries into this file are:
21806     *   main:              MINIX main program
21807     *   panic:             abort MINIX due to a fatal error
21808     */
21809
21810    #include "kernel.h"
21811    #include <signal.h>
21812    #include <unistd.h>
21813    #include <minix/callnr.h>
21814    #include <minix/com.h>
21815    #include "proc.h"
21816
21817
21818    /*===========================================================================*
21819     *                                    main                                   *
21820     *===========================================================================*/
21821    PUBLIC void main()
21822    {
21823    /* Start the ball rolling. */
21824
21825      register struct proc *rp;
21826      register int t;
21827      int sizeindex;
21828      phys_clicks text_base;
21829      vir_clicks text_clicks;
21830      vir_clicks data_clicks;
21831      phys_bytes phys_b;
21832      reg_t ktsb;                   /* kernel task stack base */
21833      struct memory *memp;
21834      struct tasktab *ttp;
21835
21836      /* Initialize the interrupt controller. */
21837      intr_init(1);
21838
21839      /* Interpret memory sizes. */
21840      mem_init();
21841
21842      /* Clear the process table.
21843       * Set up mappings for proc_addr() and proc_number() macros.
21844       */
```

```
21845     for (rp = BEG_PROC_ADDR, t = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++t) {
21846           rp->p_flags = P_SLOT_FREE;
21847           rp->p_nr = t;           /* proc number from ptr */
21848           (pproc_addr + NR_TASKS)[t] = rp;        /* proc ptr from number */
21849     }
21850
21851     /* Set up proc table entries for tasks and servers.  The stacks of the
21852      * kernel tasks are initialized to an array in data space.  The stacks
21853      * of the servers have been added to the data segment by the monitor, so
21854      * the stack pointer is set to the end of the data segment.  All the
21855      * processes are in low memory on the 8086.  On the 386 only the kernel
21856      * is in low memory, the rest if loaded in extended memory.
21857      */
21858
21859     /* Task stacks. */
21860     ktsb = (reg_t) t_stack;
21861
21862     for (t = -NR_TASKS; t <= LOW_USER; ++t) {
21863           rp = proc_addr(t);                      /* t's process slot */
21864           ttp = &tasktab[t + NR_TASKS];           /* t's task attributes */
21865           strcpy(rp->p_name, ttp->name);
21866           if (t < 0) {
21867                   if (ttp->stksize > 0) {
21868                           rp->p_stguard = (reg_t *) ktsb;
21869                           *rp->p_stguard = STACK_GUARD;
21870                   }
21871                   ktsb += ttp->stksize;
21872                   rp->p_reg.sp = ktsb;
21873                   text_base = code_base >> CLICK_SHIFT;
21874                                       /* tasks are all in the kernel */
21875                   sizeindex = 0;          /* and use the full kernel sizes */
21876                   memp = &mem[0];         /* remove from this memory chunk */
21877           } else {
21878                   sizeindex = 2 * t + 2;  /* MM, FS, INIT have their own sizes */
21879           }
21880           rp->p_reg.pc = (reg_t) ttp->initial_pc;
21881           rp->p_reg.psw = istaskp(rp) ? INIT_TASK_PSW : INIT_PSW;
21882
21883           text_clicks = sizes[sizeindex];
21884           data_clicks = sizes[sizeindex + 1];
21885           rp->p_map[T].mem_phys = text_base;
21886           rp->p_map[T].mem_len  = text_clicks;
21887           rp->p_map[D].mem_phys = text_base + text_clicks;
21888           rp->p_map[D].mem_len  = data_clicks;
21889           rp->p_map[S].mem_phys = text_base + text_clicks + data_clicks;
21890           rp->p_map[S].mem_vir  = data_clicks;    /* empty - stack is in data */
21891           text_base += text_clicks + data_clicks; /* ready for next, if server */
21892           memp->size -= (text_base - memp->base);
21893           memp->base = text_base;                 /* memory no longer free */
21894
21895           if (t >= 0) {
21896                   /* Initialize the server stack pointer.  Take it down one word
21897                    * to give crtso.s something to use as "argc".
21898                    */
21899                   rp->p_reg.sp = (rp->p_map[S].mem_vir +
21900                                   rp->p_map[S].mem_len) << CLICK_SHIFT;
21901                   rp->p_reg.sp -= sizeof(reg_t);
21902           }
21903
21904   #if _WORD_SIZE == 4
```

```
21905           /* Servers are loaded in extended memory if in 386 mode. */
21906           if (t < 0) {
21907                   memp = &mem[1];
21908                   text_base = 0x100000 >> CLICK_SHIFT;
21909           }
21910   #endif
21911           if (!isidlehardware(t)) lock_ready(rp); /* IDLE, HARDWARE nevereadv */
21912           rp->p_flags = 0;
21913
21914           alloc_segments(rp);
21915     }
21916
21917     proc[NR_TASKS+INIT_PROC_NR].p_pid = 1;/* INIT of course has pid 1 */
21918     bill_ptr = proc_addr(IDLE);           /* it has to point somewhere */
21919     lock_pick_proc();
21920
21921     /* Now go to the assembly code to start running the current process. */
21922     restart();
21923   }


21926   /*===========================================================================*
21927    *                                panic                                      *
21928    *===========================================================================*/
21929   PUBLIC void panic(s,n)
21930   _CONST char *s;
21931   int n;
21932   {
21933   /* The system has run aground of a fatal error.  Terminate execution.
21934    * If the panic originated in MM or FS, the string will be empty and the
21935    * file system already syncked.  If the panic originates in the kernel, we are
21936    * kind of stuck.
21937    */
21938
21939     if (*s != 0) {
21940           printf("\nKernel panic: %s",s);
21941           if (n != NO_NUM) printf(" %d", n);
21942           printf("\n");
21943     }
21944     wreboot(RBT_PANIC);
21945   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/mcd.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

22000   /* This file contains the driver for a Mitsumi cdrom controller.
22001    *
22002    * The file contains one entry point:
22003    *
22004    *   mcd_task:          main entry when system is brought up
22005    *
22006    *  Mar 30 1995                Author: Michel R. Prevenier
22007    */
22008
22009
```

```
22010    #include "kernel.h"
22011    #include "driver.h"
22012    #include "drvlib.h"
22013    #include <minix/cdrom.h>
22014    #include <sys/ioctl.h>
22015
22016    #if ENABLE_MITSUMI_CDROM
22017
22018    #define MCD_DEBUG               0       /* debug level */
22019
22020    /* Default IRQ. */
22021    #define MCD_IRQ                 10
22022
22023    /* Default I/O ports (offset from base address */
22024    #define MCD_IO_BASE_ADDRESS     0x300
22025    #define MCD_DATA_PORT           (mcd_io_base+0)
22026    #define MCD_FLAG_PORT           (mcd_io_base+1)
22027    #define MCD_CONTROL_PORT        (mcd_io_base+2)
22028
22029
22030    /* Miscellaneous constants. */
22031    #define MCD_SKIP                150     /* Skip first 150 blocks on cdrom */
22032    #define MCD_BLOCK_SIZE          2048    /* Block size in cooked mode */
22033    #define MCD_BLOCK_SHIFT         11      /* for division */
22034    #define MCD_BLOCK_MASK          2047    /* and remainder */
22035    #define BYTES_PER_SECTOR        2048    /* Nr. of bytes in a sector */
22036    #define SECTORS_PER_SECOND      75      /* Nr. of sectors in a second */
22037    #define SECONDS_PER_MINUTE      60      /* You never know, things change :-) */
22038    #define MCD_RETRIES             2       /* Number of retries for a command */
22039    #define REPLY_DELAY             5000    /* Count to wait for a reply */
22040    #define MAX_TRACKS              104     /* Maximum nr. of tracks */
22041    #define LEAD_OUT                0xAA    /* Lead out track is always 0xAA */
22042    #define SUB_PER_DRIVE   (NR_PARTITIONS * NR_PARTITIONS)
22043
22044
22045    /* Drive commands */
22046    #define MCD_GET_VOL_INFO        0x10    /* Read volume information */
22047    #define MCD_GET_Q_CHANNEL       0x20    /* Read q-channel information */
22048    #define MCD_GET_STATUS          0x40    /* Read status of drive */
22049    #define MCD_SET_MODE            0x50    /* Set transmission mode */
22050    #define MCD_RESET               0x60    /* Reset controller */
22051    #define MCD_STOP_AUDIO          0x70    /* Stop audio playing */
22052    #define MCD_SET_DRIVE_MODE      0xA0    /* Set drive mode */
22053    #define MCD_READ_FROM_TO        0xC0    /* Read from .. to .. */
22054    #define MCD_GET_VERSION         0xDC    /* Get version number */
22055    #define MCD_STOP                0xF0    /* Stop everything */
22056    #define MCD_EJECT               0xF6    /* Eject cd */
22057    #define MCD_PICKLE              0x04    /* Needed for newer drive models */
22058
22059
22060    /* Command bits for MCD_SET_MODE command */
22061    #define MCD_MUTE_DATA           0x01    /* 1 = Don't play back data as audio */
22062    #define MCD_GET_TOC             0x04    /* 0 = Get toc on next GET_Q_CHANNEL */
22063    #define MCD_ECC_MODE            0x20    /* 0 = Use secondary ecc */
22064    #define MCD_DATALENGTH          0x40    /* 0 = Read user data only */
22065    #define MCD_COOKED              (MCD_MUTE_DATA)
22066    #define MCD_TOC                 (MCD_MUTE_DATA | MCD_GET_TOC)
22067
22068
22069    /* Status bits */
```

```
22070   #define MCD_CMD_ERROR           0x01    /* Command error */
22071   #define MCD_AUDIO_BUSY          0x02    /* Audio disk is playing */
22072   #define MCD_READ_ERROR          0x04    /* Read error */
22073   #define MCD_AUDIO_DISK          0x08    /* Audio disk is in */
22074   #define MCD_SPINNING            0x10    /* Motor is spinning */
22075   #define MCD_DISK_CHANGED        0x20    /* Disk has been removed or changed */
22076   #define MCD_DISK_IN             0x40    /* Disk is in */
22077   #define MCD_DOOR_OPEN           0x80    /* Door is open */
22078
22079   /* Flag bits */
22080   #define MCD_DATA_AVAILABLE      0x02    /* Data available */
22081   #define MCD_BUSY                0x04    /* Drive is busy */
22082
22083   /* Function prototypes */
22084   FORWARD _PROTOTYPE ( int mcd_init, (void));
22085   FORWARD _PROTOTYPE ( int c_handler, (int irq));
22086   FORWARD _PROTOTYPE ( int mcd_play_mss, (struct cd_play_mss));
22087   FORWARD _PROTOTYPE ( int mcd_play_tracks, (struct cd_play_track tracks));
22088   FORWARD _PROTOTYPE ( int mcd_stop, (void));
22089   FORWARD _PROTOTYPE ( int mcd_eject, (void));
22090   FORWARD _PROTOTYPE ( int mcd_pause, (void));
22091   FORWARD _PROTOTYPE ( int mcd_resume, (void));
22092   FORWARD _PROTOTYPE ( u8_t bin2bcd, (u8_t b));
22093   FORWARD _PROTOTYPE ( void bcd2bin, (u8_t *bcd));
22094   FORWARD _PROTOTYPE ( long mss2block, (u8_t *mss));
22095   FORWARD _PROTOTYPE ( void block2mss, (long block, u8_t *mss));
22096   FORWARD _PROTOTYPE ( int mcd_get_reply, (u8_t *reply, int delay));
22097   FORWARD _PROTOTYPE ( int mcd_get_status, (int f));
22098   FORWARD _PROTOTYPE ( int mcd_ready, (int delay));
22099   FORWARD _PROTOTYPE ( int mcd_data_ready, (int delay));
22100   FORWARD _PROTOTYPE ( int mcd_set_mode, (int mode));
22101   FORWARD _PROTOTYPE ( int mcd_send_command, (int command));
22102   FORWARD _PROTOTYPE ( int mcd_get_disk_info, (void));
22103   FORWARD _PROTOTYPE ( int mcd_read_q_channel, (struct cd_toc_entry *qc));
22104   FORWARD _PROTOTYPE ( int mcd_read_toc, (void));
22105   FORWARD _PROTOTYPE ( int ioctl_read_toc, (message *m_ptr));
22106   FORWARD _PROTOTYPE ( int ioctl_disk_info, (message *m_ptr));
22107   FORWARD _PROTOTYPE ( int ioctl_read_sub, (message *m_ptr));
22108   FORWARD _PROTOTYPE ( int ioctl_disk_info, (message *m_ptr));
22109   FORWARD _PROTOTYPE ( int ioctl_play_mss, (message *m_ptr));
22110   FORWARD _PROTOTYPE ( int ioctl_play_ti, (message *m_ptr));
22111   FORWARD _PROTOTYPE ( int mcd_open, (struct driver *dp, message *m_ptr));
22112   FORWARD _PROTOTYPE ( int mcd_close, (struct driver *dp, message *m_ptr));
22113   FORWARD _PROTOTYPE ( int mcd_ioctl, (struct driver *dp, message *m_ptr));
22114   FORWARD _PROTOTYPE ( char *mcd_name, (void));
22115   FORWARD _PROTOTYPE ( struct device *mcd_prepare, (int dev));
22116   FORWARD _PROTOTYPE ( int mcd_schedule, (int proc_nr, struct iorequest_s *iop));
22117   FORWARD _PROTOTYPE ( int mcd_finish, (void));
22118   FORWARD _PROTOTYPE ( void mcd_geometry, (struct partition *entry));
22119
22120
22121   /* Flags displaying current status of cdrom, used with the McdStatus variable */
22122   #define TOC_UPTODATE   0x001     /* Table of contents is up to date */
22123   #define INFO_UPTODATE  0x002     /* Disk info is up to date */
22124   #define DISK_CHANGED   0x004     /* Disk has changed */
22125   #define AUDIO_PLAYING  0x008     /* Cdrom is playing audio */
22126   #define AUDIO_PAUSED   0x010     /* Cdrom is paused (only audio) */
22127   #define AUDIO_DISK     0x020     /* Disk contains audio */
22128   #define DISK_ERROR     0x040     /* An error occured */
22129   #define NO_DISK        0x080     /* No disk in device */
```

```
22130
22131   /* Entry points to this driver. */
22132   PRIVATE struct driver mcd_dtab =
22133   {
22134   #if __minix_vmd
22135     NULL,         /* No private request buffer */
22136   #endif
22137     mcd_name,     /* Current device's name */
22138     mcd_open,     /* Open request read table of contents */
22139     mcd_close,    /* Release device */
22140     mcd_ioctl,    /* Do cdrom ioctls */
22141     mcd_prepare,  /* Prepare for I/O */
22142     mcd_schedule, /* Precompute blocks */
22143     mcd_finish,   /* Do the I/O */
22144     nop_cleanup,  /* No cleanup to do */
22145     mcd_geometry  /* Tell geometry */
22146   };
22147
22148
22149   PRIVATE struct trans
22150   {
22151     struct iorequest_s *tr_iop;   /* Belongs to this I/O request */
22152     unsigned long tr_pos;         /* Byte position to transfer from */
22153     int tr_count;                 /* Byte count */
22154     phys_bytes tr_phys;           /* User physical address */
22155   } mcd_trans[NR_IOREQS];
22156
22157
22158   /* Globals */
22159   #if __minix_vmd
22160   PRIVATE int mcd_tasknr = ANY;
22161   #endif
22162   PRIVATE int mcd_avail;                /* Set if Mitsumi device exists */
22163   PRIVATE int mcd_irq;                  /* Interrupt request line */
22164   PRIVATE int mcd_io_base;              /* I/O base register */
22165   PRIVATE struct device *mcd_dv;        /* Active partition */
22166   PRIVATE struct trans *mcd_tp;         /* Pointer to add transfer requests */
22167   PRIVATE unsigned mcd_count;           /* Number of bytes to transfer */
22168   PRIVATE unsigned long mcd_nextpos;    /* Next consecutive position on disk */
22169   PRIVATE struct device mcd_part[DEV_PER_DRIVE];
22170                                         /* Primary partitions: cd[0-4] */
22171   PRIVATE struct device mcd_subpart[SUB_PER_DRIVE];
22172                                         /* Subpartitions: cd[1-4][a-d] */
22173   PRIVATE int mcd_open_ct;              /* in-use count */
22174   PRIVATE int McdStatus = NO_DISK;      /* A new (or no) disk is inserted */
22175   PRIVATE struct cd_play_mss PlayMss;   /* Keep track of where we are if we
22176                                            pause, used by resume */
22177   PRIVATE struct cd_disk_info DiskInfo; /* Contains toc header */
22178   PRIVATE struct cd_toc_entry Toc[MAX_TRACKS];  /* Buffer for toc */
22179
22180
22181
22182   /*==========================================================================*
22183    *                              mcd_task                                    *
22184    *==========================================================================*/
22185   PUBLIC void mcd_task()
22186   {
22187     long v;
22188     static char var[] = "MCD";
22189     static char fmt[] = "x:d";
```

```
22190
22191   #if __minix_vmd
22192     mcd_tasknr = proc_number(proc_ptr);
22193   #endif
22194
22195     /* Configure I/O base and IRQ. */
22196     v = MCD_IO_BASE_ADDRESS;
22197     (void) env_parse(var, fmt, 0, &v, 0x000L, 0x3FFL);
22198     mcd_io_base = v;
22199
22200     v = MCD_IRQ;
22201     (void) env_parse(var, fmt, 0, &v, 0L, (long) NR_IRQ_VECTORS - 1);
22202     mcd_irq = v;
22203
22204     driver_task(&mcd_dtab);       /* Start driver task for cdrom */
22205   }


22208   /*===========================================================================*
22209    *                              mcd_open                                     *
22210    *===========================================================================*/
22211   PRIVATE int mcd_open(dp, m_ptr)
22212   struct driver *dp;      /* pointer to this drive */
22213   message *m_ptr;         /* OPEN */
22214   {
22215     int i, status;
22216
22217     if (!mcd_avail && mcd_init() != OK) return EIO;
22218
22219     if (mcd_prepare(m_ptr->DEVICE) == NIL_DEV) return ENXIO;
22220
22221     /* A CD-ROM is read-only by definition. */
22222     if (m_ptr->COUNT & W_BIT) return EACCES;
22223
22224     if (mcd_open_ct == 0)
22225     {
22226       i = 20;
22227       for (;;) {
22228         if (mcd_get_status(1) == -1) return EIO;   /* set McdStatus flags */
22229         if (!(McdStatus & NO_DISK)) break;
22230         if (--i == 0) return EIO;
22231         milli_delay(100);
22232       }
22233
22234       /* Try to read the table of contents of the CD currently inserted */
22235       if ((status = mcd_read_toc()) != OK)
22236         return status;
22237
22238       mcd_open_ct++;
22239
22240       /* fill in size of device (= nr. of bytes on the disk) */
22241       mcd_part[0].dv_base = 0;
22242       mcd_part[0].dv_size =
22243        ((((unsigned long)DiskInfo.disk_length_mss[MINUTES] * SECONDS_PER_MINUTE
22244         + (unsigned long)DiskInfo.disk_length_mss[SECONDS]) * SECTORS_PER_SECOND)
22245         + (unsigned long)DiskInfo.disk_length_mss[SECTOR]) * BYTES_PER_SECTOR;
22246
22247   #if MCD_DEBUG >= 1
22248       printf("cd size: %lu\n", mcd_part[0].dv_size);
22249   #endif
```

```
22250
22251         /* Partition the disk. */
22252         partition(&mcd_dtab, 0, P_PRIMARY);
22253       }
22254       return OK;
22255     }


22258     /*===========================================================================*
22259      *                              mcd_close                                    *
22260      *===========================================================================*/
22261     PRIVATE int mcd_close(dp, m_ptr)
22262     struct driver *dp;      /* pointer to this drive */
22263     message *m_ptr;         /* CLOSE */
22264     {
22265       /* One less reference to the device */
22266
22267       mcd_open_ct--;
22268       return OK;
22269     }


22272     /*===========================================================================*
22273      *                              mcd_name                                     *
22274      *===========================================================================*/
22275     PRIVATE char *mcd_name()
22276     {
22277       /* Return a name for the device */
22278
22279       return "cd0";
22280     }


22283     /*===========================================================================*
22284      *                              mcd_ioctl                                    *
22285      *===========================================================================*/
22286     PRIVATE int mcd_ioctl(dp, m_ptr)
22287     struct driver *dp;      /* pointer to the drive */
22288     message *m_ptr;         /* contains ioctl command */
22289     {
22290       /* Perform the ioctl request */
22291
22292       int status;
22293
22294       if (mcd_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
22295
22296       mcd_get_status(1);    /* Update the status flags */
22297       if ((McdStatus & NO_DISK) && m_ptr->REQUEST != CDIOEJECT)
22298         return EIO;
22299
22300       switch(m_ptr->REQUEST)
22301       {
22302         case CDIOPLAYMSS:     status = ioctl_play_mss(m_ptr);break;
22303         case CDIOPLAYTI:      status = ioctl_play_ti(m_ptr);break;
22304         case CDIOREADTOCHDR:  status = ioctl_disk_info(m_ptr);break;
22305         case CDIOREADTOC:     status = ioctl_read_toc(m_ptr);break;
22306         case CDIOREADSUBCH:   status = ioctl_read_sub(m_ptr);break;
22307         case CDIOSTOP:        status = mcd_stop();break;
22308         case CDIOPAUSE:       status = mcd_pause();break;
22309         case CDIORESUME:      status = mcd_resume();break;
```

```
22310       case CDIOEJECT:       status = mcd_eject();break;
22311       default:              status = do_diocntl(dp, m_ptr);
22312     }
22313     return status;
22314   }


22317   /*==========================================================================*
22318    *                              mcd_get_reply                               *
22319    *==========================================================================*/
22320   PRIVATE int mcd_get_reply(reply, delay)
22321   u8_t *reply;            /* variable to put reply in */
22322   int delay;              /* count to wait for the reply */
22323   {
22324     /* Get a reply from the drive */
22325
22326     if (mcd_ready(delay) != OK) return EIO;           /* wait for drive to
22327                                                          become available */
22328     *reply = in_byte(MCD_DATA_PORT);      /* get the reply */
22329     return OK;
22330   }


22333   /*==========================================================================*
22334    *                              mcd_ready                                   *
22335    *==========================================================================*/
22336   PRIVATE int mcd_ready(delay)
22337   int delay;   /* count to wait for drive to become available again */
22338   {
22339     /* Wait for drive to become available */
22340
22341     struct milli_state ms;
22342
22343     milli_start(&ms);
22344     do
22345     {
22346       if (!(in_byte(MCD_FLAG_PORT) & MCD_BUSY)) return OK; /* OK, drive ready */
22347     } while(milli_elapsed(&ms) < delay);
22348
22349     return EIO; /* Timeout */
22350   }


22353   /*==========================================================================*
22354    *                              mcd_data_ready                              *
22355    *==========================================================================*/
22356   PRIVATE int mcd_data_ready(delay)
22357   int delay;      /* count to wait for the data */
22358   {
22359     /* Wait for the drive to get the data */
22360
22361     struct milli_state ms;
22362
22363     milli_start(&ms);
22364     do
22365     {
22366       if (!(in_byte(MCD_FLAG_PORT) & 2)) return OK; /* OK, data is there */
22367     } while(milli_elapsed(&ms) < delay);
22368
22369     return EIO;  /* Timeout */
```

```
22370   }


22373   /*==========================================================================*
22374    *                              mcd_get_status                              *
22375    *==========================================================================*/
22376   PRIVATE int mcd_get_status(f)
22377   int f;          /* flag */
22378   {
22379     /* Return status info from the drive and update the global McdStatus */
22380
22381     u8_t status;
22382
22383     /* If f = 1, we first send a get_status command, otherwise we just get
22384        the status info from the drive */
22385
22386     if (f) out_byte(MCD_DATA_PORT, MCD_GET_STATUS);        /* Try to get status */
22387     if (mcd_get_reply(&status,REPLY_DELAY) != OK) return -1;
22388
22389     McdStatus &= ~(NO_DISK | DISK_CHANGED | DISK_ERROR);
22390
22391     /* Fill in the McdStatus variable */
22392     if (status & MCD_DOOR_OPEN ||
22393        !(status & MCD_DISK_IN))         McdStatus = NO_DISK;
22394     else if (status & MCD_DISK_CHANGED) McdStatus = DISK_CHANGED;
22395     else if (status & MCD_READ_ERROR ||
22396              status & MCD_CMD_ERROR)    McdStatus = DISK_ERROR;
22397     else
22398     {
22399       if (status & MCD_AUDIO_DISK)
22400       {
22401         McdStatus |= AUDIO_DISK;
22402         if (!(status & MCD_AUDIO_BUSY)) McdStatus &= ~(AUDIO_PLAYING);
22403         else McdStatus |= AUDIO_PLAYING;
22404       }
22405     }
22406   #if MCD_DEBUG >= 3
22407     printf("mcd_get_status(%d) = %02x, McdStatus = %02x\n",
22408           f, status, McdStatus);
22409   #endif
22410     return status;        /* Return status */
22411   }


22414   /*==========================================================================*
22415    *                              mcd_set_mode                                *
22416    *==========================================================================*/
22417   PRIVATE int mcd_set_mode(mode)
22418   int mode; /* new drive mode */
22419   {
22420     /* Set drive mode */
22421
22422     int i;
22423
22424     for (i = 0; i < MCD_RETRIES; i++)
22425     {
22426       out_byte(MCD_DATA_PORT, MCD_SET_MODE); /* Send set mode command */
22427       out_byte(MCD_DATA_PORT, mode);         /* Send which mode */
22428       if (mcd_get_status(0) != -1 &&
22429            !(McdStatus & DISK_ERROR)) break;
```

```
22430     }
22431     if (i == MCD_RETRIES) return EIO;        /* Operation failed */
22432
22433     return OK; /* Operation succeeded */
22434   }


22437   /*==========================================================================*
22438    *                              mcd_send_command                            *
22439    *==========================================================================*/
22440   PRIVATE int mcd_send_command(command)
22441   int command;    /* command to send */
22442   {
22443     int i;
22444
22445     for (i = 0; i < MCD_RETRIES; i++)
22446     {
22447       out_byte(MCD_DATA_PORT, command);      /* send command */
22448       if (mcd_get_status(0) != -1 &&
22449            !(McdStatus & DISK_ERROR)) break;
22450     }
22451     if (i == MCD_RETRIES) return EIO;        /* operation failed */
22452
22453     return OK;
22454   }


22457   /*==========================================================================*
22458    *                              mcd_init                                    *
22459    *==========================================================================*/
22460   PRIVATE int mcd_init()
22461   {
22462     /* Initialize the drive and get the version bytes, this is done only
22463        once when the system gets up. We can't use mcd_ready because
22464        the clock task is not available yet.
22465      */
22466
22467     u8_t version[3];
22468     int i;
22469     u32_t n;
22470     struct milli_state ms;
22471
22472     /* Reset the flag port and remove all pending data, if we do
22473      * not do this properly the drive won't cooperate.
22474      */
22475     out_byte(MCD_FLAG_PORT, 0x00);
22476     for (n = 0; n < 1000000; n++)
22477       (void) in_byte(MCD_FLAG_PORT);
22478
22479     /* Now see if the drive will report its status */
22480     if (mcd_get_status(1) == -1)
22481     {
22482       /* Too bad, drive will not listen */
22483       printf("%s: init failed, no Mitsumi cdrom present\n", mcd_name());
22484       return -1;
22485     }
22486
22487     /* Find out drive version */
22488     out_byte(MCD_DATA_PORT, MCD_GET_VERSION);
22489     milli_start(&ms);
```

```
22490       for (i = 0; i < 3; i++)
22491       {
22492         while (in_byte(MCD_FLAG_PORT) & MCD_BUSY)
22493         {
22494           if (milli_elapsed(&ms) >= 1000)
22495           {
22496             printf("%s: can't get version of Mitsumi cdrom\n", mcd_name());
22497             return -1;
22498           }
22499         }
22500         version[i] = in_byte(MCD_DATA_PORT);
22501       }
22502
22503       if (version[1] == 'D')
22504         printf("%s: Mitsumi FX001D CD-ROM\n", mcd_name());
22505       else
22506         printf("%s: Mitsumi CD-ROM version %02x%02x%02x\n", mcd_name(),
22507                 version[0], version[1], version[2]);
22508
22509       /* Newer drive models need this */
22510       if (version[1] >= 4) out_byte(MCD_CONTROL_PORT, MCD_PICKLE);
22511
22512       /* Register interrupt vector and enable interrupt
22513        * currently the interrupt is not used because
22514        * the controller isn't set up to do dma.  XXX
22515        */
22516       put_irq_handler(mcd_irq, c_handler);
22517       enable_irq(mcd_irq);
22518
22519       mcd_avail = 1;
22520       return OK;
22521     }


22524     /*==========================================================================*
22525      *                              c_handler                                   *
22526      *==========================================================================*/
22527     PRIVATE int c_handler(irq)
22528     int irq;        /* irq number */
22529     {
22530       /* The interrupt handler, I never got an interrupt but its here just
22531        * in case...
22532        */
22533
22534       /* Send interrupt message to cdrom task */
22535     #if XXX
22536     #if __minix_vmd
22537       interrupt(mcd_tasknr);
22538     #else
22539       interrupt(CDROM);
22540     #endif
22541     #endif
22542
22543       return 1;
22544     }


22547     /*==========================================================================*
22548      *                              mcd_play_mss                                *
22549      *==========================================================================*/
```

```
22550   PRIVATE int mcd_play_mss(mss)
22551   struct cd_play_mss mss;  /* from where to play minute:second.sector */
22552   {
22553     /* Command the drive to start playing at min:sec.sector */
22554
22555     int i;
22556
22557   #if MCD_DEBUG >= 1
22558     printf("Play_mss: begin: %02d:%02d.%02d  end: %02d:%02d.%02d\n",
22559             mss.begin_mss[MINUTES], mss.begin_mss[SECONDS],
22560             mss.begin_mss[SECTOR], mss.end_mss[MINUTES],
22561             mss.end_mss[SECONDS], mss.end_mss[SECTOR]);
22562   #endif
22563
22564     for (i=0; i < MCD_RETRIES; i++)     /* Try it more than once */
22565     {
22566       lock();        /* No interrupts when we issue this command */
22567
22568       /* Send command with paramters to drive */
22569       out_byte(MCD_DATA_PORT, MCD_READ_FROM_TO);
22570       out_byte(MCD_DATA_PORT, bin2bcd(mss.begin_mss[MINUTES]));
22571       out_byte(MCD_DATA_PORT, bin2bcd(mss.begin_mss[SECONDS]));
22572       out_byte(MCD_DATA_PORT, bin2bcd(mss.begin_mss[SECTOR]));
22573       out_byte(MCD_DATA_PORT, bin2bcd(mss.end_mss[MINUTES]));
22574       out_byte(MCD_DATA_PORT, bin2bcd(mss.end_mss[SECONDS]));
22575       out_byte(MCD_DATA_PORT, bin2bcd(mss.end_mss[SECTOR]));
22576
22577       unlock();      /* Enable interrupts again */
22578
22579       mcd_get_status(0);                    /* See if all went well */
22580       if (McdStatus & AUDIO_PLAYING) break; /* Ok, we're playing */
22581     }
22582
22583     if (i == MCD_RETRIES) return EIO;       /* Command failed */
22584
22585     /* keep in mind where we going in case of a future resume */
22586     PlayMss.end_mss[MINUTES] = mss.end_mss[MINUTES];
22587     PlayMss.end_mss[SECONDS] = mss.end_mss[SECONDS];
22588     PlayMss.end_mss[SECTOR] = mss.end_mss[SECTOR];
22589
22590     McdStatus &= ~(AUDIO_PAUSED);
22591
22592     return(OK);
22593   }


22596   /*==========================================================================*
22597    *                              mcd_play_tracks                             *
22598    *==========================================================================*/
22599   PRIVATE int mcd_play_tracks(tracks)
22600   struct cd_play_track tracks;     /* which tracks to play */
22601   {
22602     /* Command the drive to play tracks */
22603
22604     int i, err;
22605     struct cd_play_mss mss;
22606
22607   #if MCD_DEBUG >= 1
22608     printf("Play tracks: begin: %02d end: %02d\n",
22609              tracks.begin_track, tracks.end_track);
```

```
22610   #endif
22611
22612     /* First read the table of contents */
22613     if ((err = mcd_read_toc()) != OK) return err;
22614
22615     /* Check if parameters are valid */
22616     if (tracks.begin_track < DiskInfo.first_track ||
22617         tracks.end_track > DiskInfo.last_track ||
22618         tracks.begin_track > tracks.end_track)
22619       return EINVAL;
22620
22621
22622     /* Convert the track numbers to min:sec.sector */
22623     for (i=0; i<3; i++)
22624     {
22625       mss.begin_mss[i] = Toc[tracks.begin_track].position_mss[i];
22626       mss.end_mss[i] = Toc[tracks.end_track+1].position_mss[i];
22627     }
22628
22629     return(mcd_play_mss(mss));     /* Start playing */
22630   }


22633   /*==========================================================================*
22634    *                              mcd_get_disk_info                           *
22635    *==========================================================================*/
22636   PRIVATE int mcd_get_disk_info()
22637   {
22638     /* Get disk info */
22639
22640     int i, err;
22641
22642     if (McdStatus & INFO_UPTODATE) return OK; /* No need to read info again */
22643
22644     /* Issue the get volume info command */
22645     if ((err = mcd_send_command(MCD_GET_VOL_INFO)) != OK) return err;
22646
22647     /* Fill global DiskInfo */
22648     for (i=0; i < sizeof(DiskInfo); i++)
22649     {
22650       if ((err = mcd_get_reply((u8_t *)(&DiskInfo) + i, REPLY_DELAY)) !=OK)
22651         return err;
22652       bcd2bin((u8_t *)(&DiskInfo) + i);
22653     }
22654
22655   #if MCD_DEBUG >= 1
22656     printf("Mitsumi disk info: first: %d last: %d first %02d:%02d.%02d length: %02d:%02d.%02d\n",
22657         DiskInfo.first_track,
22658         DiskInfo.last_track,
22659         DiskInfo.first_track_mss[MINUTES],
22660         DiskInfo.first_track_mss[SECONDS],
22661         DiskInfo.first_track_mss[SECTOR],
22662         DiskInfo.disk_length_mss[MINUTES],
22663         DiskInfo.disk_length_mss[SECONDS],
22664         DiskInfo.disk_length_mss[SECTOR]);
22665   #endif
22666
22667     /* Update global status info */
22668     McdStatus |= INFO_UPTODATE; /* toc header has been read */
22669     McdStatus &= ~TOC_UPTODATE; /* toc has not been read yet */
```

```
22670
22671     return OK;
22672   }


22676   /*==========================================================================*
22677    *                              mcd_read_q_channel                          *
22678    *==========================================================================*/
22679   PRIVATE int mcd_read_q_channel(qc)
22680   struct cd_toc_entry *qc;  /* struct to return q-channel info in */
22681   {
22682     /* Read the qchannel info, if we we're already playing this returns
22683      * the relative position and the absolute position of where we are
22684      * in min:sec.sector. If we're not playing, this returns an entry
22685      * from the table of contents
22686      */
22687
22688     int i, err;
22689
22690     /* Issue the command */
22691     if ((err = mcd_send_command(MCD_GET_Q_CHANNEL)) != OK) return err;
22692
22693     /* Read the info */
22694     for (i=0; i < sizeof(struct cd_toc_entry); i++)
22695     {
22696       /* Return error on timeout */
22697       if ((err = mcd_get_reply((u8_t *)qc + i, REPLY_DELAY)) != OK)
22698         return err;
22699
22700       bcd2bin((u8_t *)qc + i);  /* Convert value to binary */
22701     }
22702
22703   #if MCD_DEBUG >= 2
22704     printf("qchannel info: ctl_addr: %d track: %d index: %d length %02d:%02d.%02d pos: %02d:%02d.%02d\n",
22705         qc->control_address,
22706         qc->track_nr,
22707         qc->index_nr,
22708         qc->track_time_mss[MINUTES],
22709         qc->track_time_mss[SECONDS],
22710         qc->track_time_mss[SECTOR],
22711         qc->position_mss[MINUTES],
22712         qc->position_mss[SECONDS],
22713         qc->position_mss[SECTOR]);
22714   #endif
22715
22716     return OK;  /* All done */
22717   }


22720   /*==========================================================================*
22721    *                              mcd_read_toc                                *
22722    *==========================================================================*/
22723   PRIVATE int mcd_read_toc()
22724   {
22725     /* Read the table of contents (TOC) */
22726
22727     struct cd_toc_entry q_info;
22728     int current_track, current_index;
22729     int err,i;
```

```
22730
22731
22732     if (McdStatus & TOC_UPTODATE) return OK; /* No need to read toc again */
22733
22734     /* Clear toc table */
22735     for (i = 0; i < MAX_TRACKS; i++) Toc[i].index_nr = 0;
22736
22737     /* Read disk info */
22738     if ((err = mcd_get_disk_info()) != OK) return err;
22739
22740     /* Calculate track to start with */
22741     current_track = DiskInfo.last_track - DiskInfo.first_track + 1;
22742
22743     /* Set read toc mode */
22744     if ((err = mcd_set_mode(MCD_TOC)) != OK) return err;
22745
22746     /* Read the complete TOC, on every read-q-channel command we get a random
22747      * TOC entry depending on how far we are in the q-channel, collect entries
22748      * as long as we don't have the complete TOC. There's a limit of 600 here,
22749      * if we don't have the complete TOC after 600 reads we quit with an error
22750      */
22751     for (i = 0; (i < 600 && current_track > 0); i++)
22752     {
22753       /* Try to read a TOC entry */
22754       if ((err = mcd_read_q_channel(&q_info)) != OK) break;
22755
22756       /* Is this a valid track number and didn't we have it yet ? */
22757       current_index = q_info.index_nr;
22758       if (current_index >= DiskInfo.first_track &&
22759           current_index <= DiskInfo.last_track &&
22760           q_info.track_nr == 0)
22761       {
22762         /* Copy entry into toc table */
22763         if (Toc[current_index].index_nr == 0)
22764         {
22765           Toc[current_index].control_address = q_info.control_address;
22766           Toc[current_index].track_nr = current_index;
22767           Toc[current_index].index_nr = 1;
22768           Toc[current_index].track_time_mss[MINUTES] = q_info.track_time_mss[MINUTES];
22769           Toc[current_index].track_time_mss[SECONDS] = q_info.track_time_mss[SECONDS];
22770           Toc[current_index].track_time_mss[SECTOR] = q_info.track_time_mss[SECTOR];
22771           Toc[current_index].position_mss[MINUTES] = q_info.position_mss[MINUTES];
22772           Toc[current_index].position_mss[SECONDS] = q_info.position_mss[SECONDS];
22773           Toc[current_index].position_mss[SECTOR] = q_info.position_mss[SECTOR];
22774           current_track--;
22775         }
22776       }
22777     }
22778     if (err) return err;   /* Do we have all toc entries? */
22779
22780     /* Fill in lead out */
22781     current_index = DiskInfo.last_track + 1;
22782     Toc[current_index].control_address =
22783                                     Toc[current_index-1].control_address;
22784     Toc[current_index].track_nr = 0;
22785     Toc[current_index].index_nr = LEAD_OUT;
22786     Toc[current_index].position_mss[MINUTES] = DiskInfo.disk_length_mss[MINUTES];
22787     Toc[current_index].position_mss[SECONDS] = DiskInfo.disk_length_mss[SECONDS];
22788     Toc[current_index].position_mss[SECTOR] = DiskInfo.disk_length_mss[SECTOR];
22789
```

```
22790     /* Return to cooked mode */
22791     if ((err = mcd_set_mode(MCD_COOKED)) != OK) return err;
22792
22793     /* Update global status */
22794     McdStatus |= TOC_UPTODATE;
22795
22796   #if MCD_DEBUG >= 1
22797     for (i = DiskInfo.first_track; i <= current_index; i++)
22798     {
22799       printf("Mitsumi toc %d: trk %d  index %d  time %02d:%02d.%02d  pos: %02d:%02d.%02d\n",
22800           i,
22801           Toc[i].track_nr,
22802           Toc[i].index_nr,
22803           Toc[i].track_time_mss[MINUTES],
22804           Toc[i].track_time_mss[SECONDS],
22805           Toc[i].track_time_mss[SECTOR],
22806           Toc[i].position_mss[MINUTES],
22807           Toc[i].position_mss[SECONDS],
22808           Toc[i].position_mss[SECTOR]);
22809     }
22810   #endif
22811
22812     return OK;
22813   }


22816   /*==========================================================================*
22817    *                              mcd_stop                                    *
22818    *==========================================================================*/
22819   PRIVATE int mcd_stop()
22820   {
22821     int err;
22822
22823     if ((err = mcd_send_command(MCD_STOP)) != OK ) return err;
22824
22825     McdStatus &= ~(AUDIO_PAUSED);
22826
22827     return OK;
22828   }
22829
22830
22831   /*==========================================================================*
22832    *                              mcd_eject                                   *
22833    *==========================================================================*/
22834   PRIVATE int mcd_eject()
22835   {
22836     int err;
22837
22838     if ((err = mcd_send_command(MCD_EJECT)) != OK) return err;
22839     return OK;
22840   }
22841
22842
22843   /*==========================================================================*
22844    *                              mcd_pause                                   *
22845    *==========================================================================*/
22846   PRIVATE int mcd_pause()
22847   {
22848     int err;
22849     struct cd_toc_entry qc;
```

```
22850
22851     /* We can only pause when we are playing audio */
22852     if (!(McdStatus & AUDIO_PLAYING)) return EINVAL;
22853
22854     /* Look where we are */
22855     if ((err = mcd_read_q_channel(&qc)) != OK) return err;
22856
22857     /* Stop playing */
22858     if ((err = mcd_send_command(MCD_STOP_AUDIO)) != OK) return err;
22859
22860     /* Keep in mind were we have to start again */
22861     PlayMss.begin_mss[MINUTES] = qc.position_mss[MINUTES];
22862     PlayMss.begin_mss[SECONDS] = qc.position_mss[SECONDS];
22863     PlayMss.begin_mss[SECTOR] = qc.position_mss[SECTOR];
22864
22865     /* Update McdStatus */
22866     McdStatus |= AUDIO_PAUSED;
22867
22868   #if MCD_DEBUG >= 1
22869     printf("Mcd_paused at: %02d:%02d.%02d\n",
22870         PlayMss.begin_mss[MINUTES],
22871         PlayMss.begin_mss[SECONDS],
22872         PlayMss.begin_mss[SECTOR]);
22873   #endif
22874
22875     return OK;
22876   }
22877
22878
22879   /*==========================================================================*
22880    *                              mcd_resume                                  *
22881    *==========================================================================*/
22882   PRIVATE int mcd_resume()
22883   {
22884     int err;
22885
22886     /* we can only resume if we are in a pause state */
22887     if (!(McdStatus & AUDIO_PAUSED)) return EINVAL;
22888
22889     /* start playing where we left off */
22890     if ((err = mcd_play_mss(PlayMss)) != OK) return err;
22891
22892     McdStatus &= ~(AUDIO_PAUSED);
22893
22894   #if MCD_DEBUG >= 1
22895     printf("Mcd resumed at: %02d:%02d.%02d\n",
22896         PlayMss.begin_mss[MINUTES],
22897         PlayMss.begin_mss[SECONDS],
22898         PlayMss.begin_mss[SECTOR]);
22899   #endif
22900
22901     return OK;
22902   }
22903
22904
22905   /*==========================================================================*
22906    *                              ioctl_read_sub                              *
22907    *==========================================================================*/
22908   PRIVATE int ioctl_read_sub(m_ptr)
22909   message *m_ptr;
```

```
22910   {
22911     phys_bytes user_phys;
22912     struct cd_toc_entry sub;
22913     int err;
22914
22915     /* We can only read a sub channel when we are playing audio */
22916     if (!(McdStatus & AUDIO_PLAYING)) return EINVAL;
22917
22918     /* Read the sub channel */
22919     if ((err = mcd_read_q_channel(&sub)) != OK) return err;
22920
22921     /* Copy info to user */
22922     user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, sizeof(sub));
22923     if (user_phys == 0) return(EFAULT);
22924     phys_copy(vir2phys(&sub), user_phys, (phys_bytes) sizeof(sub));
22925
22926     return OK;
22927   }
22928
22929
22930
22931   /*==========================================================================*
22932    *                              ioctl_read_toc                              *
22933    *==========================================================================*/
22934   PRIVATE int ioctl_read_toc(m_ptr)
22935   message *m_ptr;
22936   {
22937     phys_bytes user_phys;
22938     int err, toc_size;
22939
22940     /* Try to read the table of contents */
22941     if ((err = mcd_read_toc()) != OK) return err;
22942
22943     /* Get size of toc */
22944     toc_size = (DiskInfo.last_track + 1) * sizeof(struct cd_toc_entry);
22945
22946     /* Copy to user */
22947     user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, toc_size);
22948     if (user_phys == 0) return(EFAULT);
22949     phys_copy(vir2phys(&Toc), user_phys, (phys_bytes) toc_size);
22950
22951     return OK;
22952   }
22953
22954
22955   /*==========================================================================*
22956    *                              ioctl_disk_info                             *
22957    *==========================================================================*/
22958   PRIVATE int ioctl_disk_info(m_ptr)
22959   message *m_ptr;
22960   {
22961     phys_bytes user_phys;
22962     int err;
22963
22964     /* Try to read the toc header */
22965     if ((err = mcd_get_disk_info()) != OK) return err;
22966
22967     /* Copy info to user */
22968     user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, sizeof(DiskInfo));
22969     if (user_phys == 0) return(EFAULT);
```

```
22970     phys_copy(vir2phys(&DiskInfo), user_phys, (phys_bytes) sizeof(DiskInfo));
22971
22972     return OK;
22973   }


22976   /*==========================================================================*
22977    *                              ioctl_play_mss                              *
22978    *==========================================================================*/
22979   PRIVATE int ioctl_play_mss(m_ptr)
22980   message *m_ptr;
22981   {
22982     phys_bytes user_phys;
22983     struct cd_play_mss mss;
22984
22985     /* Get user data */
22986     user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, sizeof(mss));
22987     if (user_phys == 0) return(EFAULT);
22988     phys_copy(user_phys, vir2phys(&mss), (phys_bytes) sizeof(mss));
22989
22990     /* Try to play */
22991     return mcd_play_mss(mss);
22992   }


22995   /*==========================================================================*
22996    *                              ioctl_play_ti                               *
22997    *==========================================================================*/
22998   PRIVATE int ioctl_play_ti(m_ptr)
22999   message *m_ptr;
23000   {
23001     phys_bytes user_phys;
23002     struct cd_play_track tracks;
23003
23004     /* Get user data */
23005     user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, sizeof(tracks));
23006     if (user_phys == 0) return(EFAULT);
23007     phys_copy(user_phys, vir2phys(&tracks), (phys_bytes) sizeof(tracks));
23008
23009     /* Try to play */
23010     return mcd_play_tracks(tracks);
23011   }


23014   /*==========================================================================*
23015    *                              mcd_prepare                                 *
23016    *==========================================================================*/
23017   PRIVATE struct device *mcd_prepare(device)
23018   int device;
23019   {
23020     /* Nothing to transfer as yet. */
23021     mcd_count = 0;
23022
23023     /* Select partition. */
23024     if (device < DEV_PER_DRIVE) {                /* cd0, cd1, ... */
23025       mcd_dv = &mcd_part[device];
23026     } else
23027     if ((unsigned) (device -= MINOR_hd1a) < SUB_PER_DRIVE) { /* cd1a, cd1b, ... */
23028       mcd_dv = &mcd_subpart[device];
23029     } else {
```

```
23030         return NIL_DEV;
23031       }
23032
23033       return mcd_dv;
23034     }


23037     /*===========================================================================*
23038      *                              mcd_schedule                                 *
23039      *===========================================================================*/
23040     PRIVATE int mcd_schedule(proc_nr, iop)
23041     int proc_nr;                    /* process doing the request */
23042     struct iorequest_s *iop;        /* pointer to read or write request */
23043     {
23044     /* Gather I/O requests on consecutive blocks so they may be read/written
23045      * in one controller command.  (There is enough time to compute the next
23046      * consecutive request while an unwanted block passes by.)
23047      */
23048       int r, opcode;
23049       unsigned long pos;
23050       unsigned nbytes;
23051       phys_bytes user_phys;
23052
23053       /* This many bytes to read */
23054       nbytes = iop->io_nbytes;
23055
23056       /* From/to this position on the device */
23057       pos = iop->io_position;
23058
23059       /* To/from this user address */
23060       user_phys = numap(proc_nr, (vir_bytes) iop->io_buf, nbytes);
23061       if (user_phys == 0) return(iop->io_nbytes = EINVAL);
23062
23063       /* Read or write? */
23064       opcode = iop->io_request & ~OPTIONAL_IO;
23065
23066       /* Only read permitted on cdrom */
23067       if (opcode != DEV_READ) return EIO;
23068
23069       /* What position on disk and how close to EOF? */
23070       if (pos >= mcd_dv->dv_size) return(OK);       /* At EOF */
23071       if (pos + nbytes > mcd_dv->dv_size) nbytes = mcd_dv->dv_size - pos;
23072       pos += mcd_dv->dv_base;
23073
23074       if (mcd_count > 0 && pos != mcd_nextpos) {
23075             /* This new request can't be chained to the job being built */
23076             if ((r = mcd_finish()) != OK) return(r);
23077       }
23078
23079       /* Next consecutive position. */
23080       mcd_nextpos = pos + nbytes;
23081
23082       if (mcd_count == 0)
23083       {
23084         /* The first request in a row, initialize. */
23085         mcd_tp = mcd_trans;
23086       }
23087
23088       /* Store I/O parameters */
23089       mcd_tp->tr_iop = iop;
```

```
23090       mcd_tp->tr_pos = pos;
23091       mcd_tp->tr_count = nbytes;
23092       mcd_tp->tr_phys = user_phys;
23093
23094       /* Update counters */
23095       mcd_tp++;
23096       mcd_count += nbytes;
23097       return(OK);
23098     }


23101     /*===========================================================================*
23102      *                              mcd_finish                                   *
23103      *===========================================================================*/
23104     PRIVATE int mcd_finish()
23105     {
23106     /* Carry out the I/O requests gathered in mcd_trans[]. */
23107
23108       struct trans *tp = mcd_trans;
23109       int err, errors;
23110       u8_t mss[3];
23111       unsigned long pos;
23112       unsigned count, n;
23113
23114       if (mcd_count == 0) return(OK);       /* we're already done */
23115
23116       /* Update status */
23117       mcd_get_status(1);
23118       if (McdStatus & (AUDIO_DISK | NO_DISK))
23119         return(tp->tr_iop->io_nbytes = EIO);
23120
23121       /* Set cooked mode */
23122       if ((err = mcd_set_mode(MCD_COOKED)) != OK)
23123         return(tp->tr_iop->io_nbytes = err);
23124
23125       while (mcd_count > 0)
23126       {
23127         /* Position on the CD rounded down to the CD block size */
23128         pos = tp->tr_pos & ~MCD_BLOCK_MASK;
23129
23130         /* Byte count rounded up. */
23131         count = (pos - tp->tr_pos) + mcd_count;
23132         count = (count + MCD_BLOCK_SIZE - 1) & ~MCD_BLOCK_MASK;
23133
23134         /* XXX transfer size limits? */
23135         if (count > MCD_BLOCK_SIZE) count = MCD_BLOCK_SIZE;
23136
23137         /* Compute disk position in min:sec:sector */
23138         block2mss(pos >> MCD_BLOCK_SHIFT, mss);
23139
23140         /* Now try to read a block */
23141         errors = 0;
23142         while (errors < MCD_RETRIES)
23143         {
23144           lock();
23145           out_byte(MCD_DATA_PORT, MCD_READ_FROM_TO);
23146           out_byte(MCD_DATA_PORT, bin2bcd(mss[MINUTES]));
23147           out_byte(MCD_DATA_PORT, bin2bcd(mss[SECONDS]));
23148           out_byte(MCD_DATA_PORT, bin2bcd(mss[SECTOR]));
23149           out_byte(MCD_DATA_PORT, 0);
```

```
23150         out_byte(MCD_DATA_PORT, 0);
23151         out_byte(MCD_DATA_PORT, 1);       /* XXX count in mss form? */
23152         unlock();
23153
23154         /* Wait for data */
23155         if (mcd_data_ready(REPLY_DELAY) == OK) break;
23156         printf("Mcd: data time out\n");
23157         errors++;
23158       }
23159       if (errors == MCD_RETRIES) return(tp->tr_iop->io_nbytes = EIO);
23160
23161       /* Prepare reading data. */
23162       out_byte(MCD_CONTROL_PORT, 0x04);
23163
23164       while (pos < tp->tr_pos)
23165       {
23166         /* Discard bytes before the position we are really interested in. */
23167         n = tp->tr_pos - pos;
23168         if (n > DMA_BUF_SIZE) n = DMA_BUF_SIZE;
23169         port_read_byte(MCD_DATA_PORT, tmp_phys, n);
23170   #if XXX
23171   printf("count = %u, n = %u, tr_pos = %lu, io_nbytes = %u, tr_count = %u, mcd_count = %u\n",
23172   count, n, 0, 0, 0, mcd_count);
23173   #endif
23174         pos += n;
23175         count -= n;
23176       }
23177
23178       while (mcd_count > 0 && count > 0)
23179       {
23180         /* Transfer bytes into the user buffers. */
23181         n = tp->tr_count;
23182         if (n > count) n = count;
23183         port_read_byte(MCD_DATA_PORT, tp->tr_phys, n);
23184   #if XXX
23185   printf("count = %u, n = %u, tr_pos = %lu, io_nbytes = %u, tr_count = %u, mcd_count = %u\n",
23186   count, n, tp->tr_pos, tp->tr_iop->io_nbytes, tp->tr_count, mcd_count);
23187   #endif
23188         tp->tr_phys += n;
23189         tp->tr_pos += n;
23190         tp->tr_iop->io_nbytes -= n;
23191         if ((tp->tr_count -= n) == 0) tp++;
23192         count -= n;
23193         mcd_count -= n;
23194       }
23195
23196       while (count > 0)
23197       {
23198         /* Discard excess bytes. */
23199         n = count;
23200         if (n > DMA_BUF_SIZE) n = DMA_BUF_SIZE;
23201         port_read_byte(MCD_DATA_PORT, tmp_phys, n);
23202   #if XXX
23203   printf("count = %u, n = %u, tr_pos = %lu, io_nbytes = %u, tr_count = %u, mcd_count = %u\n",
23204   count, n, 0, 0, 0, mcd_count);
23205   #endif
23206         count -= n;
23207       }
23208
23209       /* Finish reading data. */
```

```
23210         out_byte(MCD_CONTROL_PORT, 0x0c);
23211     #if 0 /*XXX*/
23212         mcd_get_status(1);
23213         if (!(McdStatus & DISK_ERROR)) done = 1; /* OK, no errors */
23214     #endif
23215       }
23216
23217       return OK;
23218     }


23221     /*==========================================================================*
23222      *                                mcd_geometry                              *
23223      *==========================================================================*/
23224     PRIVATE void mcd_geometry(entry)
23225     struct partition *entry;
23226     {
23227     /* The geometry of a cdrom doesn't look like the geometry of a regular disk,
23228      * so we invent a geometry to keep external programs happy.
23229      */
23230       entry->cylinders = (mcd_part[0].dv_size >> SECTOR_SHIFT) / (64 * 32);
23231       entry->heads = 64;
23232       entry->sectors = 32;
23233     }


23236     /*==========================================================================*
23237      *                                misc functions                            *
23238      *==========================================================================*/
23239     PRIVATE u8_t bin2bcd(u8_t b)
23240     {
23241       /* Convert a number to binary-coded-decimal */
23242       int u,t;
23243
23244       u = b%10;
23245       t = b/10;
23246       return (u8_t)(u | (t << 4));
23247     }


23250     PRIVATE void bcd2bin(u8_t *bcd)
23251     {
23252       /* Convert binary-coded-decimal to binary :-) */
23253
23254       *bcd = (*bcd >> 4) * 10 + (*bcd & 0xf);
23255     }


23258     PRIVATE void block2mss(block, mss)
23259     long block;
23260     u8_t *mss;
23261     {
23262       /* Compute disk position of a block in min:sec:sector */
23263
23264       block += MCD_SKIP;
23265       mss[MINUTES] = block/(SECONDS_PER_MINUTE * SECTORS_PER_SECOND);
23266       block %= (SECONDS_PER_MINUTE * SECTORS_PER_SECOND);
23267       mss[SECONDS] = block/(SECTORS_PER_SECOND);
23268       mss[SECTOR] = block%(SECTORS_PER_SECOND);
23269     }
```

```
23272    PRIVATE long mss2block(u8_t *mss)
23273    {
23274      /* Compute block number belonging to
23275       * disk position min:sec:sector
23276       */
23277
23278      return ((((unsigned long) mss[MINUTES] * SECONDS_PER_MINUTE
23279            + (unsigned long) mss[SECONDS]) * SECTORS_PER_SECOND)
23280            + (unsigned long) mss[SECTOR]) - MCD_SKIP;
23281    }
23282    #endif /* ENABLE_MITSUMI_CDROM */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/memory.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
23300    /* This file contains the device dependent part of the drivers for the
23301     * following special files:
23302     *     /dev/null        - null device (data sink)
23303     *     /dev/mem         - absolute memory
23304     *     /dev/kmem        - kernel virtual memory
23305     *     /dev/ram         - RAM disk
23306     *
23307     * The file contains one entry point:
23308     *
23309     *   mem_task:  main entry when system is brought up
23310     *
23311     *  Changes:
23312     *      20 Apr  1992 by Kees J. Bot: device dependent/independent split
23313     */
23314
23315    #include "kernel.h"
23316    #include "driver.h"
23317    #include <sys/ioctl.h>
23318
23319    #define NR_RAMS            4    /* number of RAM-type devices */
23320
23321    PRIVATE struct device m_geom[NR_RAMS];  /* Base and size of each RAM disk */
23322    PRIVATE int m_device;           /* current device */
23323
23324    FORWARD _PROTOTYPE( struct device *m_prepare, (int device) );
23325    FORWARD _PROTOTYPE( int m_schedule, (int proc_nr, struct iorequest_s *iop) );
23326    FORWARD _PROTOTYPE( int m_do_open, (struct driver *dp, message *m_ptr) );
23327    FORWARD _PROTOTYPE( void m_init, (void) );
23328    FORWARD _PROTOTYPE( int m_ioctl, (struct driver *dp, message *m_ptr) );
23329    FORWARD _PROTOTYPE( void m_geometry, (struct partition *entry) );
23330
23331
23332    /* Entry points to this driver. */
23333    PRIVATE struct driver m_dtab = {
23334      no_name,      /* current device's name */
23335      m_do_open,    /* open or mount */
23336      do_nop,       /* nothing on a close */
23337      m_ioctl,      /* specify ram disk geometry */
23338      m_prepare,    /* prepare for I/O on a given minor device */
23339      m_schedule,   /* do the I/O */
```

```
23340       nop_finish,   /* schedule does the work, no need to be smart */
23341       nop_cleanup,  /* nothing's dirty */
23342       m_geometry,   /* memory device "geometry" */
23343     };
23344
23345
23346     /*===========================================================================*
23347      *                              mem_task                                     *
23348      *===========================================================================*/
23349     PUBLIC void mem_task()
23350     {
23351       m_init();
23352       driver_task(&m_dtab);
23353     }


23356     /*===========================================================================*
23357      *                              m_prepare                                    *
23358      *===========================================================================*/
23359     PRIVATE struct device *m_prepare(device)
23360     int device;
23361     {
23362     /* Prepare for I/O on a device. */
23363
23364       if (device < 0 || device >= NR_RAMS) return(NIL_DEV);
23365       m_device = device;
23366
23367       return(&m_geom[device]);
23368     }


23371     /*===========================================================================*
23372      *                              m_schedule                                   *
23373      *===========================================================================*/
23374     PRIVATE int m_schedule(proc_nr, iop)
23375     int proc_nr;                    /* process doing the request */
23376     struct iorequest_s *iop;        /* pointer to read or write request */
23377     {
23378     /* Read or write /dev/null, /dev/mem, /dev/kmem, or /dev/ram. */
23379
23380       int device, count, opcode;
23381       phys_bytes mem_phys, user_phys;
23382       struct device *dv;
23383
23384       /* Type of request */
23385       opcode = iop->io_request & ~OPTIONAL_IO;
23386
23387       /* Get minor device number and check for /dev/null. */
23388       device = m_device;
23389       dv = &m_geom[device];
23390
23391       /* Determine address where data is to go or to come from. */
23392       user_phys = numap(proc_nr, (vir_bytes) iop->io_buf,
23393                                                 (vir_bytes) iop->io_nbytes);
23394       if (user_phys == 0) return(iop->io_nbytes = EINVAL);
23395
23396       if (device == NULL_DEV) {
23397             /* /dev/null: Black hole. */
23398             if (opcode == DEV_WRITE) iop->io_nbytes = 0;
23399             count = 0;
```

```
23400     } else {
23401           /* /dev/mem, /dev/kmem, or /dev/ram: Check for EOF */
23402           if (iop->io_position >= dv->dv_size) return(OK);
23403           count = iop->io_nbytes;
23404           if (iop->io_position + count > dv->dv_size)
23405                   count = dv->dv_size - iop->io_position;
23406     }
23407
23408     /* Set up 'mem_phys' for /dev/mem, /dev/kmem, or /dev/ram */
23409     mem_phys = dv->dv_base + iop->io_position;
23410
23411     /* Book the number of bytes to be transferred in advance. */
23412     iop->io_nbytes -= count;
23413
23414     if (count == 0) return(OK);
23415
23416     /* Copy the data. */
23417     if (opcode == DEV_READ)
23418           phys_copy(mem_phys, user_phys, (phys_bytes) count);
23419     else
23420           phys_copy(user_phys, mem_phys, (phys_bytes) count);
23421
23422     return(OK);
23423   }


23426   /*===========================================================================*
23427    *                              m_do_open                                    *
23428    *===========================================================================*/
23429   PRIVATE int m_do_open(dp, m_ptr)
23430   struct driver *dp;
23431   message *m_ptr;
23432   {
23433   /* Check device number on open.  Give I/O privileges to a process opening
23434    * /dev/mem or /dev/kmem.
23435    */
23436
23437     if (m_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
23438
23439   #if (CHIP == INTEL)
23440     if (m_device == MEM_DEV || m_device == KMEM_DEV)
23441           enable_iop(proc_addr(m_ptr->PROC_NR));
23442   #endif
23443
23444     return(OK);
23445   }


23448   /*===========================================================================*
23449    *                              m_init                                       *
23450    *===========================================================================*/
23451   PRIVATE void m_init()
23452   {
23453     /* Initialize this task. */
23454     extern int _end;
23455
23456     m_geom[KMEM_DEV].dv_base = vir2phys(0);
23457     m_geom[KMEM_DEV].dv_size = vir2phys(&_end);
23458
23459   #if (CHIP == INTEL)
```

```
23460     if (!protected_mode) {
23461           m_geom[MEM_DEV].dv_size =   0x100000;   /* 1M for 8086 systems */
23462     } else {
23463   #if _WORD_SIZE == 2
23464           m_geom[MEM_DEV].dv_size =  0x1000000;   /* 16M for 286 systems */
23465   #else
23466           m_geom[MEM_DEV].dv_size = 0xFFFFFFFF;   /* 4G-1 for 386 systems */
23467   #endif
23468     }
23469   #else /* !(CHIP == INTEL) */
23470   #if (CHIP == M68000)
23471     m_geom[MEM_DEV].dv_size = MEM_BYTES;
23472   #else /* !(CHIP == M68000) */
23473   #error /* memory limit not set up */
23474   #endif /* !(CHIP == M68000) */
23475   #endif /* !(CHIP == INTEL) */
23476   }


23479   /*===========================================================================*
23480    *                              m_ioctl                                      *
23481    *===========================================================================*/
23482   PRIVATE int m_ioctl(dp, m_ptr)
23483   struct driver *dp;
23484   message *m_ptr;                 /* pointer to read or write message */
23485   {
23486   /* Set parameters for one of the RAM disks. */
23487
23488     unsigned long bytesize;
23489     unsigned base, size;
23490     struct memory *memp;
23491     static struct psinfo psinfo = { NR_TASKS, NR_PROCS, (vir_bytes) proc, 0, 0 };
23492     phys_bytes psinfo_phys;
23493
23494     switch (m_ptr->REQUEST) {
23495     case MIOCRAMSIZE:
23496           /* FS sets the RAM disk size. */
23497           if (m_ptr->PROC_NR != FS_PROC_NR) return(EPERM);
23498
23499           bytesize = m_ptr->POSITION * BLOCK_SIZE;
23500           size = (bytesize + CLICK_SHIFT-1) >> CLICK_SHIFT;
23501
23502           /* Find a memory chunk big enough for the RAM disk. */
23503           memp= &mem[NR_MEMS];
23504           while ((--memp)->size < size) {
23505                   if (memp == mem) panic("RAM disk is too big", NO_NUM);
23506           }
23507           base = memp->base;
23508           memp->base += size;
23509           memp->size -= size;
23510
23511           m_geom[RAM_DEV].dv_base = (unsigned long) base << CLICK_SHIFT;
23512           m_geom[RAM_DEV].dv_size = bytesize;
23513           break;
23514     case MIOCSPSINFO:
23515           /* MM or FS set the address of their process table. */
23516           if (m_ptr->PROC_NR == MM_PROC_NR) {
23517                   psinfo.mproc = (vir_bytes) m_ptr->ADDRESS;
23518           } else
23519           if (m_ptr->PROC_NR == FS_PROC_NR) {
```

```
23520                   psinfo.fproc = (vir_bytes) m_ptr->ADDRESS;
23521           } else {
23522                   return(EPERM);
23523           }
23524           break;
23525     case MIOCGPSINFO:
23526           /* The ps program wants the process table addresses. */
23527           psinfo_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
23528                                                           sizeof(psinfo));
23529           if (psinfo_phys == 0) return(EFAULT);
23530           phys_copy(vir2phys(&psinfo), psinfo_phys, (phys_bytes) sizeof(psinfo));
23531           break;
23532     default:
23533           return(do_diocntl(&m_dtab, m_ptr));
23534     }
23535     return(OK);
23536   }


23539   /*===========================================================================*
23540    *                              m_geometry                                   *
23541    *===========================================================================*/
23542   PRIVATE void m_geometry(entry)
23543   struct partition *entry;
23544   {
23545     /* Memory devices don't have a geometry, but the outside world insists. */
23546     entry->cylinders = (m_geom[m_device].dv_size >> SECTOR_SHIFT) / (64 * 32);
23547     entry->heads = 64;
23548     entry->sectors = 32;
23549   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/misc.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

23600   /* This file contains a collection of miscellaneous procedures:
23601    *      mem_init:       initialize memory tables.  Some memory is reported
23602    *                      by the BIOS, some is guesstimated and checked later
23603    *      env_parse       parse environment variable.
23604    *      bad_assertion   for debugging
23605    *      bad_compare     for debugging
23606    */
23607
23608   #include "kernel.h"
23609   #include "assert.h"
23610   #include <stdlib.h>
23611   #include <minix/com.h>
23612
23613   #if (CHIP == INTEL)
23614
23615   #define EM_BASE     0x100000L   /* base of extended memory on AT's */
23616   #define SHADOW_BASE 0xFA0000L   /* base of RAM shadowing ROM on some AT's */
23617   #define SHADOW_MAX  0x060000L   /* maximum usable shadow memory (16M limit) */
23618
23619   /*===========================================================================*
```

```
23620     *                             mem_init                                  *
23621     *=======================================================================*/
23622    PUBLIC void mem_init()
23623    {
23624    /* Initialize the memory size tables.  This is complicated by fragmentation
23625     * and different access strategies for protected mode.  There must be a
23626     * chunk at 0 big enough to hold Minix proper.  For 286 and 386 processors,
23627     * there can be extended memory (memory above 1MB).  This usually starts at
23628     * 1MB, but there may be another chunk just below 16MB, reserved under DOS
23629     * for shadowing ROM, but available to Minix if the hardware can be re-mapped.
23630     * In protected mode, extended memory is accessible assuming CLICK_SIZE is
23631     * large enough, and is treated as ordinary memory.
23632     */
23633
23634      u32_t ext_clicks;
23635      phys_clicks max_clicks;
23636
23637      /* Get the size of ordinary memory from the BIOS. */
23638      mem[0].size = k_to_click(low_memsize);        /* base = 0 */
23639
23640      if (pc_at && protected_mode) {
23641            /* Get the size of extended memory from the BIOS.  This is special
23642             * except in protected mode, but protected mode is now normal.
23643             * Note that no more than 16M can be addressed in 286 mode, so make
23644             * sure that the highest memory address fits in a short when counted
23645             * in clicks.
23646             */
23647            ext_clicks = k_to_click((u32_t) ext_memsize);
23648            max_clicks = USHRT_MAX - (EM_BASE >> CLICK_SHIFT);
23649            mem[1].size = MIN(ext_clicks, max_clicks);
23650            mem[1].base = EM_BASE >> CLICK_SHIFT;
23651
23652            if (ext_memsize <= (unsigned) ((SHADOW_BASE - EM_BASE) / 1024)
23653                            && check_mem(SHADOW_BASE, SHADOW_MAX) == SHADOW_MAX) {
23654                    /* Shadow ROM memory. */
23655                    mem[2].size = SHADOW_MAX >> CLICK_SHIFT;
23656                    mem[2].base = SHADOW_BASE >> CLICK_SHIFT;
23657            }
23658      }
23659
23660      /* Total system memory. */
23661      tot_mem_size = mem[0].size + mem[1].size + mem[2].size;
23662    }
23663    #endif /* (CHIP == INTEL) */
23664
23665    /*=======================================================================*
23666     *                             env_parse                                 *
23667     *=======================================================================*/
23668    PUBLIC int env_parse(env, fmt, field, param, min, max)
23669    char *env;              /* environment variable to inspect */
23670    char *fmt;              /* template to parse it with */
23671    int field;              /* field number of value to return */
23672    long *param;            /* address of parameter to get */
23673    long min, max;          /* minimum and maximum values for the parameter */
23674    {
23675    /* Parse an environment variable setting, something like "DPETH0=300:3".
23676     * Panic if the parsing fails.  Return EP_UNSET if the environment variable
23677     * is not set, EP_OFF if it is set to "off", EP_ON if set to "on" or a
23678     * field is left blank, or EP_SET if a field is given (return value through
23679     * *param).  Commas and colons may be used in the environment and format
```

```
23680     * string, fields in the environment string may be empty, and punctuation
23681     * may be missing to skip fields.  The format string contains characters
23682     * 'd', 'o', 'x' and 'c' to indicate that 10, 8, 16, or 0 is used as the
23683     * last argument to strtol.
23684     */
23685
23686      char *val, *end;
23687      long newpar;
23688      int i = 0, radix, r;
23689
23690      if ((val = k_getenv(env)) == NIL_PTR) return(EP_UNSET);
23691      if (strcmp(val, "off") == 0) return(EP_OFF);
23692      if (strcmp(val, "on") == 0) return(EP_ON);
23693
23694      r = EP_ON;
23695      for (;;) {
23696            while (*val == ' ') val++;
23697
23698            if (*val == 0) return(r);       /* the proper exit point */
23699
23700            if (*fmt == 0) break;           /* too many values */
23701
23702            if (*val == ',' || *val == ':') {
23703                    /* Time to go to the next field. */
23704                    if (*fmt == ',' || *fmt == ':') i++;
23705                    if (*fmt++ == *val) val++;
23706            } else {
23707                    /* Environment contains a value, get it. */
23708                    switch (*fmt) {
23709                    case 'd':       radix =   10;   break;
23710                    case 'o':       radix =  010;   break;
23711                    case 'x':       radix = 0x10;   break;
23712                    case 'c':       radix =    0;   break;
23713                    default:        goto badenv;
23714                    }
23715                    newpar = strtol(val, &end, radix);
23716
23717                    if (end == val) break;  /* not a number */
23718                    val = end;
23719
23720                    if (i == field) {
23721                            /* The field requested. */
23722                            if (newpar < min || newpar > max) break;
23723                            *param = newpar;
23724                            r = EP_SET;
23725                    }
23726            }
23727      }
23728    badenv:
23729      printf("Bad environment setting: '%s = %s'\n", env, k_getenv(env));
23730      panic("", NO_NUM);
23731      /*NOTREACHED*/
23732    }

23734    #if DEBUG
23735    /*===========================================================================*
23736     *                              bad_assertion                                *
23737     *===========================================================================*/
23738    PUBLIC void bad_assertion(file, line, what)
23739    char *file;
```

```
23740    int line;
23741    char *what;
23742    {
23743      printf("panic at %s(%d): assertion \"%s\" failed\n", file, line, what);
23744      panic(NULL, NO_NUM);
23745    }

23747    /*===========================================================================*
23748     *                                bad_compare                                *
23749     *===========================================================================*/
23750    PUBLIC void bad_compare(file, line, lhs, what, rhs)
23751    char *file;
23752    int line;
23753    int lhs;
23754    char *what;
23755    int rhs;
23756    {
23757      printf("panic at %s(%d): compare (%d) %s (%d) failed\n",
23758            file, line, lhs, what, rhs);
23759      panic(NULL, NO_NUM);
23760    }
23761    #endif /* DEBUG */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/ne2000.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

23800    /*
23801    ne2000.c
23802
23803    Driver for the ne2000 ethernet cards. This file contains only the ne2000
23804    specific code, the rest is in dp8390.c
23805
23806    Created:        March 15, 1994 by Philip Homburg <philip@cs.vu.nl>
23807    */
23808
23809    #include "kernel.h"
23810    #include <net/gen/ether.h>
23811    #include <net/gen/eth_io.h>
23812    #include "dp8390.h"
23813    #include "ne2000.h"
23814
23815    #if ENABLE_NETWORKING
23816
23817    #if !__minix_vmd
23818    #define debug           0
23819    #endif
23820
23821    #define N 100
23822
23823    _PROTOTYPE( typedef int (*testf_t), (dpeth_t *dep, int pos, u8_t *pat)  );
23824
23825    u8_t    pat0[]= { 0x00, 0x00, 0x00, 0x00 };
23826    u8_t    pat1[]= { 0xFF, 0xFF, 0xFF, 0xFF };
23827    u8_t    pat2[]= { 0xA5, 0x5A, 0x69, 0x96 };
23828    u8_t    pat3[]= { 0x96, 0x69, 0x5A, 0xA5 };
23829
```

```
23830   _PROTOTYPE( static int test_8, (dpeth_t *dep, int pos, u8_t *pat)       );
23831   _PROTOTYPE( static int test_16, (dpeth_t *dep, int pos, u8_t *pat)      );
23832   _PROTOTYPE( static void ne_init, (dpeth_t *dep)                         );
23833   _PROTOTYPE( static void ne_stop, (dpeth_t *dep)                         );
23834
23835   /*============================================================================*
23836    *                              ne_probe                                      *
23837    *============================================================================*/
23838   int ne_probe(dep)
23839   dpeth_t *dep;
23840   {
23841           int byte;
23842           int i;
23843           int loc1, loc2;
23844           testf_t f;
23845
23846           dep->de_dp8390_port= dep->de_base_port + NE_DP8390;
23847
23848           /* We probe for an ne1000 or an ne2000 by testing whether the
23849            * on board is reachable through the dp8390. Note that the
23850            * ne1000 is an 8bit card and has a memory region distict from
23851            * the 16bit ne2000
23852            */
23853
23854           for (dep->de_16bit= 0; dep->de_16bit < 2; dep->de_16bit++)
23855           {
23856                   /* Reset the ethernet card */
23857                   byte= inb_ne(dep, NE_RESET);
23858                   milli_delay(2);
23859                   outb_ne(dep, NE_RESET, byte);
23860                   milli_delay(2);
23861
23862                   /* Reset the dp8390 */
23863                   outb_reg0(dep, DP_CR, CR_STP | CR_DM_ABORT);
23864                   for (i= 0; i < 0x1000 && ((inb_reg0(dep, DP_ISR) & ISR_RST) == 0); i++)
23865                           ; /* Do nothing */
23866
23867                   /* Check if the dp8390 is really there */
23868                   if ((inb_reg0(dep, DP_CR) & (CR_STP|CR_DM_ABORT)) !=
23869                           (CR_STP|CR_DM_ABORT))
23870                   {
23871                           return 0;
23872                   }
23873
23874                   /* Put it in loop-back mode */
23875                   outb_reg0(dep, DP_RCR, RCR_MON);
23876                   outb_reg0(dep, DP_TCR, TCR_NORMAL);
23877                   if (dep->de_16bit)
23878                   {
23879                           outb_reg0(dep, DP_DCR, DCR_WORDWIDE | DCR_8BYTES |
23880                                   DCR_BMS);
23881                   }
23882                   else
23883                   {
23884                           outb_reg0(dep, DP_DCR, DCR_BYTEWIDE | DCR_8BYTES |
23885                                   DCR_BMS);
23886                   }
23887
23888                   if (dep->de_16bit)
23889                   {
```

```
23890                           loc1= NE2000_START;
23891                           loc2= NE2000_START + NE2000_SIZE - 4;
23892                           f= test_16;
23893                   }
23894                   else
23895                   {
23896                           loc1= NE1000_START;
23897                           loc2= NE1000_START + NE1000_SIZE - 4;
23898                           f= test_8;
23899                   }
23900                   if (f(dep, loc1, pat0) && f(dep, loc1, pat1) &&
23901                           f(dep, loc1, pat2) && f(dep, loc1, pat3) &&
23902                           f(dep, loc2, pat0) && f(dep, loc2, pat1) &&
23903                           f(dep, loc2, pat2) && f(dep, loc2, pat3))
23904                   {
23905                           /* We don't need a memory segment */
23906                           dep->de_linmem= 0;
23907                           dep->de_initf= ne_init;
23908                           dep->de_stopf= ne_stop;
23909                           dep->de_prog_IO= 1;
23910                           return 1;
23911                   }
23912           }
23913           return 0;
23914   }


23917   /*============================================================================*
23918    *                              test_8                                        *
23919    *============================================================================*/
23920   static int test_8(dep, pos, pat)
23921   dpeth_t *dep;
23922   int pos;
23923   u8_t *pat;
23924   {
23925           u8_t buf[4];
23926           int i;
23927           int r;
23928
23929           outb_reg0(dep, DP_ISR, 0xFF);
23930
23931           /* Setup a transfer to put the pattern. */
23932           outb_reg0(dep, DP_RBCR0, 4);
23933           outb_reg0(dep, DP_RBCR1, 0);
23934           outb_reg0(dep, DP_RSAR0, pos & 0xFF);
23935           outb_reg0(dep, DP_RSAR1, pos >> 8);
23936           outb_reg0(dep, DP_CR, CR_DM_RW | CR_PS_P0 | CR_STA);
23937
23938           for (i= 0; i<4; i++)
23939                   outb_ne(dep, NE_DATA, pat[i]);
23940
23941           for (i= 0; i<N; i++)
23942           {
23943                   if (inb_reg0(dep, DP_ISR) & ISR_RDC)
23944                           break;
23945           }
23946           if (i == N)
23947           {
23948                   printf("ne2000, test_8: remote dma failed to complete\n");
23949                   return 0;
```

```
23950           }
23951
23952           outb_reg0(dep, DP_RBCR0, 4);
23953           outb_reg0(dep, DP_RBCR1, 0);
23954           outb_reg0(dep, DP_RSAR0, pos & 0xFF);
23955           outb_reg0(dep, DP_RSAR1, pos >> 8);
23956           outb_reg0(dep, DP_CR, CR_DM_RR | CR_PS_P0 | CR_STA);
23957
23958           for (i= 0; i<4; i++)
23959                   buf[i]= inb_ne(dep, NE_DATA);
23960
23961           r= (memcmp(buf, pat, 4) == 0);
23962           return r;
23963   }


23966   /*============================================================================*
23967    *                              test_16                                       *
23968    *============================================================================*/
23969   static int test_16(dep, pos, pat)
23970   dpeth_t *dep;
23971   int pos;
23972   u8_t *pat;
23973   {
23974           u8_t buf[4];
23975           int i;
23976           int r;
23977
23978           outb_reg0(dep, DP_ISR, 0xFF);
23979
23980           /* Setup a transfer to put the pattern. */
23981           outb_reg0(dep, DP_RBCR0, 4);
23982           outb_reg0(dep, DP_RBCR1, 0);
23983           outb_reg0(dep, DP_RSAR0, pos & 0xFF);
23984           outb_reg0(dep, DP_RSAR1, pos >> 8);
23985           outb_reg0(dep, DP_CR, CR_DM_RW | CR_PS_P0 | CR_STA);
23986
23987           for (i= 0; i<4; i += 2)
23988           {
23989                   outw_ne(dep, NE_DATA, *(u16_t *)(pat+i));
23990           }
23991
23992           for (i= 0; i<N; i++)
23993           {
23994                   if (inb_reg0(dep, DP_ISR) & ISR_RDC)
23995                           break;
23996           }
23997           if (i == N)
23998           {
23999                   printf("ne2000, test_16: remote dma failed to complete\n");
24000                   return 0;
24001           }
24002
24003           outb_reg0(dep, DP_RBCR0, 4);
24004           outb_reg0(dep, DP_RBCR1, 0);
24005           outb_reg0(dep, DP_RSAR0, pos & 0xFF);
24006           outb_reg0(dep, DP_RSAR1, pos >> 8);
24007           outb_reg0(dep, DP_CR, CR_DM_RR | CR_PS_P0 | CR_STA);
24008
24009           for (i= 0; i<4; i += 2)
```

```
24010           {
24011                   *(u16_t *)(buf+i)= inw_ne(dep, NE_DATA);
24012           }
24013
24014           r= (memcmp(buf, pat, 4) == 0);
24015           return r;
24016   }


24019   /*==========================================================================*
24020    *                              ne_init                                     *
24021    *==========================================================================*/
24022   static void ne_init(dep)
24023   dpeth_t *dep;
24024   {
24025           int i;
24026           int word, sendq_nr;
24027
24028           /* Setup a transfer to get the ethernet address. */
24029           if (dep->de_16bit)
24030                   outb_reg0(dep, DP_RBCR0, 6*2);
24031           else
24032                   outb_reg0(dep, DP_RBCR0, 6);
24033           outb_reg0(dep, DP_RBCR1, 0);
24034           outb_reg0(dep, DP_RSAR0, 0);
24035           outb_reg0(dep, DP_RSAR1, 0);
24036           outb_reg0(dep, DP_CR, CR_DM_RR | CR_PS_P0 | CR_STA);
24037
24038           for (i= 0; i<6; i++)
24039           {
24040                   if (dep->de_16bit)
24041                   {
24042                           word= inw_ne(dep, NE_DATA);
24043                           dep->de_address.ea_addr[i]= word;
24044                   }
24045                   else
24046                   {
24047                           dep->de_address.ea_addr[i] = inb_ne(dep, NE_DATA);
24048                   }
24049           }
24050           dep->de_data_port= dep->de_base_port + NE_DATA;
24051           if (dep->de_16bit)
24052           {
24053                   dep->de_ramsize= NE2000_SIZE;
24054                   dep->de_offset_page= NE2000_START / DP_PAGESIZE;
24055           }
24056           else
24057           {
24058                   dep->de_ramsize= NE1000_SIZE;
24059                   dep->de_offset_page= NE1000_START / DP_PAGESIZE;
24060           }
24061
24062           /* Allocate one send buffer (1.5KB) per 8KB of on board memory. */
24063           sendq_nr= dep->de_ramsize / 0x2000;
24064           if (sendq_nr < 1)
24065                   sendq_nr= 1;
24066           else if (sendq_nr > SENDQ_NR)
24067                   sendq_nr= SENDQ_NR;
24068           dep->de_sendq_nr= sendq_nr;
24069           for (i= 0; i<sendq_nr; i++)
```

```
24070           {
24071                   dep->de_sendq[i].sq_sendpage= dep->de_offset_page +
24072                           i*SENDQ_PAGES;
24073           }
24074
24075           dep->de_startpage= dep->de_offset_page + i*SENDQ_PAGES;
24076           dep->de_stoppage= dep->de_offset_page + dep->de_ramsize / DP_PAGESIZE;
24077
24078           /* Can't override the default IRQ. */
24079           dep->de_irq &= ~DEI_DEFAULT;
24080
24081           if (!debug)
24082           {
24083                   printf("ne2000: NE%d000 at %X:%d\n",
24084                           dep->de_16bit ? 2 : 1,
24085                           dep->de_base_port, dep->de_irq);
24086           }
24087           else
24088           {
24089                   printf("ne2000: Novell %s ethernet card ",
24090                           dep->de_16bit ? "16-bit (ne2000)" : "8-bit (ne1000)");
24091                   printf("at I/O address 0x%X, memory size 0x%X, irq %d\n",
24092                           dep->de_base_port, dep->de_ramsize, dep->de_irq);
24093           }
24094   }


24097   /*==========================================================================*
24098    *                              ne_stop                                     *
24099    *==========================================================================*/
24100   static void ne_stop(dep)
24101   dpeth_t *dep;
24102   {
24103           int byte;
24104
24105           /* Reset the ethernet card */
24106           byte= inb_ne(dep, NE_RESET);
24107           milli_delay(2);
24108           outb_ne(dep, NE_RESET, byte);
24109   }

24111   #endif /* ENABLE_NETWORKING */
24112
24113   /*
24114    * $PchHeader: /mount/hd2/minix/sys/kernel/ibm/RCS/ne2000.c,v 1.2 1995/01/12 21:48:53 philip Exp $
24115    */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/printer.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

24200    /* This file contains the printer driver. It is a fairly simple driver,
24201     * supporting only one printer.  Characters that are written to the driver
24202     * are written to the printer without any changes at all.
24203     *
24204     * The valid messages and their parameters are:
24205     *
24206     *   HARD_INT:     interrupt handler has finished current chunk of output
24207     *   DEV_WRITE:    a process wants to write on a terminal
24208     *   CANCEL:       terminate a previous incomplete system call immediately
24209     *
24210     *    m_type      TTY_LINE   PROC_NR    COUNT    ADDRESS
24211     * ---------------------------------------------------
24212     * | HARD_INT    |         |         |         |         |
24213     * |-------------+---------+---------+---------+---------|
24214     * | DEV_WRITE   |minor dev| proc nr |  count  | buf ptr |
24215     * |-------------+---------+---------+---------+---------|
24216     * | CANCEL      |minor dev| proc nr |         |         |
24217     * ---------------------------------------------------
24218     *
24219     * Note: since only 1 printer is supported, minor dev is not used at present.
24220     */
24221
24222    #include "kernel.h"
24223    #include <minix/callnr.h>
24224    #include <minix/com.h>
24225    #include "proc.h"
24226
24227    /* Control bits (in port_base + 2).  "+" means positive logic and "-" means
24228     * negative logic.  Most of the signals are negative logic on the pins but
24229     * many are converted to positive logic in the ports.  Some manuals are
24230     * misleading because they only document the pin logic.
24231     *
24232     *      +0x01   Pin 1   -Strobe
24233     *      +0x02   Pin 14  -Auto Feed
24234     *      -0x04   Pin 16  -Initialize Printer
24235     *      +0x08   Pin 17  -Select Printer
24236     *      +0x10   IRQ7 Enable
24237     *
24238     * Auto Feed and Select Printer are always enabled. Strobe is enabled briefly
24239     * when characters are output.  Initialize Printer is enabled briefly when
24240     * the task is started.  IRQ7 is enabled when the first character is output
24241     * and left enabled until output is completed (or later after certain
24242     * abnormal completions).
24243     */
24244    #define ASSERT_STROBE   0x1D    /* strobe a character to the interface */
24245    #define NEGATE_STROBE   0x1C    /* enable interrupt on interface */
24246    #define SELECT          0x0C    /* select printer bit */
24247    #define INIT_PRINTER    0x08    /* init printer bits */
24248
24249    /* Status bits (in port_base + 2).
24250     *
24251     *      -0x08   Pin 15  -Error
24252     *      +0x10   Pin 13  +Select Status
24253     *      +0x20   Pin 12  +Out of Paper
24254     *      -0x40   Pin 10  -Acknowledge
```

```
24255     *      -0x80   Pin 11  +Busy
24256     */
24257    #define BUSY_STATUS     0x10    /* printer gives this status when busy */
24258    #define NO_PAPER        0x20    /* status bit saying that paper is out */
24259    #define NORMAL_STATUS   0x90    /* printer gives this status when idle */
24260    #define ON_LINE         0x10    /* status bit saying that printer is online */
24261    #define STATUS_MASK     0xB0    /* mask to filter out status bits */
24262
24263    /* Centronics interface timing that must be met by software (in microsec).
24264     *
24265     * Strobe length:       0.5u to 100u (not sure about the upper limit).
24266     * Data set up:         0.5u before strobe.
24267     * Data hold:                   0.5u after strobe.
24268     * Init pulse length:      over 200u (not sure).
24269     *
24270     * The strobe length is about 50u with the code here and function calls for
24271     * out_byte() - not much to spare.  The 0.5u minimums may be violated if
24272     * out_byte() is generated in-line on a fast machine.  Some printer boards
24273     * are slower than 0.5u anyway.
24274     */
24275
24276    PRIVATE int caller;             /* process to tell when printing done (FS) */
24277    PRIVATE int done_status;        /* status of last output completion */
24278    PRIVATE int oleft;              /* bytes of output left in obuf */
24279    PRIVATE char obuf[128];         /* output buffer */
24280    PRIVATE int opending;           /* nonzero while expected printing not done */
24281    PRIVATE char *optr;             /* ptr to next char in obuf to print */
24282    PRIVATE int orig_count;         /* original byte count */
24283    PRIVATE int port_base;          /* I/O port for printer */
24284    PRIVATE int proc_nr;            /* user requesting the printing */
24285    PRIVATE int user_left;          /* bytes of output left in user buf */
24286    PRIVATE vir_bytes user_vir;     /* address of remainder of user buf */
24287    PRIVATE int writing;            /* nonzero while write is in progress */
24288
24289    FORWARD _PROTOTYPE( void do_cancel, (message *m_ptr) );
24290    FORWARD _PROTOTYPE( void do_done, (void) );
24291    FORWARD _PROTOTYPE( void do_write, (message *m_ptr) );
24292    FORWARD _PROTOTYPE( void pr_start, (void) );
24293    FORWARD _PROTOTYPE( void print_init, (void) );
24294    FORWARD _PROTOTYPE( void reply, (int code, int replyee, int process,
24295                    int status) );
24296    FORWARD _PROTOTYPE( int pr_handler, (int irq) );
24297
24298    /*===========================================================================*
24299     *                              printer_task                                 *
24300     *===========================================================================*/
24301    PUBLIC void printer_task()
24302    {
24303    /* Main routine of the printer task. */
24304
24305      message pr_mess;              /* buffer for all incoming messages */
24306
24307      print_init();                 /* initialize */
24308
24309      while (TRUE) {
24310            receive(ANY, &pr_mess);
24311            switch(pr_mess.m_type) {
24312                case DEV_OPEN:
24313                case DEV_CLOSE:
24314                    reply(TASK_REPLY, pr_mess.m_source, pr_mess.PROC_NR, OK);
```

```
24315                   break;
24316               case DEV_WRITE:     do_write(&pr_mess);     break;
24317               case CANCEL   :     do_cancel(&pr_mess);    break;
24318               case HARD_INT :     do_done();              break;
24319               default:
24320                   reply(TASK_REPLY, pr_mess.m_source, pr_mess.PROC_NR, EINVAL);
24321           }
24322     }
24323   }


24326   /*===========================================================================*
24327    *                                do_write                                   *
24328    *===========================================================================*/
24329   PRIVATE void do_write(m_ptr)
24330   register message *m_ptr;        /* pointer to the newly arrived message */
24331   {
24332   /* The printer is used by sending DEV_WRITE messages to it. Process one. */
24333
24334     register int r;
24335
24336     /* Reject command if last write is not finished, count not positive, or
24337      * user address bad.
24338      */
24339     if (writing) {
24340           r = EIO;
24341     } else
24342     if (m_ptr->COUNT <= 0) {
24343           r = EINVAL;
24344     } else
24345     if (numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, m_ptr->COUNT) == 0) {
24346           r = EFAULT;
24347     } else {
24348           /* Save information needed later. */
24349           caller = m_ptr->m_source;
24350           proc_nr = m_ptr->PROC_NR;
24351           user_left = m_ptr->COUNT;
24352           orig_count = m_ptr->COUNT;
24353           user_vir = (vir_bytes) m_ptr->ADDRESS;
24354           pr_start();
24355           writing = TRUE;
24356           r = SUSPEND;
24357     }
24358
24359     /* Reply to FS, no matter what happened. */
24360     reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
24361   }


24364   /*===========================================================================*
24365    *                                do_done                                    *
24366    *===========================================================================*/
24367   PRIVATE void do_done()
24368   {
24369   /* Previous chunk of printing is finished.  Continue if OK and more.
24370    * Otherwise, reply to caller (FS).
24371    */
24372
24373     register int status;
24374
```

```
24375     if (!writing) return;         /* interrupt while canceling */
24376     if (done_status != OK) {
24377           /* Printer error. */
24378           status = EIO;
24379           if ((done_status & ON_LINE) == 0) {
24380                   printf("Printer is not on line\n");
24381           } else
24382           if (done_status & NO_PAPER) {
24383                   status = EAGAIN;        /* out of paper */
24384           } else {
24385                   printf("Printer error, status is 0x%02X\n", done_status);
24386           }
24387           if (status == EAGAIN && user_left < orig_count) {
24388                   /* Some characters have been printed, tell how many. */
24389                   status = orig_count - user_left;
24390           }
24391           oleft = 0;              /* cancel output by interrupt handler */
24392     } else if (user_left != 0) {
24393           pr_start();
24394           return;
24395     } else {
24396           status = orig_count;
24397     }
24398     reply(REVIVE, caller, proc_nr, status);
24399     writing = FALSE;
24400   }


24403   /*==========================================================================*
24404    *                              do_cancel                                   *
24405    *==========================================================================*/
24406   PRIVATE void do_cancel(m_ptr)
24407   register message *m_ptr;        /* pointer to the newly arrived message */
24408   {
24409   /* Cancel a print request that has already started.  Usually this means that
24410    * the process doing the printing has been killed by a signal.  It is not
24411    * clear if there are race conditions.  Try not to cancel the wrong process,
24412    * but rely on FS to handle the EINTR reply and de-suspension properly.
24413    */
24414
24415     if (writing && m_ptr->PROC_NR == proc_nr) {
24416           oleft = 0;              /* cancel output by interrupt handler */
24417           writing = FALSE;
24418     }
24419     reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, EINTR);
24420   }


24423   /*==========================================================================*
24424    *                              reply                                       *
24425    *==========================================================================*/
24426   PRIVATE void reply(code, replyee, process, status)
24427   int code;                       /* TASK_REPLY or REVIVE */
24428   int replyee;                    /* destination for message (normally FS) */
24429   int process;                    /* which user requested the printing */
24430   int status;                     /* number of  chars printed or error code */
24431   {
24432   /* Send a reply telling FS that printing has started or stopped. */
24433
24434     message pr_mess;
```

```
24435
24436     pr_mess.m_type = code;        /* TASK_REPLY or REVIVE */
24437     pr_mess.REP_STATUS = status;  /* count or EIO */
24438     pr_mess.REP_PROC_NR = process;        /* which user does this pertain to */
24439     send(replyee, &pr_mess);      /* send the message */
24440   }


24443   /*===========================================================================*
24444    *                              print_init                                   *
24445    *===========================================================================*/
24446   PRIVATE void print_init()
24447   {
24448   /* Set global variables.  Get the port base for the first printer from the
24449    * BIOS and initialize the printer.
24450    */
24451
24452     phys_copy(0x408L, vir2phys(&port_base), 2L);
24453     out_byte(port_base + 2, INIT_PRINTER);
24454     milli_delay(2);               /* easily satisfies Centronics minimum */
24455     out_byte(port_base + 2, SELECT);
24456     put_irq_handler(PRINTER_IRQ, pr_handler);
24457     enable_irq(PRINTER_IRQ);              /* ready for printer interrupts */
24458   }


24461   /*===========================================================================*
24462    *                              pr_start                                     *
24463    *===========================================================================*/
24464   PRIVATE void pr_start()
24465   {
24466   /* Start next chunk of printer output. */
24467
24468     register int chunk;
24469     phys_bytes user_phys;
24470
24471     if ( (chunk = user_left) > sizeof obuf) chunk = sizeof obuf;
24472     user_phys = proc_vir2phys(proc_addr(proc_nr), user_vir);
24473     phys_copy(user_phys, vir2phys(obuf), (phys_bytes) chunk);
24474     optr = obuf;
24475     opending = TRUE;
24476     oleft = chunk;                /* now interrupt handler is enabled */
24477   }


24480   /*===========================================================================*
24481    *                              pr_handler                                   *
24482    *===========================================================================*/
24483   PRIVATE int pr_handler(irq)
24484   int irq;
24485   {
24486   /* This is the interrupt handler.  When a character has been printed, an
24487    * interrupt occurs, and the assembly code routine trapped to calls
24488    * pr_handler().
24489    *
24490    * One problem is that the 8259A controller generates spurious interrupts to
24491    * IRQ7 when it gets confused by mistimed interrupts on any line.  (IRQ7 for
24492    * the first controller happens to be the printer IRQ.)  Such an interrupt is
24493    * ignored as a side-affect of the method of checking the busy status.  This
24494    * is harmless for the printer task but probably fatal to the task that missed
```

```
24495     * the interrupt.  It may be possible to recover by doing more work here.
24496     */
24497
24498      register int status;
24499
24500      if (oleft == 0) {
24501            /* Nothing more to print.  Turn off printer interrupts in case they
24502             * are level-sensitive as on the PS/2.  This should be safe even
24503             * when the printer is busy with a previous character, because the
24504             * interrupt status does not affect the printer.
24505             */
24506            out_byte(port_base + 2, SELECT);
24507            return 1;
24508      }
24509
24510      do {
24511            /* Loop to handle fast (buffered) printers.  It is important that
24512             * processor interrupts are not disabled here, just printer interrupts.
24513             */
24514            status = in_byte(port_base + 1);
24515            if ((status & STATUS_MASK) == BUSY_STATUS) {
24516                    /* Still busy with last output.  This normally happens
24517                     * immediately after doing output to an unbuffered or slow
24518                     * printer.  It may happen after a call from pr_start or
24519                     * pr_restart, since they are not synchronized with printer
24520                     * interrupts.  It may happen after a spurious interrupt.
24521                     */
24522                    return 1;
24523            }
24524            if ((status & STATUS_MASK) == NORMAL_STATUS) {
24525                    /* Everything is all right.  Output another character. */
24526                    out_byte(port_base, *optr++);   /* output character */
24527                    lock();         /* ensure strobe is not too long */
24528                    out_byte(port_base + 2, ASSERT_STROBE);
24529                    out_byte(port_base + 2, NEGATE_STROBE);
24530                    unlock();
24531                    opending = FALSE;       /* show interrupt is working */
24532
24533                    user_vir++;
24534                    user_left--;
24535            } else {
24536                    /* Error.  This would be better ignored (treat as busy). */
24537                    done_status = status;
24538                    interrupt(PRINTER);
24539                    return 1;
24540            }
24541      }
24542      while (--oleft != 0);
24543
24544      /* Finished printing chunk OK. */
24545      done_status = OK;
24546      interrupt(PRINTER);
24547      return 1;     /* Reenable printer interrupt */
24548    }


24551    /*==========================================================================*
24552     *                              pr_restart                                  *
24553     *==========================================================================*/
24554    PUBLIC void pr_restart()
```

```
24555    {
24556    /* Check if printer is hung up, and if so, restart it.
24557     * Disable_irq() returns true if the irq could be disabled, so that
24558     * pr_restart() is not reentered.
24559     */
24560
24561      if (oleft != 0) {
24562            if (opending && disable_irq(PRINTER_IRQ)) {
24563                    (void) pr_handler(PRINTER_IRQ);
24564
24565                    /* ready for printer interrupts again */
24566                    enable_irq(PRINTER_IRQ);
24567            }
24568            opending = TRUE;        /* expect some printing before next call */
24569      }
24570    }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/proc.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
24600    /* This file contains essentially all of the process and message handling.
24601     * It has two main entry points from the outside:
24602     *
24603     *   sys_call:   called when a process or task does SEND, RECEIVE or SENDREC
24604     *   interrupt: called by interrupt routines to send a message to task
24605     *
24606     * It also has several minor entry points:
24607     *
24608     *   lock_ready:      put a process on one of the ready queues so it can be run
24609     *   lock_unready:    remove a process from the ready queues
24610     *   lock_sched:      a process has run too long; schedule another one
24611     *   lock_mini_send:  send a message (used by interrupt signals, etc.)
24612     *   lock_pick_proc:  pick a process to run (used by system initialization)
24613     *   unhold:          repeat all held-up interrupts
24614     */
24615
24616    #include "kernel.h"
24617    #include <minix/callnr.h>
24618    #include <minix/com.h>
24619    #include "proc.h"
24620
24621    PRIVATE unsigned char switching;        /* nonzero to inhibit interrupt() */
24622
24623    FORWARD _PROTOTYPE( int mini_send, (struct proc *caller_ptr, int dest,
24624                    message *m_ptr) );
24625    FORWARD _PROTOTYPE( int mini_rec, (struct proc *caller_ptr, int src,
24626                    message *m_ptr) );
24627    FORWARD _PROTOTYPE( void ready, (struct proc *rp) );
24628    FORWARD _PROTOTYPE( void sched, (void) );
24629    FORWARD _PROTOTYPE( void unready, (struct proc *rp) );
24630    FORWARD _PROTOTYPE( void pick_proc, (void) );
24631
24632    #if (CHIP == M68000)
24633    FORWARD _PROTOTYPE( void cp_mess, (int src, struct proc *src_p, message *src_m,
24634                    struct proc *dst_p, message *dst_m) );
```

```
24635   #endif
24636
24637   #if (CHIP == INTEL)
24638   #define CopyMess(s,sp,sm,dp,dm) \
24639           cp_mess(s, (sp)->p_map[D].mem_phys, (vir_bytes)sm, (dp)->p_map[D].mem_phys, (vir_bytes)dm)
24640   #endif
24641
24642   #if (CHIP == M68000)
24643   #define CopyMess(s,sp,sm,dp,dm) \
24644           cp_mess(s,sp,sm,dp,dm)
24645   #endif
24646
24647   /*===========================================================================*
24648    *                              interrupt                                    *
24649    *===========================================================================*/
24650   PUBLIC void interrupt(task)
24651   int task;                       /* number of task to be started */
24652   {
24653   /* An interrupt has occurred.  Schedule the task that handles it. */
24654
24655     register struct proc *rp;     /* pointer to task's proc entry */
24656
24657     rp = proc_addr(task);
24658
24659     /* If this call would compete with other process-switching functions, put
24660      * it on the 'held' queue to be flushed at the next non-competing restart().
24661      * The competing conditions are:
24662      * (1) k_reenter == (typeof k_reenter) -1:
24663      *     Call from the task level, typically from an output interrupt
24664      *     routine.  An interrupt handler might reenter interrupt().  Rare,
24665      *     so not worth special treatment.
24666      * (2) k_reenter > 0:
24667      *     Call from a nested interrupt handler.  A previous interrupt handler
24668      *     might be inside interrupt() or sys_call().
24669      * (3) switching != 0:
24670      *     Some process-switching function other than interrupt() is being
24671      *     called from the task level, typically sched() from CLOCK.  An
24672      *     interrupt handler might call interrupt and pass the k_reenter test.
24673      */
24674     if (k_reenter != 0 || switching) {
24675           lock();
24676           if (!rp->p_int_held) {
24677                   rp->p_int_held = TRUE;
24678                   if (held_head != NIL_PROC)
24679                           held_tail->p_nextheld = rp;
24680                   else
24681                           held_head = rp;
24682                   held_tail = rp;
24683                   rp->p_nextheld = NIL_PROC;
24684           }
24685           unlock();
24686           return;
24687     }
24688
24689     /* If task is not waiting for an interrupt, record the blockage. */
24690     if ( (rp->p_flags & (RECEIVING | SENDING)) != RECEIVING ||
24691         !isrxhardware(rp->p_getfrom)) {
24692           rp->p_int_blocked = TRUE;
24693           return;
24694     }
```

```
24695
24696       /* Destination is waiting for an interrupt.
24697        * Send it a message with source HARDWARE and type HARD_INT.
24698        * No more information can be reliably provided since interrupt messages
24699        * are not queued.
24700        */
24701       rp->p_messbuf->m_source = HARDWARE;
24702       rp->p_messbuf->m_type = HARD_INT;
24703       rp->p_flags &= ~RECEIVING;
24704       rp->p_int_blocked = FALSE;
24705
24706        /* Make rp ready and run it unless a task is already running.  This is
24707         * ready(rp) in-line for speed.
24708         */
24709       if (rdy_head[TASK_Q] != NIL_PROC)
24710             rdy_tail[TASK_Q]->p_nextready = rp;
24711       else
24712             proc_ptr = rdy_head[TASK_Q] = rp;
24713       rdy_tail[TASK_Q] = rp;
24714       rp->p_nextready = NIL_PROC;
24715   }

24717   /*==========================================================================*
24718    *                              sys_call                                    *
24719    *==========================================================================*/
24720   PUBLIC int sys_call(function, src_dest, m_ptr)
24721   int function;                   /* SEND, RECEIVE, or BOTH */
24722   int src_dest;                   /* source to receive from or dest to send to */
24723   message *m_ptr;                 /* pointer to message */
24724   {
24725   /* The only system calls that exist in MINIX are sending and receiving
24726    * messages.  These are done by trapping to the kernel with an INT instruction.
24727    * The trap is caught and sys_call() is called to send or receive a message
24728    * (or both). The caller is always given by proc_ptr.
24729    */
24730
24731     register struct proc *rp;
24732     int n;
24733
24734     /* Check for bad system call parameters. */
24735     if (!isoksrc_dest(src_dest)) return(E_BAD_SRC);
24736     rp = proc_ptr;
24737
24738     if (isuserp(rp) && function != BOTH) return(E_NO_PERM);
24739
24740     /* The parameters are ok. Do the call. */
24741     if (function & SEND) {
24742           /* Function = SEND or BOTH. */
24743           n = mini_send(rp, src_dest, m_ptr);
24744           if (function == SEND || n != OK)
24745                   return(n);      /* done, or SEND failed */
24746     }
24747
24748     /* Function = RECEIVE or BOTH.
24749      * We have checked user calls are BOTH, and trust 'function' otherwise.
24750      */
24751     return(mini_rec(rp, src_dest, m_ptr));
24752   }

24754   /*==========================================================================*
```

```
24755    *                              mini_send                                        *
24756    *===============================================================================*/
24757   PRIVATE int mini_send(caller_ptr, dest, m_ptr)
24758   register struct proc *caller_ptr;       /* who is trying to send a message? */
24759   int dest;                       /* to whom is message being sent? */
24760   message *m_ptr;                 /* pointer to message buffer */
24761   {
24762   /* Send a message from 'caller_ptr' to 'dest'. If 'dest' is blocked waiting
24763    * for this message, copy the message to it and unblock 'dest'. If 'dest' is
24764    * not waiting at all, or is waiting for another source, queue 'caller_ptr'.
24765    */
24766
24767     register struct proc *dest_ptr, *next_ptr;
24768     vir_bytes vb;                 /* message buffer pointer as vir_bytes */
24769     vir_clicks vlo, vhi;          /* virtual clicks containing message to send */
24770
24771     /* User processes are only allowed to send to FS and MM.  Check for this. */
24772     if (isuserp(caller_ptr) && !issysentn(dest)) return(E_BAD_DEST);
24773     dest_ptr = proc_addr(dest);   /* pointer to destination's proc entry */
24774     if (dest_ptr->p_flags & P_SLOT_FREE) return(E_BAD_DEST);      /* dead dest */
24775
24776   #if ALLOW_GAP_MESSAGES
24777     /* This check allows a message to be anywhere in data or stack or gap.
24778      * It will have to be made more elaborate later for machines which
24779      * don't have the gap mapped.
24780      */
24781     vb = (vir_bytes) m_ptr;
24782     vlo = vb >> CLICK_SHIFT;      /* vir click for bottom of message */
24783     vhi = (vb + MESS_SIZE - 1) >> CLICK_SHIFT;    /* vir click for top of msg */
24784     if (vlo < caller_ptr->p_map[D].mem_vir || vlo > vhi ||
24785         vhi >= caller_ptr->p_map[S].mem_vir + caller_ptr->p_map[S].mem_len)
24786           return(EFAULT);
24787   #else
24788     /* Check for messages wrapping around top of memory or outside data seg. */
24789     vb = (vir_bytes) m_ptr;
24790     vlo = vb >> CLICK_SHIFT;      /* vir click for bottom of message */
24791     vhi = (vb + MESS_SIZE - 1) >> CLICK_SHIFT;    /* vir click for top of msg */
24792     if (vhi < vlo ||
24793         vhi - caller_ptr->p_map[D].mem_vir >= caller_ptr->p_map[D].mem_len)
24794           return(EFAULT);
24795   #endif
24796
24797     /* Check for deadlock by 'caller_ptr' and 'dest' sending to each other. */
24798     if (dest_ptr->p_flags & SENDING) {
24799           next_ptr = proc_addr(dest_ptr->p_sendto);
24800           while (TRUE) {
24801                   if (next_ptr == caller_ptr) return(ELOCKED);
24802                   if (next_ptr->p_flags & SENDING)
24803                           next_ptr = proc_addr(next_ptr->p_sendto);
24804                   else
24805                           break;
24806           }
24807     }
24808
24809     /* Check to see if 'dest' is blocked waiting for this message. */
24810     if ( (dest_ptr->p_flags & (RECEIVING | SENDING)) == RECEIVING &&
24811          (dest_ptr->p_getfrom == ANY ||
24812           dest_ptr->p_getfrom == proc_number(caller_ptr))) {
24813           /* Destination is indeed waiting for this message. */
24814           CopyMess(proc_number(caller_ptr), caller_ptr, m_ptr, dest_ptr,
```

```
24815                   dest_ptr->p_messbuf);
24816           dest_ptr->p_flags &= ~RECEIVING;        /* deblock destination */
24817           if (dest_ptr->p_flags == 0) ready(dest_ptr);
24818      } else {
24819           /* Destination is not waiting.  Block and queue caller. */
24820           caller_ptr->p_messbuf = m_ptr;
24821           if (caller_ptr->p_flags == 0) unready(caller_ptr);
24822           caller_ptr->p_flags |= SENDING;
24823           caller_ptr->p_sendto= dest;
24824
24825           /* Process is now blocked.  Put in on the destination's queue. */
24826           if ( (next_ptr = dest_ptr->p_callerq) == NIL_PROC)
24827                   dest_ptr->p_callerq = caller_ptr;
24828           else {
24829                   while (next_ptr->p_sendlink != NIL_PROC)
24830                           next_ptr = next_ptr->p_sendlink;
24831                   next_ptr->p_sendlink = caller_ptr;
24832           }
24833           caller_ptr->p_sendlink = NIL_PROC;
24834      }
24835      return(OK);
24836    }

24838    /*===========================================================================*
24839     *                              mini_rec                                     *
24840     *===========================================================================*/
24841    PRIVATE int mini_rec(caller_ptr, src, m_ptr)
24842    register struct proc *caller_ptr;       /* process trying to get message */
24843    int src;                        /* which message source is wanted (or ANY) */
24844    message *m_ptr;                 /* pointer to message buffer */
24845    {
24846    /* A process or task wants to get a message.  If one is already queued,
24847     * acquire it and deblock the sender.  If no message from the desired source
24848     * is available, block the caller.  No need to check parameters for validity.
24849     * Users calls are always sendrec(), and mini_send() has checked already.
24850     * Calls from the tasks, MM, and FS are trusted.
24851     */
24852
24853      register struct proc *sender_ptr;
24854      register struct proc *previous_ptr;
24855
24856      /* Check to see if a message from desired source is already available. */
24857      if (!(caller_ptr->p_flags & SENDING)) {
24858           /* Check caller queue. */
24859        for (sender_ptr = caller_ptr->p_callerq; sender_ptr != NIL_PROC;
24860             previous_ptr = sender_ptr, sender_ptr = sender_ptr->p_sendlink) {
24861           if (src == ANY || src == proc_number(sender_ptr)) {
24862                   /* An acceptable message has been found. */
24863                   CopyMess(proc_number(sender_ptr), sender_ptr,
24864                            sender_ptr->p_messbuf, caller_ptr, m_ptr);
24865                   if (sender_ptr == caller_ptr->p_callerq)
24866                           caller_ptr->p_callerq = sender_ptr->p_sendlink;
24867                   else
24868                           previous_ptr->p_sendlink = sender_ptr->p_sendlink;
24869                   if ((sender_ptr->p_flags &= ~SENDING) == 0)
24870                           ready(sender_ptr);      /* deblock sender */
24871                   return(OK);
24872           }
24873        }
24874
```

```
24875         /* Check for blocked interrupt. */
24876         if (caller_ptr->p_int_blocked && isrxhardware(src)) {
24877             m_ptr->m_source = HARDWARE;
24878             m_ptr->m_type = HARD_INT;
24879             caller_ptr->p_int_blocked = FALSE;
24880             return(OK);
24881         }
24882       }
24883
24884       /* No suitable message is available.  Block the process trying to receive. */
24885       caller_ptr->p_getfrom = src;
24886       caller_ptr->p_messbuf = m_ptr;
24887       if (caller_ptr->p_flags == 0) unready(caller_ptr);
24888       caller_ptr->p_flags |= RECEIVING;
24889
24890       /* If MM has just blocked and there are kernel signals pending, now is the
24891        * time to tell MM about them, since it will be able to accept the message.
24892        */
24893       if (sig_procs > 0 && proc_number(caller_ptr) == MM_PROC_NR && src == ANY)
24894             inform();
24895       return(OK);
24896     }

24898     /*===========================================================================*
24899      *                              pick_proc                                    *
24900      *===========================================================================*/
24901     PRIVATE void pick_proc()
24902     {
24903     /* Decide who to run now.  A new process is selected by setting 'proc_ptr'.
24904      * When a fresh user (or idle) process is selected, record it in 'bill_ptr',
24905      * so the clock task can tell who to bill for system time.
24906      */
24907
24908       register struct proc *rp;     /* process to run */
24909
24910       if ( (rp = rdy_head[TASK_Q]) != NIL_PROC) {
24911             proc_ptr = rp;
24912             return;
24913       }
24914       if ( (rp = rdy_head[SERVER_Q]) != NIL_PROC) {
24915             proc_ptr = rp;
24916             return;
24917       }
24918       if ( (rp = rdy_head[USER_Q]) != NIL_PROC) {
24919             proc_ptr = rp;
24920             bill_ptr = rp;
24921             return;
24922       }
24923       /* No one is ready.  Run the idle task.  The idle task might be made an
24924        * always-ready user task to avoid this special case.
24925        */
24926       bill_ptr = proc_ptr = proc_addr(IDLE);
24927     }

24929     /*===========================================================================*
24930      *                              ready                                        *
24931      *===========================================================================*/
24932     PRIVATE void ready(rp)
24933     register struct proc *rp;       /* this process is now runnable */
24934     {
```

```
24935   /* Add 'rp' to the end of one of the queues of runnable processes. Three
24936    * queues are maintained:
24937    *   TASK_Q   - (highest priority) for runnable tasks
24938    *   SERVER_Q - (middle priority) for MM and FS only
24939    *   USER_Q   - (lowest priority) for user processes
24940    */
24941
24942     if (istaskp(rp)) {
24943           if (rdy_head[TASK_Q] != NIL_PROC)
24944                   /* Add to tail of nonempty queue. */
24945                   rdy_tail[TASK_Q]->p_nextready = rp;
24946           else {
24947                   proc_ptr =              /* run fresh task next */
24948                   rdy_head[TASK_Q] = rp;  /* add to empty queue */
24949           }
24950           rdy_tail[TASK_Q] = rp;
24951           rp->p_nextready = NIL_PROC;     /* new entry has no successor */
24952           return;
24953     }
24954     if (!isuserp(rp)) {           /* others are similar */
24955           if (rdy_head[SERVER_Q] != NIL_PROC)
24956                   rdy_tail[SERVER_Q]->p_nextready = rp;
24957           else
24958                   rdy_head[SERVER_Q] = rp;
24959           rdy_tail[SERVER_Q] = rp;
24960           rp->p_nextready = NIL_PROC;
24961           return;
24962     }
24963   #if (SHADOWING == 1)
24964     if (isshadowp(rp)) {          /* others are similar */
24965           if (rdy_head[SHADOW_Q] != NIL_PROC)
24966                   rdy_tail[SHADOW_Q]->p_nextready = rp;
24967           else
24968                   rdy_head[SHADOW_Q] = rp;
24969           rdy_tail[SHADOW_Q] = rp;
24970           rp->p_nextready = NIL_PROC;
24971           return;
24972     }
24973   #endif
24974     if (rdy_head[USER_Q] == NIL_PROC)
24975           rdy_tail[USER_Q] = rp;
24976     rp->p_nextready = rdy_head[USER_Q];
24977     rdy_head[USER_Q] = rp;
24978   /*
24979     if (rdy_head[USER_Q] != NIL_PROC)
24980           rdy_tail[USER_Q]->p_nextready = rp;
24981     else
24982           rdy_head[USER_Q] = rp;
24983     rdy_tail[USER_Q] = rp;
24984     rp->p_nextready = NIL_PROC;
24985   */
24986   }

24988   /*===========================================================================*
24989    *                               unready                                     *
24990    *===========================================================================*/
24991   PRIVATE void unready(rp)
24992   register struct proc *rp;       /* this process is no longer runnable */
24993   {
24994   /* A process has blocked. */
```

```
24995
24996     register struct proc *xp;
24997     register struct proc **qtail;  /* TASK_Q, SERVER_Q, or USER_Q rdy_tail */
24998
24999     if (istaskp(rp)) {
25000           /* task stack still ok? */
25001           if (*rp->p_stguard != STACK_GUARD)
25002                   panic("stack overrun by task", proc_number(rp));
25003
25004           if ( (xp = rdy_head[TASK_Q]) == NIL_PROC) return;
25005           if (xp == rp) {
25006                   /* Remove head of queue */
25007                   rdy_head[TASK_Q] = xp->p_nextready;
25008                   if (rp == proc_ptr) pick_proc();
25009                   return;
25010           }
25011           qtail = &rdy_tail[TASK_Q];
25012     }
25013     else if (!isuserp(rp)) {
25014           if ( (xp = rdy_head[SERVER_Q]) == NIL_PROC) return;
25015           if (xp == rp) {
25016                   rdy_head[SERVER_Q] = xp->p_nextready;
25017   #if (CHIP == M68000)
25018                   if (rp == proc_ptr)
25019   #endif
25020                   pick_proc();
25021                   return;
25022           }
25023           qtail = &rdy_tail[SERVER_Q];
25024     } else
25025   #if (SHADOWING == 1)
25026     if (isshadowp(rp)) {
25027           if ( (xp = rdy_head[SHADOW_Q]) == NIL_PROC) return;
25028           if (xp == rp) {
25029                   rdy_head[SHADOW_Q] = xp->p_nextready;
25030                   if (rp == proc_ptr)
25031                           pick_proc();
25032                   return;
25033           }
25034           qtail = &rdy_tail[SHADOW_Q];
25035     } else
25036   #endif
25037     {
25038           if ( (xp = rdy_head[USER_Q]) == NIL_PROC) return;
25039           if (xp == rp) {
25040                   rdy_head[USER_Q] = xp->p_nextready;
25041   #if (CHIP == M68000)
25042                   if (rp == proc_ptr)
25043   #endif
25044                   pick_proc();
25045                   return;
25046           }
25047           qtail = &rdy_tail[USER_Q];
25048     }
25049
25050     /* Search body of queue.  A process can be made unready even if it is
25051      * not running by being sent a signal that kills it.
25052      */
25053     while (xp->p_nextready != rp)
25054           if ( (xp = xp->p_nextready) == NIL_PROC) return;
```

```
25055     xp->p_nextready = xp->p_nextready->p_nextready;
25056     if (*qtail == rp) *qtail = xp;
25057   }

25059   /*===========================================================================*
25060    *                              sched                                        *
25061    *===========================================================================*/
25062   PRIVATE void sched()
25063   {
25064   /* The current process has run too long.  If another low priority (user)
25065    * process is runnable, put the current process on the end of the user queue,
25066    * possibly promoting another user to head of the queue.
25067    */
25068
25069     if (rdy_head[USER_Q] == NIL_PROC) return;
25070
25071     /* One or more user processes queued. */
25072     rdy_tail[USER_Q]->p_nextready = rdy_head[USER_Q];
25073     rdy_tail[USER_Q] = rdy_head[USER_Q];
25074     rdy_head[USER_Q] = rdy_head[USER_Q]->p_nextready;
25075     rdy_tail[USER_Q]->p_nextready = NIL_PROC;
25076     pick_proc();
25077   }

25079   /*===========================================================================*
25080    *                              lock_mini_send                               *
25081    *===========================================================================*/
25082   PUBLIC int lock_mini_send(caller_ptr, dest, m_ptr)
25083   struct proc *caller_ptr;        /* who is trying to send a message? */
25084   int dest;                       /* to whom is message being sent? */
25085   message *m_ptr;                 /* pointer to message buffer */
25086   {
25087   /* Safe gateway to mini_send() for tasks. */
25088
25089     int result;
25090
25091     switching = TRUE;
25092     result = mini_send(caller_ptr, dest, m_ptr);
25093     switching = FALSE;
25094     return(result);
25095   }

25097   /*===========================================================================*
25098    *                              lock_pick_proc                               *
25099    *===========================================================================*/
25100   PUBLIC void lock_pick_proc()
25101   {
25102   /* Safe gateway to pick_proc() for tasks. */
25103
25104     switching = TRUE;
25105     pick_proc();
25106     switching = FALSE;
25107   }

25109   /*===========================================================================*
25110    *                              lock_ready                                   *
25111    *===========================================================================*/
25112   PUBLIC void lock_ready(rp)
25113   struct proc *rp;                /* this process is now runnable */
25114   {
```

```
25115   /* Safe gateway to ready() for tasks. */
25116
25117     switching = TRUE;
25118     ready(rp);
25119     switching = FALSE;
25120   }

25122   /*===========================================================================*
25123    *                                lock_unready                               *
25124    *===========================================================================*/
25125   PUBLIC void lock_unready(rp)
25126   struct proc *rp;                /* this process is no longer runnable */
25127   {
25128   /* Safe gateway to unready() for tasks. */
25129
25130     switching = TRUE;
25131     unready(rp);
25132     switching = FALSE;
25133   }

25135   /*===========================================================================*
25136    *                                lock_sched                                 *
25137    *===========================================================================*/
25138   PUBLIC void lock_sched()
25139   {
25140   /* Safe gateway to sched() for tasks. */
25141
25142     switching = TRUE;
25143     sched();
25144     switching = FALSE;
25145   }

25147   /*===========================================================================*
25148    *                                unhold                                     *
25149    *===========================================================================*/
25150   PUBLIC void unhold()
25151   {
25152   /* Flush any held-up interrupts.  k_reenter must be 0.  held_head must not
25153    * be NIL_PROC.  Interrupts must be disabled.  They will be enabled but will
25154    * be disabled when this returns.
25155    */
25156
25157     register struct proc *rp;     /* current head of held queue */
25158
25159     if (switching) return;
25160     rp = held_head;
25161     do {
25162           if ( (held_head = rp->p_nextheld) == NIL_PROC) held_tail = NIL_PROC;
25163           rp->p_int_held = FALSE;
25164           unlock();               /* reduce latency; held queue may change! */
25165           interrupt(proc_number(rp));
25166           lock();                 /* protect the held queue again */
25167     }
25168     while ( (rp = held_head) != NIL_PROC);
25169   }

25171   #if (CHIP == M68000)
25172   /*===========================================================================*
25173    *                                cp_mess                                    *
25174    *===========================================================================*/
```

```
25175    PRIVATE void cp_mess(src, src_p, src_m, dst_p, dst_m)
25176    int src;                        /* sender process */
25177    register struct proc *src_p;    /* source proc entry */
25178    message *src_m;                 /* source message */
25179    register struct proc *dst_p;    /* destination proc entry */
25180    message *dst_m;                 /* destination buffer */
25181    {
25182    #if (SHADOWING == 0)
25183      /* convert virtual address to physical address */
25184      /* The caller has already checked if all addresses are within bounds */
25185
25186      src_m = (message *)((char *)src_m + (((phys_bytes)src_p->p_map[D].mem_phys
25187                                  - src_p->p_map[D].mem_vir) << CLICK_SHIFT));
25188      dst_m = (message *)((char *)dst_m + (((phys_bytes)dst_p->p_map[D].mem_phys
25189                                  - dst_p->p_map[D].mem_vir) << CLICK_SHIFT));
25190    #else
25191      register phys_bytes correction;
25192
25193      if (correction = src_p->p_shadow) {
25194            correction = (correction - src_p->p_map[D].mem_phys) << CLICK_SHIFT;
25195            src_m = (message *)((char *)src_m + correction);
25196      }
25197      if (correction = dst_p->p_shadow) {
25198            correction = (correction - dst_p->p_map[D].mem_phys) << CLICK_SHIFT;
25199            dst_m = (message *)((char *)dst_m + correction);
25200      }
25201    #endif
25202    #ifdef NEEDFSTRUCOPY
25203      phys_copy(src_m,dst_m,(phys_bytes) sizeof(message));
25204    #else
25205      *dst_m = *src_m;
25206    #endif
25207      dst_m->m_source = src;
25208    }
25209    #endif
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/protect.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

25300    /* This file contains code for initialization of protected mode, to initialize
25301     * code and data segment descriptors, and to initialize global descriptors
25302     * for local descriptors in the process table.
25303     */
25304
25305    #include "kernel.h"
25306    #include "proc.h"
25307    #include "protect.h"
25308
25309    #if _WORD_SIZE == 4
25310    #define INT_GATE_TYPE   (INT_286_GATE | DESC_386_BIT)
25311    #define TSS_TYPE        (AVL_286_TSS  | DESC_386_BIT)
25312    #else
25313    #define INT_GATE_TYPE   INT_286_GATE
25314    #define TSS_TYPE        AVL_286_TSS
```

```
25315   #endif
25316
25317   struct desctableptr_s {
25318     char limit[sizeof(u16_t)];
25319     char base[sizeof(u32_t)];             /* really u24_t + pad for 286 */
25320   };
25321
25322   struct gatedesc_s {
25323     u16_t offset_low;
25324     u16_t selector;
25325     u8_t pad;                     /* |000|XXXXX| ig & trpg, |XXXXXXXX| task g */
25326     u8_t p_dpl_type;              /* |P|DL|0|TYPE| */
25327   #if _WORD_SIZE == 4
25328     u16_t offset_high;
25329   #else
25330     u16_t reserved;
25331   #endif
25332   };
25333
25334   struct tss_s {
25335     reg_t backlink;
25336     reg_t sp0;                    /* stack pointer to use during interrupt */
25337     reg_t ss0;                    /*   "   segment  "  "    "        "     */
25338     reg_t sp1;
25339     reg_t ss1;
25340     reg_t sp2;
25341     reg_t ss2;
25342   #if _WORD_SIZE == 4
25343     reg_t cr3;
25344   #endif
25345     reg_t ip;
25346     reg_t flags;
25347     reg_t ax;
25348     reg_t cx;
25349     reg_t dx;
25350     reg_t bx;
25351     reg_t sp;
25352     reg_t bp;
25353     reg_t si;
25354     reg_t di;
25355     reg_t es;
25356     reg_t cs;
25357     reg_t ss;
25358     reg_t ds;
25359   #if _WORD_SIZE == 4
25360     reg_t fs;
25361     reg_t gs;
25362   #endif
25363     reg_t ldt;
25364   #if _WORD_SIZE == 4
25365     u16_t trap;
25366     u16_t iobase;
25367   /* u8_t iomap[0]; */
25368   #endif
25369   };
25370
25371   PUBLIC struct segdesc_s gdt[GDT_SIZE];
25372   PRIVATE struct gatedesc_s idt[IDT_SIZE];        /* zero-init so none present */
25373   PUBLIC struct tss_s tss;        /* zero init */
25374
```

```
25375   FORWARD _PROTOTYPE( void int_gate, (unsigned vec_nr, phys_bytes base,
25376                   unsigned dpl_type) );
25377   FORWARD _PROTOTYPE( void sdesc, (struct segdesc_s *segdp, phys_bytes base,
25378                   phys_bytes size) );
25379
25380   /*===========================================================================*
25381    *                               prot_init                                   *
25382    *===========================================================================*/
25383   PUBLIC void prot_init()
25384   {
25385   /* Set up tables for protected mode.
25386    * All GDT slots are allocated at compile time.
25387    */
25388
25389     phys_bytes code_bytes;
25390     phys_bytes data_bytes;
25391     struct gate_table_s *gtp;
25392     struct desctableptr_s *dtp;
25393     unsigned ldt_selector;
25394     register struct proc *rp;
25395
25396     static struct gate_table_s {
25397           _PROTOTYPE( void (*gate), (void) );
25398           unsigned char vec_nr;
25399           unsigned char privilege;
25400     }
25401     gate_table[] = {
25402           divide_error, DIVIDE_VECTOR, INTR_PRIVILEGE,
25403           single_step_exception, DEBUG_VECTOR, INTR_PRIVILEGE,
25404           nmi, NMI_VECTOR, INTR_PRIVILEGE,
25405           breakpoint_exception, BREAKPOINT_VECTOR, USER_PRIVILEGE,
25406           overflow, OVERFLOW_VECTOR, USER_PRIVILEGE,
25407           bounds_check, BOUNDS_VECTOR, INTR_PRIVILEGE,
25408           inval_opcode, INVAL_OP_VECTOR, INTR_PRIVILEGE,
25409           copr_not_available, COPROC_NOT_VECTOR, INTR_PRIVILEGE,
25410           double_fault, DOUBLE_FAULT_VECTOR, INTR_PRIVILEGE,
25411           copr_seg_overrun, COPROC_SEG_VECTOR, INTR_PRIVILEGE,
25412           inval_tss, INVAL_TSS_VECTOR, INTR_PRIVILEGE,
25413           segment_not_present, SEG_NOT_VECTOR, INTR_PRIVILEGE,
25414           stack_exception, STACK_FAULT_VECTOR, INTR_PRIVILEGE,
25415           general_protection, PROTECTION_VECTOR, INTR_PRIVILEGE,
25416   #if _WORD_SIZE == 4
25417           page_fault, PAGE_FAULT_VECTOR, INTR_PRIVILEGE,
25418           copr_error, COPROC_ERR_VECTOR, INTR_PRIVILEGE,
25419   #endif
25420           { hwint00, VECTOR( 0), INTR_PRIVILEGE },
25421           { hwint01, VECTOR( 1), INTR_PRIVILEGE },
25422           { hwint02, VECTOR( 2), INTR_PRIVILEGE },
25423           { hwint03, VECTOR( 3), INTR_PRIVILEGE },
25424           { hwint04, VECTOR( 4), INTR_PRIVILEGE },
25425           { hwint05, VECTOR( 5), INTR_PRIVILEGE },
25426           { hwint06, VECTOR( 6), INTR_PRIVILEGE },
25427           { hwint07, VECTOR( 7), INTR_PRIVILEGE },
25428           { hwint08, VECTOR( 8), INTR_PRIVILEGE },
25429           { hwint09, VECTOR( 9), INTR_PRIVILEGE },
25430           { hwint10, VECTOR(10), INTR_PRIVILEGE },
25431           { hwint11, VECTOR(11), INTR_PRIVILEGE },
25432           { hwint12, VECTOR(12), INTR_PRIVILEGE },
25433           { hwint13, VECTOR(13), INTR_PRIVILEGE },
25434           { hwint14, VECTOR(14), INTR_PRIVILEGE },
```

```
25435           { hwint15, VECTOR(15), INTR_PRIVILEGE },
25436       };
25437
25438       /* This is called early and can't use tables set up by main(). */
25439       data_bytes = (phys_bytes) sizes[1] << CLICK_SHIFT;
25440       if (sizes[0] == 0)
25441           code_bytes = data_bytes;        /* common I&D */
25442       else
25443           code_bytes = (phys_bytes) sizes[0] << CLICK_SHIFT;
25444
25445       /* Build gdt and idt pointers in GDT where the BIOS expects them. */
25446       dtp= (struct desctableptr_s *) &gdt[GDT_INDEX];
25447       * (u16_t *) dtp->limit = (sizeof gdt) - 1;
25448       * (u32_t *) dtp->base = vir2phys(gdt);
25449
25450       dtp= (struct desctableptr_s *) &gdt[IDT_INDEX];
25451       * (u16_t *) dtp->limit = (sizeof idt) - 1;
25452       * (u32_t *) dtp->base = vir2phys(idt);
25453
25454       /* Build segment descriptors for tasks and interrupt handlers. */
25455       init_codeseg(&gdt[CS_INDEX], code_base, code_bytes, INTR_PRIVILEGE);
25456       init_dataseg(&gdt[DS_INDEX], data_base, data_bytes, INTR_PRIVILEGE);
25457       init_dataseg(&gdt[ES_INDEX], 0L, 0L, TASK_PRIVILEGE);
25458
25459       /* Build scratch descriptors for functions in klib88. */
25460       init_dataseg(&gdt[DS_286_INDEX], (phys_bytes) 0,
25461                   (phys_bytes) MAX_286_SEG_SIZE, TASK_PRIVILEGE);
25462       init_dataseg(&gdt[ES_286_INDEX], (phys_bytes) 0,
25463                   (phys_bytes) MAX_286_SEG_SIZE, TASK_PRIVILEGE);
25464
25465       /* Build local descriptors in GDT for LDT's in process table.
25466        * The LDT's are allocated at compile time in the process table, and
25467        * initialized whenever a process' map is initialized or changed.
25468        */
25469       for (rp = BEG_PROC_ADDR, ldt_selector = FIRST_LDT_INDEX * DESC_SIZE;
25470            rp < END_PROC_ADDR; ++rp, ldt_selector += DESC_SIZE) {
25471             init_dataseg(&gdt[ldt_selector / DESC_SIZE], vir2phys(rp->p_ldt),
25472                          (phys_bytes) sizeof rp->p_ldt, INTR_PRIVILEGE);
25473             gdt[ldt_selector / DESC_SIZE].access = PRESENT | LDT;
25474             rp->p_ldt_sel = ldt_selector;
25475       }
25476
25477       /* Build main TSS.
25478        * This is used only to record the stack pointer to be used after an
25479        * interrupt.
25480        * The pointer is set up so that an interrupt automatically saves the
25481        * current process's registers ip:cs:f:sp:ss in the correct slots in the
25482        * process table.
25483        */
25484       tss.ss0 = DS_SELECTOR;
25485       init_dataseg(&gdt[TSS_INDEX], vir2phys(&tss), (phys_bytes) sizeof tss,
25486                                                               INTR_PRIVILEGE);
25487       gdt[TSS_INDEX].access = PRESENT | (INTR_PRIVILEGE << DPL_SHIFT) | TSS_TYPE;
25488
25489       /* Build descriptors for interrupt gates in IDT. */
25490       for (gtp = &gate_table[0];
25491            gtp < &gate_table[sizeof gate_table / sizeof gate_table[0]]; ++gtp) {
25492             int_gate(gtp->vec_nr, (phys_bytes) (vir_bytes) gtp->gate,
25493                      PRESENT | INT_GATE_TYPE | (gtp->privilege << DPL_SHIFT));
25494       }
```

```
25495     int_gate(SYS_VECTOR, (phys_bytes) (vir_bytes) p_s_call,
25496              PRESENT | (USER_PRIVILEGE << DPL_SHIFT) | INT_GATE_TYPE);
25497     int_gate(LEVEL0_VECTOR, (phys_bytes) (vir_bytes) level0_call,
25498              PRESENT | (TASK_PRIVILEGE << DPL_SHIFT) | INT_GATE_TYPE);
25499
25500   #if _WORD_SIZE == 4
25501     /* Complete building of main TSS. */
25502     tss.iobase = sizeof tss;      /* empty i/o permissions map */
25503
25504     /* Complete building of interrupt gates. */
25505     int_gate(SYS386_VECTOR, (phys_bytes) (vir_bytes) s_call,
25506              PRESENT | (USER_PRIVILEGE << DPL_SHIFT) | INT_GATE_TYPE);
25507   #endif
25508   }

25510   /*===========================================================================*
25511    *                              init_codeseg                                 *
25512    *===========================================================================*/
25513   PUBLIC void init_codeseg(segdp, base, size, privilege)
25514   register struct segdesc_s *segdp;
25515   phys_bytes base;
25516   phys_bytes size;
25517   int privilege;
25518   {
25519   /* Build descriptor for a code segment. */
25520
25521     sdesc(segdp, base, size);
25522     segdp->access = (privilege << DPL_SHIFT)
25523                   | (PRESENT | SEGMENT | EXECUTABLE | READABLE);
25524                   /* CONFORMING = 0, ACCESSED = 0 */
25525   }

25527   /*===========================================================================*
25528    *                              init_dataseg                                 *
25529    *===========================================================================*/
25530   PUBLIC void init_dataseg(segdp, base, size, privilege)
25531   register struct segdesc_s *segdp;
25532   phys_bytes base;
25533   phys_bytes size;
25534   int privilege;
25535   {
25536   /* Build descriptor for a data segment. */
25537
25538     sdesc(segdp, base, size);
25539     segdp->access = (privilege << DPL_SHIFT) | (PRESENT | SEGMENT | WRITEABLE);
25540                   /* EXECUTABLE = 0, EXPAND_DOWN = 0, ACCESSED = 0 */
25541   }

25543   /*===========================================================================*
25544    *                              sdesc                                        *
25545    *===========================================================================*/
25546   PRIVATE void sdesc(segdp, base, size)
25547   register struct segdesc_s *segdp;
25548   phys_bytes base;
25549   phys_bytes size;
25550   {
25551   /* Fill in the size fields (base, limit and granularity) of a descriptor. */
25552
25553     segdp->base_low = base;
25554     segdp->base_middle = base >> BASE_MIDDLE_SHIFT;
```

```
25555
25556   #if _WORD_SIZE == 4
25557     segdp->base_high = base >> BASE_HIGH_SHIFT;
25558     --size;                       /* convert to a limit, 0 size means 4G */
25559     if (size > BYTE_GRAN_MAX) {
25560           segdp->limit_low = size >> PAGE_GRAN_SHIFT;
25561           segdp->granularity = GRANULAR | (size >>
25562                                       (PAGE_GRAN_SHIFT + GRANULARITY_SHIFT));
25563     } else {
25564           segdp->limit_low = size;
25565           segdp->granularity = size >> GRANULARITY_SHIFT;
25566     }
25567     segdp->granularity |= DEFAULT;        /* means BIG for data seg */
25568   #else
25569     segdp->limit_low = size - 1;
25570   #endif
25571   }

25573   /*==========================================================================*
25574    *                              seg2phys                                    *
25575    *==========================================================================*/
25576   PUBLIC phys_bytes seg2phys(seg)
25577   U16_t seg;
25578   {
25579   /* Return the base address of a segment, with seg being either a 8086 segment
25580    * register, or a 286/386 segment selector.
25581    */
25582     phys_bytes base;
25583     struct segdesc_s *segdp;
25584
25585     if (!protected_mode) {
25586           base = hclick_to_physb(seg);
25587     } else {
25588           segdp = &gdt[seg >> 3];
25589           base = segdp->base_low | ((u32_t) segdp->base_middle << 16);
25590   #if _WORD_SIZE == 4
25591           base |= ((u32_t) segdp->base_high << 24);
25592   #endif
25593     }
25594     return base;
25595   }

25597   /*==========================================================================*
25598    *                              int_gate                                    *
25599    *==========================================================================*/
25600   PRIVATE void int_gate(vec_nr, base, dpl_type)
25601   unsigned vec_nr;
25602   phys_bytes base;
25603   unsigned dpl_type;
25604   {
25605   /* Build descriptor for an interrupt gate. */
25606
25607     register struct gatedesc_s *idp;
25608
25609     idp = &idt[vec_nr];
25610     idp->offset_low = base;
25611     idp->selector = CS_SELECTOR;
25612     idp->p_dpl_type = dpl_type;
25613   #if _WORD_SIZE == 4
25614     idp->offset_high = base >> OFFSET_HIGH_SHIFT;
```

```
25615    #endif
25616    }

25618    /*===========================================================================*
25619     *                              enable_iop                                   *
25620     *===========================================================================*/
25621    PUBLIC void enable_iop(pp)
25622    struct proc *pp;
25623    {
25624    /* Allow a user process to use I/O instructions.  Change the I/O Permission
25625     * Level bits in the psw. These specify least-privileged Current Permission
25626     * Level allowed to execute I/O instructions. Users and servers have CPL 3.
25627     * You can't have less privilege than that. Kernel has CPL 0, tasks CPL 1.
25628     */
25629      pp->p_reg.psw |= 0x3000;
25630    }
```

++
src/kernel/pty.c
++

```
25700    /*      pty.c - pseudo terminal driver                  Author: Kees J. Bot
25701     *                                                              30 Dec 1995
25702     * PTYs can be seen as a bidirectional pipe with TTY
25703     * input and output processing.  For example a simple rlogin session:
25704     *
25705     *      keyboard -> rlogin -> in.rld -> /dev/ptypX -> /dev/ttypX -> shell
25706     *      shell -> /dev/ttypX -> /dev/ptypX -> in.rld -> rlogin -> screen
25707     *
25708     * This file takes care of copying data between the tty/pty device pairs and
25709     * the open/read/write/close calls on the pty devices.  The TTY task takes
25710     * care of the input and output processing (interrupt, backspace, raw I/O,
25711     * etc.) using the pty_read() and pty_write() functions as the "keyboard" and
25712     * "screen" functions of the ttypX devices.
25713     * Be careful when reading this code, the terms "reading" and "writing" are
25714     * used both for the tty and the pty end of the pseudo tty.  Writes to one
25715     * end are to be read at the other end and vice-versa.
25716     */
25717
25718    #include "kernel.h"
25719    #include <termios.h>
25720    #include <signal.h>
25721    #include <minix/com.h>
25722    #include <minix/callnr.h>
25723    #include "tty.h"
25724    #include "proc.h"
25725
25726    #if NR_PTYS > 0
25727
25728    /* PTY bookkeeping structure, one per pty/tty pair. */
25729    typedef struct pty {
25730      tty_t         *tty;           /* associated TTY structure */
25731      char          state;          /* flags: busy, closed, ... */
25732
25733      /* Read call on /dev/ptypX. */
25734      char          rdrepcode;      /* reply code, TASK_REPLY or REVIVE */
```

```
25735     char          rdcaller;       /* process making the call (usually FS) */
25736     char          rdproc;         /* process that wants to read from the pty */
25737     vir_bytes     rdvir;          /* virtual address in readers address space */
25738     int           rdleft;         /* # bytes yet to be read */
25739     int           rdcum;          /* # bytes written so far */
25740
25741     /* Write call to /dev/ptypX. */
25742     char          wrrepcode;      /* reply code, TASK_REPLY or REVIVE */
25743     char          wrcaller;       /* process making the call (usually FS) */
25744     char          wrproc;         /* process that wants to write to the pty */
25745     vir_bytes     wrvir;          /* virtual address in writers address space */
25746     int           wrleft;         /* # bytes yet to be written */
25747     int           wrcum;          /* # bytes written so far */
25748
25749     /* Output buffer. */
25750     int           ocount;         /* # characters in the buffer */
25751     char          *ohead, *otail; /* head and tail of the circular buffer */
25752     char          obuf[128];      /* buffer for bytes going to the pty reader */
25753   } pty_t;
25754
25755   #define PTY_ACTIVE      0x01    /* pty is open/active */
25756   #define TTY_CLOSED      0x02    /* tty side has closed down */
25757   #define PTY_CLOSED      0x04    /* pty side has closed down */
25758
25759   PRIVATE pty_t pty_table[NR_PTYS];       /* PTY bookkeeping */
25760
25761
25762   FORWARD _PROTOTYPE( void pty_write, (tty_t *tp)                       );
25763   FORWARD _PROTOTYPE( void pty_echo, (tty_t *tp, int c)                 );
25764   FORWARD _PROTOTYPE( void pty_start, (pty_t *pp)                       );
25765   FORWARD _PROTOTYPE( void pty_finish, (pty_t *pp)                      );
25766   FORWARD _PROTOTYPE( void pty_read, (tty_t *tp)                        );
25767   FORWARD _PROTOTYPE( void pty_close, (tty_t *tp)                       );
25768   FORWARD _PROTOTYPE( void pty_icancel, (tty_t *tp)                     );
25769   FORWARD _PROTOTYPE( void pty_ocancel, (tty_t *tp)                     );
25770
25771
25772   /*==========================================================================*
25773    *                              do_pty                                      *
25774    *==========================================================================*/
25775   PUBLIC void do_pty(tp, m_ptr)
25776   tty_t *tp;
25777   message *m_ptr;
25778   {
25779   /* Perform an open/close/read/write call on a /dev/ptypX device. */
25780     pty_t *pp = tp->tty_priv;
25781     int r;
25782
25783     switch (m_ptr->m_type) {
25784       case DEV_READ:
25785           /* Check, store information on the reader, do I/O. */
25786           if (pp->state & TTY_CLOSED) {
25787                   r = 0;
25788                   break;
25789           }
25790           if (pp->rdleft != 0) {
25791                   r = EIO;
25792                   break;
25793           }
25794           if (m_ptr->COUNT <= 0) {
```

```
25795                   r = EINVAL;
25796                   break;
25797           }
25798           if (numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
25799                                                   m_ptr->COUNT) == 0) {
25800                   r = EFAULT;
25801                   break;
25802           }
25803           pp->rdrepcode = TASK_REPLY;
25804           pp->rdcaller = m_ptr->m_source;
25805           pp->rdproc = m_ptr->PROC_NR;
25806           pp->rdvir = (vir_bytes) m_ptr->ADDRESS;
25807           pp->rdleft = m_ptr->COUNT;
25808           pty_start(pp);
25809           handle_events(tp);
25810           if (pp->rdleft == 0) return;                    /* already done */
25811
25812           if (m_ptr->TTY_FLAGS & O_NONBLOCK) {
25813                   r = EAGAIN;                             /* don't suspend */
25814                   pp->rdleft = pp->rdcum = 0;
25815           } else {
25816                   r = SUSPEND;                            /* do suspend */
25817                   pp->rdrepcode = REVIVE;
25818           }
25819           break;
25820
25821       case DEV_WRITE:
25822           /* Check, store information on the writer, do I/O. */
25823           if (pp->state & TTY_CLOSED) {
25824                   r = EIO;
25825                   break;
25826           }
25827           if (pp->wrleft != 0) {
25828                   r = EIO;
25829                   break;
25830           }
25831           if (m_ptr->COUNT <= 0) {
25832                   r = EINVAL;
25833                   break;
25834           }
25835           if (numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
25836                                                   m_ptr->COUNT) == 0) {
25837                   r = EFAULT;
25838                   break;
25839           }
25840           pp->wrrepcode = TASK_REPLY;
25841           pp->wrcaller = m_ptr->m_source;
25842           pp->wrproc = m_ptr->PROC_NR;
25843           pp->wrvir = (vir_bytes) m_ptr->ADDRESS;
25844           pp->wrleft = m_ptr->COUNT;
25845           handle_events(tp);
25846           if (pp->wrleft == 0) return;                    /* already done */
25847
25848           if (m_ptr->TTY_FLAGS & O_NONBLOCK) {            /* don't suspend */
25849                   r = pp->wrcum > 0 ? pp->wrcum : EAGAIN;
25850                   pp->wrleft = pp->wrcum = 0;
25851           } else {
25852                   pp->wrrepcode = REVIVE;                 /* do suspend */
25853                   r = SUSPEND;
25854           }
```

```
25855           break;
25856
25857       case DEV_IOCTL:
25858           /* No ioctl's allowed on the pty side. */
25859           r = ENOTTY;
25860           break;
25861
25862       case DEV_OPEN:
25863           r = pp->state != 0 ? EIO : OK;
25864           pp->state |= PTY_ACTIVE;
25865           break;
25866
25867       case DEV_CLOSE:
25868           r = OK;
25869           if (pp->state & TTY_CLOSED) {
25870                   pp->state = 0;
25871           } else {
25872                   pp->state |= PTY_CLOSED;
25873                   sigchar(tp, SIGHUP);
25874           }
25875           break;
25876
25877       case CANCEL:
25878           if (m_ptr->PROC_NR == pp->rdproc) {
25879                   /* Cancel a read from a PTY. */
25880                   pp->rdleft = pp->rdcum = 0;
25881           }
25882           if (m_ptr->PROC_NR == pp->wrproc) {
25883                   /* Cancel a write to a PTY. */
25884                   pp->wrleft = pp->wrcum = 0;
25885           }
25886           r = EINTR;
25887           break;
25888
25889       default:
25890           r = EINVAL;
25891     }
25892     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
25893   }


25896   /*===========================================================================*
25897    *                                pty_write                                  *
25898    *===========================================================================*/
25899   PRIVATE void pty_write(tp)
25900   tty_t *tp;
25901   {
25902   /* (*dev_write)() routine for PTYs.  Transfer bytes from the writer on
25903    * /dev/ttypX to the output buffer.
25904    */
25905     pty_t *pp = tp->tty_priv;
25906     int count, ocount;
25907     phys_bytes user_phys;
25908
25909     /* PTY closed down? */
25910     if (pp->state & PTY_CLOSED) {
25911           if (tp->tty_outleft > 0) {
25912                   tty_reply(tp->tty_outrepcode, tp->tty_outcaller,
25913                                                           tp->tty_outproc, EIO);
25914                   tp->tty_outleft = tp->tty_outcum = 0;
```

```
25915           }
25916           return;
25917       }
25918
25919       /* While there is something to do. */
25920       for (;;) {
25921           ocount = buflen(pp->obuf) - pp->ocount;
25922           count = bufend(pp->obuf) - pp->ohead;
25923           if (count > ocount) count = ocount;
25924           if (count > tp->tty_outleft) count = tp->tty_outleft;
25925           if (count == 0 || tp->tty_inhibited) break;
25926
25927           /* Copy from user space to the PTY output buffer. */
25928           user_phys = proc_vir2phys(proc_addr(tp->tty_outproc), tp->tty_out_vir);
25929           phys_copy(user_phys, vir2phys(pp->ohead), (phys_bytes) count);
25930
25931           /* Perform output processing on the output buffer. */
25932           out_process(tp, pp->obuf, pp->ohead, bufend(pp->obuf), &count, &ocount);
25933           if (count == 0) break;
25934
25935           /* Assume echoing messed up by output. */
25936           tp->tty_reprint = TRUE;
25937
25938           /* Bookkeeping. */
25939           pp->ocount += ocount;
25940           if ((pp->ohead += ocount) >= bufend(pp->obuf))
25941                   pp->ohead -= buflen(pp->obuf);
25942           pty_start(pp);
25943           tp->tty_out_vir += count;
25944           tp->tty_outcum += count;
25945           if ((tp->tty_outleft -= count) == 0) {
25946                   /* Output is finished, reply to the writer. */
25947                   tty_reply(tp->tty_outrepcode, tp->tty_outcaller,
25948                                           tp->tty_outproc, tp->tty_outcum);
25949                   tp->tty_outcum = 0;
25950           }
25951       }
25952       pty_finish(pp);
25953   }


25956   /*==========================================================================*
25957    *                              pty_echo                                    *
25958    *==========================================================================*/
25959   PRIVATE void pty_echo(tp, c)
25960   tty_t *tp;
25961   int c;
25962   {
25963   /* Echo one character.  (Like pty_write, but only one character, optionally.) */
25964
25965       pty_t *pp = tp->tty_priv;
25966       int count, ocount;
25967
25968       ocount = buflen(pp->obuf) - pp->ocount;
25969       if (ocount == 0) return;              /* output buffer full */
25970       count = 1;
25971       *pp->ohead = c;                       /* add one character */
25972
25973       out_process(tp, pp->obuf, pp->ohead, bufend(pp->obuf), &count, &ocount);
25974       if (count == 0) return;
```

```
25975
25976     pp->ocount += ocount;
25977     if ((pp->ohead += ocount) >= bufend(pp->obuf)) pp->ohead -= buflen(pp->obuf);
25978     pty_start(pp);
25979   }


25982   /*===========================================================================*
25983    *                              pty_start                                    *
25984    *===========================================================================*/
25985   PRIVATE void pty_start(pp)
25986   pty_t *pp;
25987   {
25988   /* Transfer bytes written to the output buffer to the PTY reader. */
25989     int count;
25990     phys_bytes user_phys;
25991
25992     /* While there are things to do. */
25993     for (;;) {
25994           count = bufend(pp->obuf) - pp->otail;
25995           if (count > pp->ocount) count = pp->ocount;
25996           if (count > pp->rdleft) count = pp->rdleft;
25997           if (count == 0) break;
25998
25999           /* Copy from the output buffer to the readers address space. */
26000           user_phys = proc_vir2phys(proc_addr(pp->rdproc), pp->rdvir);
26001           phys_copy(vir2phys(pp->otail), user_phys, (phys_bytes) count);
26002
26003           /* Bookkeeping. */
26004           pp->ocount -= count;
26005           if ((pp->otail += count) == bufend(pp->obuf)) pp->otail = pp->obuf;
26006           pp->rdvir += count;
26007           pp->rdcum += count;
26008           pp->rdleft -= count;
26009     }
26010   }


26013   /*===========================================================================*
26014    *                              pty_finish                                   *
26015    *===========================================================================*/
26016   PRIVATE void pty_finish(pp)
26017   pty_t *pp;
26018   {
26019   /* Finish the read request of a PTY reader if there is at least one byte
26020    * transferred.
26021    */
26022
26023     if (pp->rdcum > 0) {
26024           tty_reply(pp->rdrepcode, pp->rdcaller, pp->rdproc, pp->rdcum);
26025           pp->rdleft = pp->rdcum = 0;
26026     }
26027   }


26030   /*===========================================================================*
26031    *                              pty_read                                     *
26032    *===========================================================================*/
26033   PRIVATE void pty_read(tp)
26034   tty_t *tp;
```

```
26035   {
26036   /* Offer bytes from the PTY writer for input on the TTY.  (Do it one byte at
26037    * a time, 99% of the writes will be for one byte, so no sense in being smart.)
26038    */
26039     pty_t *pp = tp->tty_priv;
26040     phys_bytes user_phys;
26041     char c;
26042
26043     if (pp->state & PTY_CLOSED) {
26044           if (tp->tty_inleft > 0) {
26045                   tty_reply(tp->tty_inrepcode, tp->tty_incaller, tp->tty_inproc,
26046                                                                   tp->tty_incum);
26047                   tp->tty_inleft = tp->tty_incum = 0;
26048           }
26049           return;
26050     }
26051
26052     while (pp->wrleft > 0) {
26053           /* Transfer one character to 'c'. */
26054           user_phys = proc_vir2phys(proc_addr(pp->wrproc), pp->wrvir);
26055           phys_copy(user_phys, vir2phys(&c), 1L);
26056
26057           /* Input processing. */
26058           if (in_process(tp, &c, 1) == 0) break;
26059
26060           /* PTY writer bookkeeping. */
26061           pp->wrvir++;
26062           pp->wrcum++;
26063           if (--pp->wrleft == 0) {
26064                   tty_reply(pp->wrrepcode, pp->wrcaller, pp->wrproc, pp->wrcum);
26065                   pp->wrcum = 0;
26066           }
26067     }
26068   }


26071   /*===========================================================================*
26072    *                                pty_close                                  *
26073    *===========================================================================*/
26074   PRIVATE void pty_close(tp)
26075   tty_t *tp;
26076   {
26077   /* The tty side has closed, so shut down the pty side. */
26078     pty_t *pp = tp->tty_priv;
26079
26080     if (!(pp->state & PTY_ACTIVE)) return;
26081
26082     if (pp->rdleft > 0) {
26083           tty_reply(pp->rdrepcode, pp->rdcaller, pp->rdproc, 0);
26084           pp->rdleft = pp->rdcum = 0;
26085     }
26086
26087     if (pp->wrleft > 0) {
26088           tty_reply(pp->wrrepcode, pp->wrcaller, pp->wrproc, EIO);
26089           pp->wrleft = pp->wrcum = 0;
26090     }
26091
26092     if (pp->state & PTY_CLOSED) pp->state = 0; else pp->state |= TTY_CLOSED;
26093   }
```

```
26096   /*==========================================================================*
26097    *                              pty_icancel                                 *
26098    *==========================================================================*/
26099   PRIVATE void pty_icancel(tp)
26100   tty_t *tp;
26101   {
26102   /* Discard waiting input. */
26103     pty_t *pp = tp->tty_priv;
26104
26105     if (pp->wrleft > 0) {
26106           tty_reply(pp->wrrepcode, pp->wrcaller, pp->wrproc,
26107                                                  pp->wrcum + pp->wrleft);
26108           pp->wrleft = pp->wrcum = 0;
26109     }
26110   }


26113   /*==========================================================================*
26114    *                              pty_ocancel                                 *
26115    *==========================================================================*/
26116   PRIVATE void pty_ocancel(tp)
26117   tty_t *tp;
26118   {
26119   /* Drain the output buffer. */
26120     pty_t *pp = tp->tty_priv;
26121
26122     pp->ocount = 0;
26123     pp->otail = pp->ohead;
26124   }


26127   /*==========================================================================*
26128    *                              pty_init                                    *
26129    *==========================================================================*/
26130   PUBLIC void pty_init(tp)
26131   tty_t *tp;
26132   {
26133     pty_t *pp;
26134     int line;
26135
26136     /* Associate PTY and TTY structures. */
26137     line = tp - &tty_table[NR_CONS + NR_RS_LINES];
26138     pp = tp->tty_priv = &pty_table[line];
26139     pp->tty = tp;
26140
26141     /* Set up output queue. */
26142     pp->ohead = pp->otail = pp->obuf;
26143
26144     /* Fill in TTY function hooks. */
26145     tp->tty_devread = pty_read;
26146     tp->tty_devwrite = pty_write;
26147     tp->tty_echo = pty_echo;
26148     tp->tty_icancel = pty_icancel;
26149     tp->tty_ocancel = pty_ocancel;
26150     tp->tty_close = pty_close;
26151   }
26152   #endif /* NR_PTYS > 0 */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/rs232.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

26200    /*=============================================================================*
26201     *              rs232.c - serial driver for 8250 and 16450 UARTs               *
26202     *              Added support for Atari ST M68901 and YM-2149   --kub          *
26203     *=============================================================================*/
26204
26205    #include "kernel.h"
26206    #include <termios.h>
26207    #include <signal.h>
26208    #include "tty.h"
26209    #include "proc.h"
26210
26211    #if NR_RS_LINES > 0
26212
26213    #if (MACHINE != IBM_PC) && (MACHINE != ATARI)
26214    #error                          /* rs232.c only supports PC and Atari ST */
26215    #endif
26216
26217    #if (MACHINE == ATARI)
26218    #include "staddr.h"
26219    #include "stsound.h"
26220    #include "stmfp.h"
26221    #if (NR_RS_LINES > 1)
26222    #error                          /* Only one physical RS232 line available */
26223    #endif
26224    #endif
26225
26226    #if (MACHINE == IBM_PC)         /* PC/AT 8250/16450 chip combination */
26227
26228    /* 8250 constants. */
26229    #define UART_FREQ         115200L       /* timer frequency */
26230
26231    /* Interrupt enable bits. */
26232    #define IE_RECEIVER_READY       1
26233    #define IE_TRANSMITTER_READY    2
26234    #define IE_LINE_STATUS_CHANGE   4
26235    #define IE_MODEM_STATUS_CHANGE  8
26236
26237    /* Interrupt status bits. */
26238    #define IS_MODEM_STATUS_CHANGE  0
26239    #define IS_TRANSMITTER_READY    2
26240    #define IS_RECEIVER_READY       4
26241    #define IS_LINE_STATUS_CHANGE   6
26242
26243    /* Line control bits. */
26244    #define LC_2STOP_BITS        0x04
26245    #define LC_PARITY            0x08
26246    #define LC_PAREVEN           0x10
26247    #define LC_BREAK             0x40
26248    #define LC_ADDRESS_DIVISOR   0x80
26249
26250    /* Line status bits. */
26251    #define LS_OVERRUN_ERR          2
26252    #define LS_PARITY_ERR           4
26253    #define LS_FRAMING_ERR          8
26254    #define LS_BREAK_INTERRUPT   0x10
```

```
26255   #define LS_TRANSMITTER_READY 0x20
26256
26257   /* Modem control bits. */
26258   #define MC_DTR                  1
26259   #define MC_RTS                  2
26260   #define MC_OUT2                 8       /* required for PC & AT interrupts */
26261
26262   /* Modem status bits. */
26263   #define MS_CTS              0x10
26264   #define MS_RLSD             0x80        /* Received Line Signal Detect */
26265   #define MS_DRLSD            0x08        /* RLSD Delta */
26266
26267   #else /* MACHINE == ATARI */            /* Atari ST 68901 USART */
26268
26269   /* Most of the USART constants are already defined in stmfp.h . The local
26270    * definitions made here are for keeping C code changes smaller.   --kub
26271    */
26272
26273   #define UART_FREQ          19200L       /* timer frequency */
26274
26275   /* Line status bits. */
26276   #define LS_OVERRUN_ERR      R_OE
26277   #define LS_PARITY_ERR       R_PE
26278   #define LS_FRAMING_ERR      R_FE
26279   #define LS_BREAK_INTERRUPT  R_BREAK
26280
26281   /* Modem status bits. */
26282   #define MS_CTS              IO_SCTS     /* 0x04 */
26283
26284   #endif /* MACHINE == ATARI */
26285
26286   #define DATA_BITS_SHIFT         8       /* amount data bits shifted in mode */
26287   #define DEF_BAUD             1200       /* default baud rate */
26288
26289   #define RS_IBUFSIZE          1024       /* RS232 input buffer size */
26290   #define RS_OBUFSIZE          1024       /* RS232 output buffer size */
26291
26292   /* Input buffer watermarks.
26293    * The external device is asked to stop sending when the buffer
26294    * exactly reaches high water, or when TTY requests it.  Sending restarts
26295    * when the input buffer empties below the low watermark.
26296    */
26297   #define RS_ILOWWATER    (1 * RS_IBUFSIZE / 4)
26298   #define RS_IHIGHWATER   (3 * RS_IBUFSIZE / 4)
26299
26300   /* Output buffer low watermark.
26301    * TTY is notified when the output buffer empties below the low watermark, so
26302    * it may continue filling the buffer if doing a large write.
26303    */
26304   #define RS_OLOWWATER    (1 * RS_OBUFSIZE / 4)
26305
26306   #if (MACHINE == IBM_PC)
26307
26308   /* Macros to handle flow control.
26309    * Interrupts must be off when they are used.
26310    * Time is critical - already the function call for out_byte() is annoying.
26311    * If out_byte() can be done in-line, tests to avoid it can be dropped.
26312    * istart() tells external device we are ready by raising RTS.
26313    * istop() tells external device we are not ready by dropping RTS.
26314    * DTR is kept high all the time (it probably should be raised by open and
```

```
26315     * dropped by close of the device).
26316     * OUT2 is also kept high all the time.
26317     */
26318    #define istart(rs) \
26319            (out_byte((rs)->modem_ctl_port, MC_OUT2 | MC_RTS | MC_DTR), \
26320                    (rs)->idevready = TRUE)
26321    #define istop(rs) \
26322            (out_byte((rs)->modem_ctl_port, MC_OUT2 | MC_DTR), \
26323                    (rs)->idevready = FALSE)
26324
26325    /* Macro to tell if device is ready.  The rs->cts field is set to MS_CTS if
26326     * CLOCAL is in effect for a line without a CTS wire.
26327     */
26328    #define devready(rs) ((in_byte(rs->modem_status_port) | rs->cts) & MS_CTS)
26329
26330    /* Macro to tell if transmitter is ready. */
26331    #define txready(rs) (in_byte(rs->line_status_port) & LS_TRANSMITTER_READY)
26332
26333    /* Macro to tell if carrier has dropped.
26334     * The RS232 Carrier Detect (CD) line is usually connected to the 8250
26335     * Received Line Signal Detect pin, reflected by bit MS_RLSD in the Modem
26336     * Status Register.  The MS_DRLSD bit tells if MS_RLSD has just changed state.
26337     * So if MS_DRLSD is set and MS_RLSD cleared, we know that carrier has just
26338     * dropped.
26339     */
26340    #define devhup(rs)      \
26341            (in_byte(rs->modem_status_port) & (MS_RLSD|MS_DRLSD) == MS_DRLSD)
26342
26343    #else /* MACHINE == ATARI */
26344
26345    /* Macros to handle flow control.
26346     * Time is critical - already the function call for lock()/restore() is
26347     * annoying.
26348     * istart() tells external device we are ready by raising RTS.
26349     * istop() tells external device we are not ready by dropping RTS.
26350     * DTR is kept high all the time (it probably should be raised by open and
26351     * dropped by close of the device). NOTE: The modem lines are active low.
26352     */
26353    #define set_porta(msk,val) { register int s = lock();           \
26354                                 SOUND->sd_selr = YM_IOA;           \
26355                                 SOUND->sd_wdat =                   \
26356                                    SOUND->sd_rdat & (msk) | (val); \
26357                                 restore(s);        }
26358    #define istart(rs)         { set_porta( ~(PA_SRTS|PA_SDTR),0 ); \
26359                                 (rs)->idevready = TRUE;    }
26360    #define istop(rs)          { set_porta( ~PA_SDTR, PA_SRTS );    \
26361                                 (rs)->idevready = FALSE;   }
26362
26363    /* Macro to tell if device is ready.  The rs->cts field is set to MS_CTS if
26364     * CLOCAL is in effect for a line without a CTS wire.
26365     */
26366    #define devready(rs)         ((~MFP->mf_gpip | rs->cts) & MS_CTS)
26367
26368    /* Transmitter ready test */
26369    #define txready(rs)          (MFP->mf_tsr & (T_EMPTY | T_UE))
26370
26371    #endif /* MACHINE == ATARI */
26372
26373    /* Types. */
26374    typedef unsigned char bool_t;   /* boolean */
```

```
26375
26376   /* RS232 device structure, one per device. */
26377   typedef struct rs232 {
26378     tty_t *tty;                   /* associated TTY structure */
26379
26380     int icount;                   /* number of bytes in the input buffer */
26381     char *ihead;                  /* next free spot in input buffer */
26382     char *itail;                  /* first byte to give to TTY */
26383     bool_t idevready;             /* nonzero if we are ready to receive (RTS) */
26384     char cts;                     /* normally 0, but MS_CTS if CLOCAL is set */
26385
26386     unsigned char ostate;         /* combination of flags: */
26387   #define ODONE          1        /* output completed (< output enable bits) */
26388   #define ORAW           2        /* raw mode for xoff disable (< enab. bits) */
26389   #define OWAKEUP        4        /* tty_wakeup() pending (asm code only) */
26390   #define ODEVREADY MS_CTS        /* external device hardware ready (CTS) */
26391   #define OQUEUED     0x20        /* output buffer not empty */
26392   #define OSWREADY    0x40        /* external device software ready (no xoff) */
26393   #define ODEVHUP  MS_RLSD        /* external device has dropped carrier */
26394   #define OSOFTBITS  (ODONE | ORAW | OWAKEUP | OQUEUED | OSWREADY)
26395                                   /* user-defined bits */
26396   #if (OSOFTBITS | ODEVREADY | ODEVHUP) == OSOFTBITS
26397                                   /* a weak sanity check */
26398   #error                          /* bits are not unique */
26399   #endif
26400     unsigned char oxoff;          /* char to stop output */
26401     bool_t inhibited;             /* output inhibited? (follows tty_inhibited) */
26402     bool_t drain;                 /* if set drain output and reconfigure line */
26403     int ocount;                   /* number of bytes in the output buffer */
26404     char *ohead;                  /* next free spot in output buffer */
26405     char *otail;                  /* next char to output */
26406
26407   #if (MACHINE == IBM_PC)
26408     port_t xmit_port;             /* i/o ports */
26409     port_t recv_port;
26410     port_t div_low_port;
26411     port_t div_hi_port;
26412     port_t int_enab_port;
26413     port_t int_id_port;
26414     port_t line_ctl_port;
26415     port_t modem_ctl_port;
26416     port_t line_status_port;
26417     port_t modem_status_port;
26418   #endif
26419
26420     unsigned char lstatus;        /* last line status */
26421     unsigned char pad;            /* ensure alignment for 16-bit ints */
26422     unsigned framing_errors;      /* error counts (no reporting yet) */
26423     unsigned overrun_errors;
26424     unsigned parity_errors;
26425     unsigned break_interrupts;
26426
26427     char ibuf[RS_IBUFSIZE];       /* input buffer */
26428     char obuf[RS_OBUFSIZE];       /* output buffer */
26429   } rs232_t;
26430
26431   PUBLIC rs232_t rs_lines[NR_RS_LINES];
26432
26433   /* Table and macro to translate an RS232 line number to its rs_lines entry. */
26434   PRIVATE rs232_t *p_rs_addr[NR_RS_LINES];
```

```
26435
26436   #define rs_addr(line)   (p_rs_addr[line])
26437
26438   #if (MACHINE == IBM_PC)
26439   /* 8250 base addresses. */
26440   PRIVATE port_t addr_8250[] = {
26441     0x3F8,                        /* COM1: (line 0); COM3 might be at 0x3E8 */
26442     0x2F8,                        /* COM2: (line 1); COM4 might be at 0x2E8 */
26443   };
26444   #endif
26445
26446   FORWARD _PROTOTYPE( int rs232_1handler, (int irq)                       );
26447   FORWARD _PROTOTYPE( int rs232_2handler, (int irq)                       );
26448   FORWARD _PROTOTYPE( void in_int, (rs232_t *rs)                          );
26449   FORWARD _PROTOTYPE( void line_int, (rs232_t *rs)                        );
26450   FORWARD _PROTOTYPE( void modem_int, (rs232_t *rs)                       );
26451   FORWARD _PROTOTYPE( void rs_write, (tty_t *tp)                          );
26452   FORWARD _PROTOTYPE( void rs_echo, (tty_t *tp, int c)                    );
26453   FORWARD _PROTOTYPE( void rs_ioctl, (tty_t *tp)                          );
26454   FORWARD _PROTOTYPE( void rs_config, (rs232_t *rs)                       );
26455   FORWARD _PROTOTYPE( void rs_read, (tty_t *tp)                           );
26456   FORWARD _PROTOTYPE( void rs_icancel, (tty_t *tp)                        );
26457   FORWARD _PROTOTYPE( void rs_ocancel, (tty_t *tp)                        );
26458   FORWARD _PROTOTYPE( void rs_ostart, (rs232_t *rs)                       );
26459   FORWARD _PROTOTYPE( void rs_break, (tty_t *tp)                          );
26460   FORWARD _PROTOTYPE( void out_int, (rs232_t *rs)                         );
26461
26462
26463   /*===========================================================================*
26464    *                               rs_write                                    *
26465    *===========================================================================*/
26466   PRIVATE void rs_write(tp)
26467   register tty_t *tp;
26468   {
26469   /* (*devwrite)() routine for RS232. */
26470
26471     rs232_t *rs = tp->tty_priv;
26472     int count, ocount;
26473     phys_bytes user_phys;
26474
26475     if (rs->inhibited != tp->tty_inhibited) {
26476           /* Inhibition state has changed. */
26477           lock();
26478           rs->ostate |= OSWREADY;
26479           if (tp->tty_inhibited) rs->ostate &= ~OSWREADY;
26480           unlock();
26481           rs->inhibited = tp->tty_inhibited;
26482     }
26483
26484     if (rs->drain) {
26485           /* Wait for the line to drain then reconfigure and continue output. */
26486           if (rs->ocount > 0) return;
26487           rs->drain = FALSE;
26488           rs_config(rs);
26489     }
26490
26491     /* While there is something to do. */
26492     for (;;) {
26493           ocount = buflen(rs->obuf) - rs->ocount;
26494           count = bufend(rs->obuf) - rs->ohead;
```

```
26495           if (count > ocount) count = ocount;
26496           if (count > tp->tty_outleft) count = tp->tty_outleft;
26497           if (count == 0 || tp->tty_inhibited) return;
26498
26499           /* Copy from user space to the RS232 output buffer. */
26500           user_phys = proc_vir2phys(proc_addr(tp->tty_outproc), tp->tty_out_vir);
26501           phys_copy(user_phys, vir2phys(rs->ohead), (phys_bytes) count);
26502
26503           /* Perform output processing on the output buffer. */
26504           out_process(tp, rs->obuf, rs->ohead, bufend(rs->obuf), &count, &ocount);
26505           if (count == 0) break;
26506
26507           /* Assume echoing messed up by output. */
26508           tp->tty_reprint = TRUE;
26509
26510           /* Bookkeeping. */
26511           lock();                 /* protect interrupt sensitive rs->ocount */
26512           rs->ocount += ocount;
26513           rs_ostart(rs);
26514           unlock();
26515           if ((rs->ohead += ocount) >= bufend(rs->obuf))
26516                   rs->ohead -= buflen(rs->obuf);
26517           tp->tty_out_vir += count;
26518           tp->tty_outcum += count;
26519           if ((tp->tty_outleft -= count) == 0) {
26520                   /* Output is finished, reply to the writer. */
26521                   tty_reply(tp->tty_outrepcode, tp->tty_outcaller,
26522                                           tp->tty_outproc, tp->tty_outcum);
26523                   tp->tty_outcum = 0;
26524           }
26525     }
26526   }


26529   /*==========================================================================*
26530    *                              rs_echo                                     *
26531    *==========================================================================*/
26532   PRIVATE void rs_echo(tp, c)
26533   tty_t *tp;                      /* which TTY */
26534   int c;                          /* character to echo */
26535   {
26536   /* Echo one character.  (Like rs_write, but only one character, optionally.) */
26537
26538     rs232_t *rs = tp->tty_priv;
26539     int count, ocount;
26540
26541     ocount = buflen(rs->obuf) - rs->ocount;
26542     if (ocount == 0) return;              /* output buffer full */
26543     count = 1;
26544     *rs->ohead = c;                       /* add one character */
26545
26546     out_process(tp, rs->obuf, rs->ohead, bufend(rs->obuf), &count, &ocount);
26547     if (count == 0) return;
26548
26549     lock();
26550     rs->ocount += ocount;
26551     rs_ostart(rs);
26552     unlock();
26553     if ((rs->ohead += ocount) >= bufend(rs->obuf)) rs->ohead -= buflen(rs->obuf);
26554   }
```

```
26557   /*===========================================================================*
26558    *                              rs_ioctl                                     *
26559    *===========================================================================*/
26560   PRIVATE void rs_ioctl(tp)
26561   tty_t *tp;                      /* which TTY */
26562   {
26563   /* Reconfigure the line as soon as the output has drained. */
26564     rs232_t *rs = tp->tty_priv;
26565
26566     rs->drain = TRUE;
26567   }


26570   /*===========================================================================*
26571    *                              rs_config                                    *
26572    *===========================================================================*/
26573   PRIVATE void rs_config(rs)
26574   rs232_t *rs;                    /* which line */
26575   {
26576   /* Set various line control parameters for RS232 I/O.
26577    * If DataBits == 5 and StopBits == 2, 8250 will generate 1.5 stop bits.
26578    * The 8250 can't handle split speed, so we use the input speed.
26579    */
26580
26581     tty_t *tp = rs->tty;
26582     int divisor;
26583     int line_controls;
26584     static struct speed2divisor {
26585           speed_t speed;
26586           int     divisor;
26587     } s2d[] = {
26588   #if (MACHINE == IBM_PC)
26589           { B50,          UART_FREQ / 50          },
26590   #endif
26591           { B75,          UART_FREQ / 75          },
26592           { B110,         UART_FREQ / 110         },
26593           { B134,         UART_FREQ * 10 / 1345   },
26594           { B150,         UART_FREQ / 150         },
26595           { B200,         UART_FREQ / 200         },
26596           { B300,         UART_FREQ / 300         },
26597           { B600,         UART_FREQ / 600         },
26598           { B1200,        UART_FREQ / 1200        },
26599   #if (MACHINE == IBM_PC)
26600           { B1800,        UART_FREQ / 1800        },
26601   #endif
26602           { B2400,        UART_FREQ / 2400        },
26603           { B4800,        UART_FREQ / 4800        },
26604           { B9600,        UART_FREQ / 9600        },
26605           { B19200,       UART_FREQ / 19200       },
26606   #if (MACHINE == IBM_PC)
26607           { B38400,       UART_FREQ / 38400       },
26608           { B57600,       UART_FREQ / 57600       },
26609           { B115200,      UART_FREQ / 115200L     },
26610   #endif
26611     };
26612     struct speed2divisor *s2dp;
26613
26614     /* RS232 needs to know the xoff character, and if CTS works. */
```

```
26615     rs->oxoff = tp->tty_termios.c_cc[VSTOP];
26616     rs->cts = (tp->tty_termios.c_cflag & CLOCAL) ? MS_CTS : 0;
26617
26618     /* Look up the 8250 rate divisor from the output speed. */
26619     divisor = 0;
26620     for (s2dp = s2d; s2dp < s2d + sizeof(s2d)/sizeof(s2d[0]); s2dp++) {
26621           if (s2dp->speed == tp->tty_termios.c_ospeed) divisor = s2dp->divisor;
26622     }
26623     if (divisor == 0) return;     /* B0? */
26624
26625   #if (MACHINE == IBM_PC)
26626     /* Compute line control flag bits. */
26627     line_controls = 0;
26628     if (tp->tty_termios.c_cflag & PARENB) {
26629           line_controls |= LC_PARITY;
26630           if (!(tp->tty_termios.c_cflag & PARODD)) line_controls |= LC_PAREVEN;
26631     }
26632     if (divisor >= (UART_FREQ / 110)) line_controls |= LC_2STOP_BITS;
26633     line_controls |= (tp->tty_termios.c_cflag & CSIZE) >> 2;
26634
26635     /* Lock out interrupts while setting the speed. The receiver register is
26636      * going to be hidden by the div_low register, but the input interrupt
26637      * handler relies on reading it to clear the interrupt and avoid looping
26638      * forever.
26639      */
26640     lock();
26641
26642     /* Select the baud rate divisor registers and change the rate. */
26643     out_byte(rs->line_ctl_port, LC_ADDRESS_DIVISOR);
26644     out_byte(rs->div_low_port, divisor);
26645     out_byte(rs->div_hi_port, divisor >> 8);
26646
26647     /* Change the line controls and reselect the usual registers. */
26648     out_byte(rs->line_ctl_port, line_controls);
26649
26650     rs->ostate |= ORAW;
26651     if ((tp->tty_termios.c_lflag & IXON) && rs->oxoff != _POSIX_VDISABLE)
26652           rs->ostate &= ~ORAW;
26653
26654     unlock();
26655
26656   #else /* MACHINE == ATARI */
26657
26658     line_controls = U_Q16;
26659     if (tp->tty_termios.c_cflag & PARENB) {
26660           line_controls |= U_PAR;
26661           if (!(tp->tty_termios.c_cflag & PARODD)) line_controls |= U_EVEN;
26662     }
26663     line_controls |= (divisor >= (UART_FREQ / 110)) ? U_ST2 : U_ST1;
26664
26665     switch (tp->tty_termios.c_cflag & CSIZE) {    /* XXX - are U_Dn like CSn? */
26666           case CS5:       line_controls |= U_D5; break;
26667           case CS5:       line_controls |= U_D6; break;
26668           case CS5:       line_controls |= U_D7; break;
26669           case CS5:       line_controls |= U_D8; break;
26670     }
26671     lock();
26672     MFP->mf_ucr = line_controls;
26673     MFP->mf_tddr = divisor;
26674     unlock();
```

```
26675   #endif /* MACHINE == ATARI */
26676   }


26679   /*===========================================================================*
26680    *                              rs_init                                      *
26681    *===========================================================================*/
26682   PUBLIC void rs_init(tp)
26683   tty_t *tp;                      /* which TTY */
26684   {
26685   /* Initialize RS232 for one line. */
26686
26687     register rs232_t *rs;
26688     int line;
26689   #if (MACHINE == IBM_PC)
26690     port_t this_8250;
26691     int irq;
26692     long v;
26693   #endif
26694
26695     /* Associate RS232 and TTY structures. */
26696     line = tp - &tty_table[NR_CONS];
26697     rs = tp->tty_priv = &rs_lines[line];
26698     rs->tty = tp;
26699
26700     /* Set up input queue. */
26701     rs->ihead = rs->itail = rs->ibuf;
26702
26703   #if (MACHINE == IBM_PC)
26704     /* Precalculate port numbers for speed. Magic numbers in the code (once). */
26705     this_8250 = addr_8250[line];
26706     rs->xmit_port = this_8250 + 0;
26707     rs->recv_port = this_8250 + 0;
26708     rs->div_low_port = this_8250 + 0;
26709     rs->div_hi_port = this_8250 + 1;
26710     rs->int_enab_port = this_8250 + 1;
26711     rs->int_id_port = this_8250 + 2;
26712     rs->line_ctl_port = this_8250 + 3;
26713     rs->modem_ctl_port = this_8250 + 4;
26714     rs->line_status_port = this_8250 + 5;
26715     rs->modem_status_port = this_8250 + 6;
26716   #endif
26717
26718     /* Set up the hardware to a base state, in particular
26719      *    o turn off DTR (MC_DTR) to try to stop the external device.
26720      *    o be careful about the divisor latch.  Some BIOS's leave it enabled
26721      *      here and that caused trouble (no interrupts) in version 1.5 by
26722      *      hiding the interrupt enable port in the next step, and worse trouble
26723      *      (continual interrupts) in an old version by hiding the receiver
26724      *      port in the first interrupt.  Call rs_ioctl() early to avoid this.
26725      *    o disable interrupts at the chip level, to force an edge transition
26726      *      on the 8259 line when interrupts are next enabled and active.
26727      *      RS232 interrupts are guaranteed to be disabled now by the 8259
26728      *      mask, but there used to be trouble if the mask was set without
26729      *      handling a previous interrupt.
26730      */
26731     istop(rs);                      /* sets modem_ctl_port */
26732     rs_config(rs);
26733   #if (MACHINE == IBM_PC)
26734     out_byte(rs->int_enab_port, 0);
```

```
26735   #endif
26736
26737     /* Clear any harmful leftover interrupts.  An output interrupt is harmless
26738      * and will occur when interrupts are enabled anyway.  Set up the output
26739      * queue using the status from clearing the modem status interrupt.
26740      */
26741   #if (MACHINE == IBM_PC)
26742     in_byte(rs->line_status_port);
26743     in_byte(rs->recv_port);
26744   #endif
26745     rs->ostate = devready(rs) | ORAW | OSWREADY;  /* reads modem_ctl_port */
26746     rs->ohead = rs->otail = rs->obuf;
26747
26748   #if (MACHINE == IBM_PC)
26749     /* Enable interrupts for both interrupt controller and device. */
26750     irq = (line & 1) ? SECONDARY_IRQ : RS232_IRQ;
26751
26752   #if ENABLE_NETWORKING
26753     /* The ethernet driver may steal the IRQ of an RS232 line. */
26754     v = ETHER_IRQ;
26755     switch (env_parse("DPETH0", "x:d:x", 1, &v, 0L, (long) NR_IRQ_VECTORS-1)) {
26756     case EP_ON:
26757     case EP_SET:
26758           if (v == irq) return;           /* IRQ in use, don't configure line */
26759     }
26760   #endif
26761
26762     put_irq_handler(irq, (line & 1) ? rs232_2handler : rs232_1handler);
26763     enable_irq(irq);
26764     out_byte(rs->int_enab_port, IE_LINE_STATUS_CHANGE | IE_MODEM_STATUS_CHANGE
26765                               | IE_RECEIVER_READY | IE_TRANSMITTER_READY);
26766   #else /* MACHINE == ATARI */
26767     /* Initialize the 68901 chip, then enable interrupts. */
26768     MFP->mf_scr = 0x00;
26769     MFP->mf_tcdcr |= T_Q004;
26770     MFP->mf_rsr = R_ENA;
26771     MFP->mf_tsr = T_ENA;
26772     MFP->mf_aer = (MFP->mf_aer | (IO_SCTS|IO_SDCD)) ^
26773                     (MFP->mf_gpip & (IO_SCTS|IO_SDCD));
26774     MFP->mf_ddr = (MFP->mf_ddr & ~ (IO_SCTS|IO_SDCD));
26775     MFP->mf_iera |= (IA_RRDY|IA_RERR|IA_TRDY|IA_TERR);
26776     MFP->mf_imra |= (IA_RRDY|IA_RERR|IA_TRDY|IA_TERR);
26777     MFP->mf_ierb |= (IB_SCTS|IB_SDCD);
26778     MFP->mf_imrb |= (IB_SCTS|IB_SDCD);
26779   #endif /* MACHINE == ATARI */
26780
26781     /* Fill in TTY function hooks. */
26782     tp->tty_devread = rs_read;
26783     tp->tty_devwrite = rs_write;
26784     tp->tty_echo = rs_echo;
26785     tp->tty_icancel = rs_icancel;
26786     tp->tty_ocancel = rs_ocancel;
26787     tp->tty_ioctl = rs_ioctl;
26788     tp->tty_break = rs_break;
26789
26790     /* Tell external device we are ready. */
26791     istart(rs);
26792   }
```

```
26795   /*===========================================================================*
26796    *                              rs_icancel                                   *
26797    *===========================================================================*/
26798   PRIVATE void rs_icancel(tp)
26799   tty_t *tp;                      /* which TTY */
26800   {
26801   /* Cancel waiting input. */
26802     rs232_t *rs = tp->tty_priv;
26803
26804     lock();
26805     rs->icount = 0;
26806     rs->itail = rs->ihead;
26807     istart(rs);
26808     unlock();
26809   }


26812   /*===========================================================================*
26813    *                              rs_ocancel                                   *
26814    *===========================================================================*/
26815   PRIVATE void rs_ocancel(tp)
26816   tty_t *tp;                      /* which TTY */
26817   {
26818   /* Cancel pending output. */
26819     rs232_t *rs = tp->tty_priv;
26820
26821     lock();
26822     rs->ostate &= ~(ODONE | OQUEUED);
26823     rs->ocount = 0;
26824     rs->otail = rs->ohead;
26825     unlock();
26826   }


26829   /*===========================================================================*
26830    *                              rs_read                                      *
26831    *===========================================================================*/
26832   PRIVATE void rs_read(tp)
26833   tty_t *tp;                      /* which tty */
26834   {
26835   /* Process characters from the circular input buffer. */
26836
26837     rs232_t *rs = tp->tty_priv;
26838     int icount, count, ostate;
26839
26840     if (!(tp->tty_termios.c_cflag & CLOCAL)) {
26841           /* Send a SIGHUP if hangup detected. */
26842           lock();
26843           ostate = rs->ostate;
26844           rs->ostate &= ~ODEVHUP;         /* save ostate, clear DEVHUP */
26845           unlock();
26846           if (ostate & ODEVHUP) { sigchar(tp, SIGHUP); return; }
26847     }
26848
26849     while ((count = rs->icount) > 0) {
26850           icount = bufend(rs->ibuf) - rs->itail;
26851           if (count > icount) count = icount;
26852
26853           /* Perform input processing on (part of) the input buffer. */
26854           if ((count = in_process(tp, rs->itail, count)) == 0) break;
```

```
26855
26856           lock();                 /* protect interrupt sensitive variables */
26857           rs->icount -= count;
26858           if (!rs->idevready && rs->icount < RS_ILOWWATER) istart(rs);
26859           unlock();
26860           if ((rs->itail += count) == bufend(rs->ibuf)) rs->itail = rs->ibuf;
26861       }
26862   }


26865   /*===========================================================================*
26866    *                              rs_ostart                                    *
26867    *===========================================================================*/
26868   PRIVATE void rs_ostart(rs)
26869   rs232_t *rs;                    /* which rs line */
26870   {
26871   /* Tell RS232 there is something waiting in the output buffer. */
26872
26873     rs->ostate |= OQUEUED;
26874     if (txready(rs)) out_int(rs);
26875   }


26878   /*===========================================================================*
26879    *                              rs_break                                     *
26880    *===========================================================================*/
26881   PRIVATE void rs_break(tp)
26882   tty_t *tp;                      /* which tty */
26883   {
26884   /* Generate a break condition by setting the BREAK bit for 0.4 sec. */
26885     rs232_t *rs = tp->tty_priv;
26886     int line_controls;
26887
26888     line_controls = in_byte(rs->line_ctl_port);
26889     out_byte(rs->line_ctl_port, line_controls | LC_BREAK);
26890     milli_delay(400);                                       /* ouch */
26891     out_byte(rs->line_ctl_port, line_controls);
26892   }


26895   /* Low level (interrupt) routines. */
26896
26897   #if (MACHINE == IBM_PC)
26898   /*===========================================================================*
26899    *                              rs232_1handler                               *
26900    *===========================================================================*/
26901   PRIVATE int rs232_1handler(irq)
26902   int irq;
26903   {
26904   /* Interrupt hander for IRQ4.
26905    * Only 1 line (usually COM1) should use it.
26906    */
26907
26908     register rs232_t *rs = &rs_lines[0];
26909
26910     while (TRUE) {
26911           /* Loop to pick up ALL pending interrupts for device.
26912            * This usually just wastes time unless the hardware has a buffer
26913            * (and then we have to worry about being stuck in the loop too long).
26914            * Unfortunately, some serial cards lock up without this.
```

```
26915            */
26916           switch (in_byte(rs->int_id_port)) {
26917           case IS_RECEIVER_READY:
26918                   in_int(rs);
26919                   continue;
26920           case IS_TRANSMITTER_READY:
26921                   out_int(rs);
26922                   continue;
26923           case IS_MODEM_STATUS_CHANGE:
26924                   modem_int(rs);
26925                   continue;
26926           case IS_LINE_STATUS_CHANGE:
26927                   line_int(rs);
26928                   continue;
26929           }
26930           return(1);      /* reenable serial interrupt */
26931     }
26932   }


26935   /*==========================================================================*
26936    *                              rs232_2handler                              *
26937    *==========================================================================*/
26938   PRIVATE int rs232_2handler(irq)
26939   int irq;
26940   {
26941   /* Interrupt hander for IRQ3.
26942    * Only 1 line (usually COM2) should use it.
26943    */
26944
26945     register rs232_t *rs = &rs_lines[1];
26946
26947     while (TRUE) {
26948           switch (in_byte(rs->int_id_port)) {
26949           case IS_RECEIVER_READY:
26950                   in_int(rs);
26951                   continue;
26952           case IS_TRANSMITTER_READY:
26953                   out_int(rs);
26954                   continue;
26955           case IS_MODEM_STATUS_CHANGE:
26956                   modem_int(rs);
26957                   continue;
26958           case IS_LINE_STATUS_CHANGE:
26959                   line_int(rs);
26960                   continue;
26961           }
26962           return(1);      /* reenable serial interrupt */
26963     }
26964   }
26965   #else /* MACHINE == ATARI */
26966   /*==========================================================================*
26967    *                              siaint                                      *
26968    *==========================================================================*/
26969   PRIVATE void siaint(type)
26970   int    type;           /* interrupt type */
26971   {
26972   /* siaint is the rs232 interrupt procedure for Atari ST's. For ST there are
26973    * as much as 5 interrupt lines used for rs232. The trap type byte left on the
26974    * stack by the assembler interrupt handler identifies the interrupt cause.
```

```
26975     */
26976
26977      register unsigned char  code;
26978      register rs232_t *rs = &rs_lines[0];
26979      int s = lock();
26980
26981      switch (type & 0x00FF)
26982      {
26983            case 0x00:              /* receive buffer full */
26984                    in_int(rs);
26985                    break;
26986            case 0x01:              /* receive error */
26987                    line_int(rs);
26988                    break;
26989            case 0x02:              /* transmit buffer empty */
26990                    out_int(rs);
26991                    break;
26992            case 0x03:              /* transmit error */
26993                    code = MFP->mf_tsr;
26994                    if (code & ~(T_ENA | T_UE | T_EMPTY))
26995                    {
26996                        printf("sia: transmit error: status=%x\r\n", code);
26997                        /* MFP->mf_udr = lastchar; */ /* retry */
26998                    }
26999                    break;
27000            case 0x04:               /* modem lines change */
27001                    modem_int(rs);
27002                    break;
27003      }
27004      restore(s);
27005    }
27006    #endif /* MACHINE == ATARI */
27007
27008
27009    /*==========================================================================*
27010     *                              in_int                                      *
27011     *==========================================================================*/
27012    PRIVATE void in_int(rs)
27013    register rs232_t *rs;           /* line with input interrupt */
27014    {
27015    /* Read the data which just arrived.
27016     * If it is the oxoff char, clear OSWREADY, else if OSWREADY was clear, set
27017     * it and restart output (any char does this, not just xon).
27018     * Put data in the buffer if room, otherwise discard it.
27019     * Set a flag for the clock interrupt handler to eventually notify TTY.
27020     */
27021
27022      int c;
27023
27024    #if (MACHINE == IBM_PC)
27025      c = in_byte(rs->recv_port);
27026    #else /* MACHINE == ATARI */
27027      c = MFP->mf_udr;
27028    #endif
27029
27030      if (!(rs->ostate & ORAW)) {
27031            if (c == rs->oxoff) {
27032                    rs->ostate &= ~OSWREADY;
27033            } else
27034            if (!(rs->ostate & OSWREADY)) {
```

```
27035                   rs->ostate |= OSWREADY;
27036                   if (txready(rs)) out_int(rs);
27037           }
27038     }
27039
27040     if (rs->icount == buflen(rs->ibuf)) return;   /* input buffer full, discard */
27041
27042     if (++rs->icount == RS_IHIGHWATER && rs->idevready) istop(rs);
27043     *rs->ihead = c;
27044     if (++rs->ihead == bufend(rs->ibuf)) rs->ihead = rs->ibuf;
27045     if (rs->icount == 1) {
27046           rs->tty->tty_events = 1;
27047           force_timeout();
27048     }
27049   }


27052   /*===========================================================================*
27053    *                               line_int                                    *
27054    *===========================================================================*/
27055   PRIVATE void line_int(rs)
27056   register rs232_t *rs;           /* line with line status interrupt */
27057   {
27058   /* Check for and record errors. */
27059
27060   #if (MACHINE == IBM_PC)
27061     rs->lstatus = in_byte(rs->line_status_port);
27062   #else /* MACHINE == ATARI */
27063     rs->lstatus = MFP->mf_rsr;
27064     MFP->mf_rsr &= R_ENA;
27065     rs->pad = MFP->mf_udr;        /* discard char in case of LS_OVERRUN_ERR */
27066   #endif /* MACHINE == ATARI */
27067     if (rs->lstatus & LS_FRAMING_ERR) ++rs->framing_errors;
27068     if (rs->lstatus & LS_OVERRUN_ERR) ++rs->overrun_errors;
27069     if (rs->lstatus & LS_PARITY_ERR) ++rs->parity_errors;
27070     if (rs->lstatus & LS_BREAK_INTERRUPT) ++rs->break_interrupts;
27071   }


27074   /*===========================================================================*
27075    *                               modem_int                                   *
27076    *===========================================================================*/
27077   PRIVATE void modem_int(rs)
27078   register rs232_t *rs;           /* line with modem interrupt */
27079   {
27080   /* Get possibly new device-ready status, and clear ODEVREADY if necessary.
27081    * If the device just became ready, restart output.
27082    */
27083
27084   #if (MACHINE == ATARI)
27085     /* Set active edge interrupt so that next change causes a new interrupt */
27086     MFP->mf_aer = (MFP->mf_aer | (IO_SCTS|IO_SDCD)) ^
27087                    (MFP->mf_gpip & (IO_SCTS|IO_SDCD));
27088   #endif
27089
27090     if (devhup(rs)) {
27091           rs->ostate |= ODEVHUP;
27092           rs->tty->tty_events = 1;
27093           force_timeout();
27094     }
```

```
27095
27096     if (!devready(rs))
27097           rs->ostate &= ~ODEVREADY;
27098     else if (!(rs->ostate & ODEVREADY)) {
27099           rs->ostate |= ODEVREADY;
27100           if (txready(rs)) out_int(rs);
27101     }
27102   }


27105   /*===========================================================================*
27106    *                               out_int                                     *
27107    *===========================================================================*/
27108   PRIVATE void out_int(rs)
27109   register rs232_t *rs;           /* line with output interrupt */
27110   {
27111   /* If there is output to do and everything is ready, do it (local device is
27112    * known ready).
27113    * Notify TTY when the buffer goes empty.
27114    */
27115
27116     if (rs->ostate >= (ODEVREADY | OQUEUED | OSWREADY)) {
27117           /* Bit test allows ORAW and requires the others. */
27118   #if (MACHINE == IBM_PC)
27119           out_byte(rs->xmit_port, *rs->otail);
27120   #else /* MACHINE == ATARI */
27121           MFP->mf_udr = *rs->otail;
27122   #endif
27123           if (++rs->otail == bufend(rs->obuf)) rs->otail = rs->obuf;
27124           if (--rs->ocount == 0) {
27125                   rs->ostate ^= (ODONE | OQUEUED);  /* ODONE on, OQUEUED off */
27126                   rs->tty->tty_events = 1;
27127                   force_timeout();
27128           } else
27129           if (rs->ocount == RS_OLOWWATER) {       /* running low? */
27130                   rs->tty->tty_events = 1;
27131                   force_timeout();
27132           }
27133     }
27134   }
27135   #endif /* NR_RS_LINES > 0 */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/sb16_dsp.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

27200   /* This file contains the driver for a DSP (Digital Sound Processor) on
27201    * a SoundBlaster 16 (ASP) soundcard.
27202    *
27203    * The driver supports the following operations (using message format m2):
27204    *
27205    *    m_type      DEVICE    PROC_NR     COUNT    POSITION  ADRRESS
27206    * ----------------------------------------------------------------
27207    * |  DEV_OPEN  | device  | proc nr |         |         |         |
27208    * |------------+---------+---------+---------+---------+---------|
27209    * |  DEV_CLOSE | device  | proc nr |         |         |         |
```

```
27210     * |------------+---------+---------+---------+---------+---------|
27211     * |  DEV_READ  | device  | proc nr |  bytes  |         | buf ptr |
27212     * |------------+---------+---------+---------+---------+---------|
27213     * |  DEV_WRITE | device  | proc nr |  bytes  |         | buf ptr |
27214     * |------------+---------+---------+---------+---------+---------|
27215     * |  DEV_IOCTL | device  | proc nr |func code|         | buf ptr |
27216     * ----------------------------------------------------------------
27217     *
27218     * The file contains one entry point:
27219     *
27220     *   dsp_task:  main entry when system is brought up
27221     *
27222     *  May 20 1995                Author: Michel R. Prevenier
27223     */
27224
27225
27226    #include "kernel.h"
27227    #include <minix/com.h>
27228    #include <minix/callnr.h>
27229    #include <sys/ioctl.h>
27230    #if __minix_vmd
27231    #include "proc.h"
27232    #include "config.h"
27233    #endif
27234    #include "sb16.h"
27235
27236    #if ENABLE_SB_AUDIO
27237
27238    /* prototypes */
27239    FORWARD _PROTOTYPE( void init_buffer, (void));
27240    FORWARD _PROTOTYPE( int dsp_init, (void));
27241    FORWARD _PROTOTYPE( int dsp_handler, (int irq));
27242    FORWARD _PROTOTYPE( int dsp_open, (message *m_ptr));
27243    FORWARD _PROTOTYPE( int dsp_close, (message *m_ptr));
27244    FORWARD _PROTOTYPE( int dsp_ioctl, (message *m_ptr));
27245    FORWARD _PROTOTYPE( int dsp_write, (message *m_ptr));
27246    FORWARD _PROTOTYPE( int dsp_read, (message *m_ptr));
27247    FORWARD _PROTOTYPE( int dsp_reset, (void));
27248    FORWARD _PROTOTYPE( int dsp_command, (int value));
27249    FORWARD _PROTOTYPE( int dsp_set_speed, (unsigned int speed));
27250    FORWARD _PROTOTYPE( int dsp_set_size, (unsigned int size));
27251    FORWARD _PROTOTYPE( int dsp_set_stereo, (unsigned int stereo));
27252    FORWARD _PROTOTYPE( int dsp_set_bits, (unsigned int bits));
27253    FORWARD _PROTOTYPE( int dsp_set_sign, (unsigned int sign));
27254    FORWARD _PROTOTYPE( void dsp_dma_setup, (phys_bytes address, int count));
27255    FORWARD _PROTOTYPE( void dsp_setup, (void));
27256
27257    /* globals */
27258    #if __minix_vmd
27259    PRIVATE int DspTasknr = ANY;
27260    #endif
27261    PRIVATE int DspVersion[2];
27262    PRIVATE unsigned int DspStereo = DEFAULT_STEREO;
27263    PRIVATE unsigned int DspSpeed = DEFAULT_SPEED;
27264    PRIVATE unsigned int DspBits = DEFAULT_BITS;
27265    PRIVATE unsigned int DspSign = DEFAULT_SIGN;
27266    PRIVATE unsigned int DspFragmentSize = DSP_MAX_FRAGMENT_SIZE;
27267    PRIVATE int DspAvail = 0;
27268    PRIVATE int DspBusy = 0;
27269    PRIVATE int DmaBusy = 0;
```

```
27270   PRIVATE int DmaDone = 1;
27271   PRIVATE int DmaMode = 0;
27272
27273   PRIVATE char DmaBuffer[(long)2 * DMA_SIZE];
27274   PRIVATE char *DmaPtr;
27275   PRIVATE phys_bytes DmaPhys;
27276
27277
27278   /*============================================================================*
27279    *                              dsp_task                                      *
27280    *============================================================================*/
27281   PUBLIC void dsp_task()
27282   {
27283     message mess;
27284     int err, caller, proc_nr;
27285
27286   #if __minix_vmd
27287     DspTasknr = proc_number(proc_ptr);
27288   #endif
27289
27290     /* initialize the DMA buffer */
27291     init_buffer();
27292
27293     /* Here is the main loop of the sound task. It waits for a message, carries
27294      * it out, and sends a reply.
27295      */
27296     while (TRUE)
27297     {
27298           receive(ANY, &mess);
27299
27300           caller = mess.m_source;
27301           proc_nr = mess.PROC_NR;
27302
27303           switch (caller)
27304           {
27305             case HARDWARE:
27306                   /* Leftover interrupt. */
27307                   continue;
27308             case FS_PROC_NR:
27309                   /* The only legitimate caller. */
27310                   break;
27311             default:
27312                   printf("sb16: got message from %d\n", caller);
27313                   continue;
27314           }
27315
27316           /* Now carry out the work. */
27317           switch(mess.m_type)
27318           {
27319               case DEV_OPEN:      err = dsp_open(&mess);break;
27320               case DEV_CLOSE:     err = dsp_close(&mess);break;
27321               case DEV_IOCTL:     err = dsp_ioctl(&mess);break;
27322               case DEV_READ:      err = dsp_read(&mess);break;
27323               case DEV_WRITE:     err = dsp_write(&mess);break;
27324               default:            err = EINVAL;break;
27325           }
27326
27327           /* Finally, prepare and send the reply message. */
27328           mess.m_type = TASK_REPLY;
27329           mess.REP_PROC_NR = proc_nr;
```

```
27330
27331           mess.REP_STATUS = err;  /* #bytes transfered or error code */
27332           send(caller, &mess);    /* send reply to caller */
27333     }
27334   }


27337   /*==========================================================================*
27338    *                              init_buffer                                 *
27339    *==========================================================================*/
27340   PRIVATE void init_buffer()
27341   {
27342   /* Select a buffer that can safely be used for dma transfers.
27343    * Its absolute address is 'DmaPhys', the normal address is 'DmaPtr'.
27344    */
27345
27346     DmaPtr = DmaBuffer;
27347     DmaPhys = vir2phys(DmaBuffer);
27348
27349     if (dma_bytes_left(DmaPhys) < DMA_SIZE) {
27350           /* First half of buffer crosses a 64K boundary, can't DMA into that */
27351           DmaPtr += DMA_SIZE;
27352           DmaPhys += DMA_SIZE;
27353     }
27354   }


27357   /*==========================================================================*
27358    *                              dsp_open                                    *
27359    *==========================================================================*/
27360   PRIVATE int dsp_open(m_ptr)
27361   message *m_ptr;
27362   {
27363
27364   #if SB_DEBUG
27365     printf("sb16_open\n");
27366   #endif
27367
27368     /* try to detect SoundBlaster card */
27369     if (!DspAvail && dsp_init() != OK) return EIO;
27370
27371     /* Only one open at a time with soundcards */
27372     if (DspBusy) return EBUSY;
27373
27374     /* Start with a clean DSP */
27375     if (dsp_reset() != OK) return EIO;
27376
27377     /* Setup default values */
27378     DspStereo = DEFAULT_STEREO;
27379     DspSpeed = DEFAULT_SPEED;
27380     DspBits = DEFAULT_BITS;
27381     DspSign = DEFAULT_SIGN;
27382     DspFragmentSize = DMA_SIZE;
27383
27384     DspBusy = 1;
27385     DmaBusy = 0;
27386
27387     return OK;
27388   }
```

```
27391  /*============================================================================*
27392   *                              dsp_close                                     *
27393   *============================================================================*/
27394  PRIVATE int dsp_close(m_ptr)
27395  message *m_ptr;
27396  {
27397
27398  #if SB_DEBUG
27399    printf("dsp_close\n");
27400  #endif
27401
27402    DspBusy = 0;                  /* soundcard available again */
27403    DmaBusy = 0;
27404
27405    return OK;
27406  }


27409  /*============================================================================*
27410   *                              dsp_ioctl                                     *
27411   *============================================================================*/
27412  PRIVATE int dsp_ioctl(m_ptr)
27413  message *m_ptr;
27414  {
27415    int status;
27416    phys_bytes user_phys;
27417    unsigned int val;
27418
27419    /* Cannot change parameters during play or recording */
27420    if (DmaBusy) return EBUSY;
27421
27422    /* Get user data */
27423    if (m_ptr->REQUEST != DSPIORESET)
27424    {
27425      user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
27426                                                    sizeof(unsigned int));
27427      if (user_phys == 0) return(EFAULT);
27428      phys_copy(user_phys, vir2phys(&val), (phys_bytes) sizeof(val));
27429    }
27430
27431  #if SB_DEBUG
27432    printf("dsp_ioctl: got ioctl %d, argument: %d\n", m_ptr->REQUEST, val);
27433  #endif
27434
27435    switch(m_ptr->REQUEST)
27436    {
27437      case DSPIORATE:     status = dsp_set_speed(val);break;
27438      case DSPIOSTEREO:   status = dsp_set_stereo(val);break;
27439      case DSPIOBITS:     status = dsp_set_bits(val);break;
27440      case DSPIOSIZE:     status = dsp_set_size(val);break;
27441      case DSPIOSIGN:     status = dsp_set_sign(val);break;
27442      case DSPIOMAX:      {
27443                             val = DMA_SIZE;
27444                             phys_copy(vir2phys(&val), user_phys,
27445                                              (phys_bytes) sizeof(val));
27446                             status = OK;
27447                          };break;
27448      case DSPIORESET:    status = dsp_reset();break;
27449      default:            status = ENOTTY;break;
```

```
27450       }
27451
27452       return status;
27453     }


27456     /*============================================================================*
27457      *                              dsp_init                                      *
27458      *============================================================================*/
27459     PRIVATE int dsp_init()
27460     {
27461       int i;
27462
27463       if (dsp_reset () != OK)
27464       {
27465         printf("sb16: No SoundBlaster card detected\n");
27466         return -1;
27467       }
27468
27469       DspVersion[0] = DspVersion[1] = 0;
27470       dsp_command(DSP_GET_VERSION); /* Get DSP version bytes */
27471
27472       for (i = 1000; i; i--)
27473       {
27474         if (in_byte (DSP_DATA_AVL) & 0x80)
27475         {
27476           if (DspVersion[0] == 0)
27477             DspVersion[0] = in_byte (DSP_READ);
27478           else
27479           {
27480             DspVersion[1] = in_byte (DSP_READ);
27481             break;
27482           }
27483         }
27484       }
27485
27486       if (DspVersion[0] < 4)
27487       {
27488         printf("sb16: No SoundBlaster 16 compatible card detected\n");
27489         return -1;
27490       }
27491       else
27492         printf ("sb16: SoundBlaster DSP version %d.%d detected\n",
27493               DspVersion[0], DspVersion[1]);
27494
27495       /* set IRQ and DMA channels */
27496       mixer_set(MIXER_SET_IRQ, (1 << (SB_IRQ / 2 - 1)));
27497       mixer_set(MIXER_SET_DMA, (1 << SB_DMA_8 | 1 << SB_DMA_16));
27498
27499       /* register interrupt vector and enable irq */
27500       put_irq_handler(SB_IRQ, dsp_handler);
27501       enable_irq(SB_IRQ);
27502
27503       DspAvail = 1;
27504       return OK;
27505     }


27508     /*============================================================================*
27509      *                              dsp_handler                                   *
```

```
27510     *===========================================================================*/
27511    PRIVATE int dsp_handler(irq)
27512    int irq;
27513    {
27514
27515    #if SB_DEBUG2
27516      printf("SoundBlaster interrupt %d\n", irq);
27517    #endif
27518
27519      if (DmaDone)     /* Dma transfer is done */
27520      {
27521        /* Send DSP command to stop dma */
27522        dsp_command((DspBits == 8 ? DSP_CMD_DMA8HALT : DSP_CMD_DMA16HALT));
27523
27524        DmaBusy = 0;  /* Dma available again */
27525      }
27526
27527      /* Send interrupt to audio task and enable again */
27528    #if __minix_vmd
27529      interrupt(DspTasknr);
27530    #else
27531      interrupt(AUDIO);
27532    #endif
27533
27534      /* Acknowledge the interrupt on the DSP */
27535      (void) in_byte((DspBits == 8 ? DSP_DATA_AVL : DSP_DATA16_AVL));
27536
27537      return 1;
27538    }


27541    /*===========================================================================*
27542     *                               dsp_command                                 *
27543     *===========================================================================*/
27544    PRIVATE int dsp_command(value)
27545    int value;
27546    {
27547      int             i;
27548
27549      for (i = 0; i < SB_TIMEOUT; i++)
27550      {
27551        if ((in_byte (DSP_STATUS) & 0x80) == 0)
27552        {
27553          out_byte (DSP_COMMAND, value);
27554          return OK;
27555        }
27556      }
27557
27558      printf ("sb16: SoundBlaster: DSP Command(%x) timeout\n", value);
27559      return -1;
27560    }


27563    /*===========================================================================*
27564     *                               dsp_reset                                   *
27565     *===========================================================================*/
27566    PRIVATE int dsp_reset(void)
27567    {
27568      int i;
27569
```

```
27570       out_byte (DSP_RESET, 1);
27571       for(i =0; i<1000; i++); /* wait a while */
27572       out_byte (DSP_RESET, 0);
27573
27574       for (i = 0; i < 1000 && !(in_byte (DSP_DATA_AVL) & 0x80); i++);
27575
27576       if (in_byte (DSP_READ) != 0xAA) return EIO; /* No SoundBlaster */
27577
27578       DmaBusy = 0;
27579       DmaDone = 1;
27580
27581       return OK;
27582     }


27585     /*============================================================================*
27586      *                              dsp_set_speed                                 *
27587      *============================================================================*/
27588     static int dsp_set_speed(speed)
27589     unsigned int speed;
27590     {
27591     #if SB_DEBUG
27592       printf("sb16: setting speed to %u, stereo = %d\n", speed, DspStereo);
27593     #endif
27594
27595       if (speed < DSP_MIN_SPEED || speed > DSP_MAX_SPEED)
27596         return EPERM;
27597
27598       /* Soundblaster 16 can be programmed with real sample rates
27599        * instead of time constants
27600        *
27601        * Since you cannot sample and play at the same time
27602        * we set in- and output rate to the same value
27603        */
27604
27605       lock();                                   /* disable interrupts */
27606       dsp_command(DSP_INPUT_RATE);              /* set input rate */
27607       dsp_command(speed >> 8);                  /* high byte of speed */
27608       dsp_command(speed);                       /* low byte of speed */
27609       dsp_command(DSP_OUTPUT_RATE);             /* same for output rate */
27610       dsp_command(speed >> 8);
27611       dsp_command(speed);
27612       unlock();                                 /* enable interrupts */
27613
27614       DspSpeed = speed;
27615
27616       return OK;
27617     }


27620     /*============================================================================*
27621      *                              dsp_set_stereo                                *
27622      *============================================================================*/
27623     static int dsp_set_stereo(stereo)
27624     unsigned int stereo;
27625     {
27626       if (stereo)
27627         DspStereo = 1;
27628       else
27629         DspStereo = 0;
```

```
27630
27631     return OK;
27632   }


27635   /*==========================================================================*
27636    *                              dsp_set_bits                                *
27637    *==========================================================================*/
27638   static int dsp_set_bits(bits)
27639   unsigned int bits;
27640   {
27641     /* Sanity checks */
27642     if (bits != 8 && bits != 16) return EINVAL;
27643
27644     DspBits = bits;
27645
27646     return OK;
27647   }


27650   /*==========================================================================*
27651    *                              dsp_set_size                                *
27652    *==========================================================================*/
27653   static int dsp_set_size(size)
27654   unsigned int size;
27655   {
27656
27657   #if SB_DEBUG
27658     printf("sb16: set fragment size to %u\n", size);
27659   #endif
27660
27661     /* Sanity checks */
27662     if (size < DSP_MIN_FRAGMENT_SIZE ||
27663         size > DSP_MAX_FRAGMENT_SIZE ||
27664         size % 2 != 0)
27665       return EINVAL;
27666
27667     DspFragmentSize = size;
27668
27669     return OK;
27670   }


27673   /*==========================================================================*
27674    *                              dsp_set_sign                                *
27675    *==========================================================================*/
27676   static int dsp_set_sign(sign)
27677   unsigned int sign;
27678   {
27679
27680   #if SB_DEBUG
27681     printf("sb16: set sign to %u\n", sign);
27682   #endif
27683
27684     DspSign = (sign > 0 ? 1 : 0);
27685
27686     return OK;
27687   }
```

```
27690   /*==========================================================================*
27691    *                              dsp_dma_setup                               *
27692    *==========================================================================*/
27693   PRIVATE void dsp_dma_setup(address, count)
27694   phys_bytes address;
27695   int count;
27696
27697   {
27698   #if SB_DEBUG
27699     printf("Setting up %d bit DMA\n", DspBits);
27700   #endif
27701
27702     if (DspBits == 8)    /* 8 bit sound */
27703     {
27704       count--;
27705
27706       lock();
27707       out_byte(DMA8_MASK, SB_DMA_8 | 0x04);      /* Disable DMA channel */
27708       out_byte(DMA8_CLEAR, 0x00);                /* Clear flip flop */
27709
27710       /* set DMA mode */
27711       out_byte(DMA8_MODE,
27712                  (DmaMode == DEV_WRITE ? DMA8_AUTO_PLAY : DMA8_AUTO_REC));
27713
27714       out_byte(DMA8_ADDR, address >>  0);        /* Low_byte of address */
27715       out_byte(DMA8_ADDR, address >>  8);        /* High byte of address */
27716       out_byte(DMA8_PAGE, address >> 16);        /* 64K page number */
27717       out_byte(DMA8_COUNT, count >> 0);          /* Low byte of count */
27718       out_byte(DMA8_COUNT, count >> 8);          /* High byte of count */
27719       out_byte(DMA8_MASK, SB_DMA_8);             /* Enable DMA channel */
27720       unlock();
27721     }
27722     else  /* 16 bit sound */
27723     {
27724       count-= 2;
27725
27726       lock();
27727       out_byte(DMA16_MASK, (SB_DMA_16 & 3) | 0x04); /* Disable DMA channel */
27728       out_byte(DMA16_CLEAR, 0x00);                  /* Clear flip flop */
27729
27730       /* Set dma mode */
27731       out_byte(DMA16_MODE,
27732                  (DmaMode == DEV_WRITE ? DMA16_AUTO_PLAY : DMA16_AUTO_REC));
27733
27734       out_byte(DMA16_ADDR, (address >> 1) & 0xFF);  /* Low_byte of address */
27735       out_byte(DMA16_ADDR, (address >> 9) & 0xFF);  /* High byte of address */
27736       out_byte(DMA16_PAGE, (address >> 16) & 0xFE); /* 128K page number */
27737       out_byte(DMA16_COUNT, count >> 1);            /* Low byte of count */
27738       out_byte(DMA16_COUNT, count >> 9);            /* High byte of count */
27739       out_byte(DMA16_MASK, SB_DMA_16 & 3);          /* Enable DMA channel */
27740       unlock();
27741     }
27742   }


27745   /*==========================================================================*
27746    *                              dsp_setup                                   *
27747    *==========================================================================*/
27748   PRIVATE void dsp_setup()
27749   {
```

```
27750     /* Set current sample speed */
27751     dsp_set_speed(DspSpeed);
27752
27753     /* Put the speaker on */
27754      if (DmaMode == DEV_WRITE)
27755      {
27756        dsp_command (DSP_CMD_SPKON); /* put speaker on */
27757
27758         /* Program DSP with dma mode */
27759         dsp_command((DspBits == 8 ? DSP_CMD_8BITAUTO_OUT : DSP_CMD_16BITAUTO_OUT));
27760      }
27761      else
27762      {
27763        dsp_command (DSP_CMD_SPKOFF); /* put speaker off */
27764
27765        /* Program DSP with dma mode */
27766        dsp_command((DspBits == 8 ? DSP_CMD_8BITAUTO_IN : DSP_CMD_16BITAUTO_IN));
27767      }
27768
27769     /* Program DSP with transfer mode */
27770     if (!DspSign)
27771       dsp_command((DspStereo == 1 ? DSP_MODE_STEREO_US : DSP_MODE_MONO_US));
27772     else
27773       dsp_command((DspStereo == 1 ? DSP_MODE_STEREO_S : DSP_MODE_MONO_S));
27774
27775     /* Give length of fragment to DSP */
27776     if (DspBits == 8) /* 8 bit transfer */
27777     {
27778       /* #bytes - 1 */
27779       dsp_command((DspFragmentSize - 1) >> 0);
27780       dsp_command((DspFragmentSize - 1) >> 8);
27781     }
27782     else              /* 16 bit transfer */
27783     {
27784       /* #words - 1 */
27785       dsp_command((DspFragmentSize - 1) >> 1);
27786       dsp_command((DspFragmentSize - 1) >> 9);
27787     }
27788   }
27789
27790
27791   /*============================================================================*
27792    *                               dsp_write                                    *
27793    *============================================================================*/
27794   PRIVATE int dsp_write(m_ptr)
27795   message *m_ptr;
27796   {
27797     phys_bytes user_phys;
27798     message mess;
27799
27800     if (m_ptr->COUNT != DspFragmentSize) return EINVAL;
27801
27802     /* From this user address */
27803     user_phys = numap(m_ptr->PROC_NR, (vir_bytes)m_ptr->ADDRESS, DspFragmentSize);
27804     if (user_phys == 0) return EINVAL;
27805
27806     if (DmaBusy)    /* Dma already started */
27807     {
27808       if (DmaMode != m_ptr->m_type) return EBUSY;
27809
```

```
27810         DmaDone = 0;  /* No, we're not done yet */
27811
27812         /* Wait for next block to become free */
27813         receive(HARDWARE, &mess);
27814
27815         /* Copy first block to dma buffer */
27816         phys_copy(user_phys, DmaPhys, (phys_bytes) DspFragmentSize);
27817
27818       }
27819       else /* A new dma transfer has started */
27820       {
27821         DmaMode = DEV_WRITE;           /* Dma mode is writing */
27822
27823         /* Copy fragment to dma buffer */
27824         phys_copy(user_phys, DmaPhys, (phys_bytes) DspFragmentSize);
27825
27826         /* Set up the dma chip */
27827         dsp_dma_setup(DmaPhys, DspFragmentSize);
27828
27829         /* Set up the DSP */
27830         dsp_setup();
27831
27832         DmaBusy = 1;         /* Dma is busy */
27833       }
27834
27835       DmaDone = 1;            /* dma done for now */
27836
27837       return(DspFragmentSize);
27838     }


27841     /*============================================================================*
27842      *                              dsp_read                                      *
27843      *============================================================================*/
27844     PRIVATE int dsp_read(m_ptr)
27845     message *m_ptr;
27846     {
27847       phys_bytes user_phys;
27848       message mess;
27849
27850       if (m_ptr->COUNT != DspFragmentSize) return EINVAL;
27851
27852       /* To this user address */
27853       user_phys = numap(m_ptr->PROC_NR, (vir_bytes)m_ptr->ADDRESS, DspFragmentSize);
27854       if (user_phys == 0) return EINVAL;
27855
27856       if (DmaBusy)     /* Dma already started */
27857       {
27858         if (DmaMode != m_ptr->m_type) return EBUSY;
27859
27860         DmaDone = 0;   /* No, we're not done yet */
27861
27862         /* Wait for a full dma buffer */
27863         receive(HARDWARE, &mess);
27864
27865         /* Copy the buffer */
27866         phys_copy(DmaPhys, user_phys, (phys_bytes) DspFragmentSize);
27867       }
27868       else  /* A new dma transfer has started */
27869       {
```

```
27870         DmaMode = DEV_READ;               /* Dma mode is reading */
27871
27872         /* Set up the dma chip */
27873         dsp_dma_setup(DmaPhys, DspFragmentSize);
27874
27875         /* Set up the DSP */
27876         dsp_setup();
27877
27878         DmaBusy = 1;        /* Dma has started */
27879         DmaDone = 0;        /* Dma not done */
27880
27881         /* Wait for dma to finish with first block */
27882         receive(HARDWARE, &mess);
27883
27884         /* Copy dma buffer to user */
27885         phys_copy(DmaPhys, user_phys, (phys_bytes) DspFragmentSize);
27886       }
27887
27888       DmaDone = 1;   /* dma done for now */
27889
27890       return(DspFragmentSize);
27891     }
27892     #endif /* ENABLE_AUDIO */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/sb16_mixer.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
27900     /* This file contains the driver for the mixer on
27901      * a SoundBlaster 16 (ASP) soundcard.
27902      *
27903      * The driver supports the following operations (using message format m2):
27904      *
27905      *    m_type      DEVICE    PROC_NR     COUNT    POSITION  ADRRESS
27906      * ----------------------------------------------------------------
27907      * |  DEV_OPEN  | device  | proc nr |         |         |         |
27908      * |------------+---------+---------+---------+---------+---------|
27909      * |  DEV_CLOSE | device  | proc nr |         |         |         |
27910      * |------------+---------+---------+---------+---------+---------|
27911      * |  DEV_IOCTL | device  | proc nr |func code|         | buf_ptr |
27912      * ----------------------------------------------------------------
27913      *
27914      * The file contains one entry point:
27915      *
27916      *   mixer_task:  main entry when system is brought up
27917      *
27918      *  May 20 1995                 Author: Michel R. Prevenier
27919      */
27920
27921
27922     #include "kernel.h"
27923     #include <minix/com.h>
27924     #include <minix/callnr.h>
27925     #include <sys/ioctl.h>
27926     #include <minix/sound.h>
27927     #if __minix_vmd
27928     #include "config.h"
27929     #endif
```

```
27930    #include "sb16.h"
27931
27932    #if ENABLE_SB_AUDIO
27933
27934    /* Function prototypes */
27935    FORWARD _PROTOTYPE( int mixer_init, (void));
27936    FORWARD _PROTOTYPE( int mixer_open, (message *m_ptr));
27937    FORWARD _PROTOTYPE( int mixer_close, (message *m_ptr));
27938    FORWARD _PROTOTYPE( int mixer_ioctl, (message *m_ptr));
27939    FORWARD _PROTOTYPE( int mixer_get, (int reg));
27940    FORWARD _PROTOTYPE( int get_set_volume, (message *m_ptr, int flag));
27941    FORWARD _PROTOTYPE( int get_set_input, (message *m_ptr, int flag, int channel));
27942    FORWARD _PROTOTYPE( int get_set_output, (message *m_ptr, int flag));
27943
27944    PRIVATE int mixer_avail = 0;    /* Mixer exists? */
27945
27946
27947    /*===========================================================================*
27948     *                               mixer_task                                  *
27949     *===========================================================================*/
27950    PUBLIC void mixer_task()
27951    {
27952      message mess;
27953      int err, caller, proc_nr;
27954
27955      /* Here is the main loop of the mixer task. It waits for a message, carries
27956       * it out, and sends a reply.
27957       */
27958      while (TRUE)
27959      {
27960            receive(ANY, &mess);
27961
27962            caller = mess.m_source;
27963            proc_nr = mess.PROC_NR;
27964
27965            switch (caller)
27966            {
27967              case HARDWARE:
27968                    /* Leftover interrupt. */
27969                    continue;
27970              case FS_PROC_NR:
27971                    /* The only legitimate caller. */
27972                    break;
27973              default:
27974                    printf("sb16: got message from %d\n", caller);
27975                    continue;
27976            }
27977
27978            /* Now carry out the work. */
27979            switch(mess.m_type)
27980            {
27981                case DEV_OPEN:      err = mixer_open(&mess);break;
27982                case DEV_CLOSE:     err = mixer_close(&mess);break;
27983                case DEV_IOCTL:     err = mixer_ioctl(&mess);break;
27984                default:            err = EINVAL;break;
27985            }
27986
27987            /* Finally, prepare and send the reply message. */
27988            mess.m_type = TASK_REPLY;
27989            mess.REP_PROC_NR = proc_nr;
```

```
27990
27991           mess.REP_STATUS = err;  /* error code */
27992           send(caller, &mess);    /* send reply to caller */
27993       }
27994   }


27997   /*==========================================================================*
27998    *                              mixer_open                                  *
27999    *==========================================================================*/
28000   PRIVATE int mixer_open(m_ptr)
28001   message *m_ptr;
28002   {
28003
28004   #if SB_DEBUG
28005     printf("mixer_open\n");
28006   #endif
28007
28008     /* try to detect the mixer type */
28009     if (!mixer_avail && !mixer_init() != OK) return EIO;
28010
28011     return OK;
28012   }


28015   /*==========================================================================*
28016    *                              mixer_close                                 *
28017    *==========================================================================*/
28018   PRIVATE int mixer_close(m_ptr)
28019   message *m_ptr;
28020   {
28021
28022   #if SB_DEBUG
28023     printf("mixer_close\n");
28024   #endif
28025
28026     return OK;
28027   }


28030   /*==========================================================================*
28031    *                              mixer_ioctl                                 *
28032    *==========================================================================*/
28033   PRIVATE int mixer_ioctl(m_ptr)
28034   message *m_ptr;
28035   {
28036     int status;
28037
28038   #if SB_DEBUG
28039     printf("mixer: got ioctl %d\n", m_ptr->REQUEST);
28040   #endif
28041
28042
28043     switch(m_ptr->REQUEST)
28044     {
28045       case MIXIOGETVOLUME:      status = get_set_volume(m_ptr, 0);break;
28046       case MIXIOSETVOLUME:      status = get_set_volume(m_ptr, 1);break;
28047       case MIXIOGETINPUTLEFT:   status = get_set_input(m_ptr, 0, 0);break;
28048       case MIXIOGETINPUTRIGHT:  status = get_set_input(m_ptr, 0, 1);break;
28049       case MIXIOGETOUTPUT:      status = get_set_output(m_ptr, 0);break;
```

```
28050         case MIXIOSETINPUTLEFT:   status = get_set_input(m_ptr, 1, 0);break;
28051         case MIXIOSETINPUTRIGHT:  status = get_set_input(m_ptr, 1, 1);break;
28052         case MIXIOSETOUTPUT:      status = get_set_output(m_ptr, 1);break;
28053         default:                  status = ENOTTY;
28054       }
28055
28056       return status;
28057     }


28060     /*============================================================================*
28061      *                              mixer_init                                    *
28062      *============================================================================*/
28063     PRIVATE int mixer_init()
28064     {
28065       /* Try to detect the mixer by writing to MIXER_DAC_LEVEL if the
28066        * value written can be read back the mixer is there
28067        */
28068
28069       mixer_set(MIXER_DAC_LEVEL, 0x10);       /* write something to it */
28070       if (mixer_get(MIXER_DAC_LEVEL) != 0x10)
28071       {
28072         printf("sb16: Mixer not detected\n");
28073         return EIO;
28074       }
28075
28076       /* Enable Automatic Gain Control */
28077       mixer_set(MIXER_AGC, 0x01);
28078
28079     #if SB_DEBUG
28080       printf("Mixer detected\n");
28081     #endif
28082
28083       mixer_avail = 1;
28084       return OK;
28085     }


28088     /*============================================================================*
28089      *                              mixer_set                                     *
28090      *============================================================================*/
28091     PUBLIC int mixer_set(reg, data)
28092     int reg;
28093     int data;
28094     {
28095       int i;
28096
28097       out_byte(MIXER_REG, reg);
28098       for(i=0;i<100;i++);
28099       out_byte(MIXER_DATA, data);
28100
28101       return OK;
28102     }
28103
28104
28105     /*============================================================================*
28106      *                              mixer_get                                     *
28107      *============================================================================*/
28108     PRIVATE int mixer_get(reg)
28109     int reg;
```

```
28110   {
28111     int i;
28112
28113     out_byte(MIXER_REG, reg);
28114     for(i=0;i<100;i++);
28115     return (in_byte(MIXER_DATA) & 0xff);
28116   }
28117
28118
28119   /*============================================================================*
28120    *                              get_set_volume                                *
28121    *============================================================================*/
28122   PRIVATE int get_set_volume(m_ptr, flag)
28123   message *m_ptr;
28124   int flag;       /* 0 = get, 1 = set */
28125   {
28126     phys_bytes user_phys;
28127     struct volume_level level;
28128     int cmd_left, cmd_right, shift, max_level;
28129
28130     user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
28131                                         sizeof(struct volume_level));
28132     if (user_phys == 0) return(EFAULT);
28133     phys_copy(user_phys, vir2phys(&level), (phys_bytes) sizeof(level));
28134
28135     shift = 3;
28136     max_level = 0x1F;
28137
28138     switch (level.device)
28139     {
28140       case Master:  { cmd_left = MIXER_MASTER_LEFT;
28141                       cmd_right = MIXER_MASTER_RIGHT;
28142                     };break;
28143       case Dac:     { cmd_left = MIXER_DAC_LEFT;
28144                       cmd_right = MIXER_DAC_RIGHT;
28145                     };break;
28146       case Fm:      { cmd_left = MIXER_FM_LEFT;
28147                       cmd_right = MIXER_FM_RIGHT;
28148                     };break;
28149       case Cd:      { cmd_left = MIXER_CD_LEFT;
28150                       cmd_right = MIXER_CD_RIGHT;
28151                     };break;
28152       case Line:    { cmd_left = MIXER_LINE_LEFT;
28153                       cmd_right = MIXER_LINE_RIGHT;
28154                     };break;
28155       case Mic:     { cmd_left = cmd_right = MIXER_MIC_LEVEL;
28156                     };break;
28157       case Speaker: { cmd_left = cmd_right = MIXER_PC_LEVEL;
28158                       shift = 6;
28159                       max_level = 0x03;
28160                     };break;
28161       case Treble:  { cmd_left = MIXER_TREBLE_LEFT;
28162                       cmd_right = MIXER_TREBLE_RIGHT;
28163                       shift = 4;
28164                       max_level = 0x0F;
28165                     };break;
28166       case Bass:    { cmd_left = MIXER_BASS_LEFT;
28167                       cmd_right = MIXER_BASS_RIGHT;
28168                       shift = 4;
28169                       max_level = 0x0F;
```

```
28170                     };break;
28171       default:     return EINVAL;
28172     }
28173
28174     if (flag)  /* Set volume level */
28175     {
28176       if (level.right < 0) level.right = 0;
28177       else if (level.right > max_level) level.right = max_level;
28178       if (level.left < 0) level.left = 0;
28179       else if (level.left > max_level) level.left = max_level;
28180
28181       mixer_set(cmd_right, (level.right << shift));
28182       mixer_set(cmd_left, (level.left << shift));
28183     }
28184     else       /* Get volume level */
28185     {
28186       level.left = mixer_get(cmd_left);
28187       level.right = mixer_get(cmd_right);
28188
28189       level.left >>= shift;
28190       level.right >>= shift;
28191
28192       /* Copy back to user */
28193       phys_copy(vir2phys(&level), user_phys, (phys_bytes) sizeof(level));
28194     }
28195
28196     return OK;
28197   }


28200   /*==========================================================================*
28201    *                              get_set_input                               *
28202    *==========================================================================*/
28203   PRIVATE int get_set_input(m_ptr, flag, channel)
28204   message *m_ptr;
28205   int flag;       /* 0 = get, 1 = set */
28206   int channel;    /* 0 = left, 1 = right */
28207   {
28208     phys_bytes user_phys;
28209     struct inout_ctrl input;
28210     int input_cmd, input_mask, mask, del_mask, shift;
28211
28212     user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
28213                                           sizeof(struct inout_ctrl));
28214     if (user_phys == 0) return(EFAULT);
28215     phys_copy(user_phys, vir2phys(&input), (phys_bytes) sizeof(input));
28216
28217     input_cmd = (channel == 0 ? MIXER_IN_LEFT : MIXER_IN_RIGHT);
28218
28219     mask = mixer_get(input_cmd);
28220
28221     switch (input.device)
28222     {
28223       case Fm:   { shift = 5;
28224                    del_mask = 0x1F;
28225                  };break;
28226       case Cd:   { shift = 1;
28227                    del_mask = 0x79;
28228                  };break;
28229       case Line: { shift = 3;
```

```
28230                   del_mask = 0x67;
28231                 };break;
28232       case Mic:  { shift = 0;
28233                   del_mask = 0x7E;
28234                 };break;
28235       default:   return EINVAL;
28236     }
28237
28238     if (flag)  /* Set input */
28239     {
28240       input_mask =
28241              ((input.left == ON ? 1 : 0) << 1) | (input.right == ON ? 1 : 0);
28242
28243       if (shift > 0)
28244        input_mask <<= shift;
28245       else
28246        input_mask >>= 1;
28247
28248       mask &= del_mask;
28249       mask |= input_mask;
28250
28251       mixer_set(input_cmd, mask);
28252     }
28253     else      /* Get input */
28254     {
28255       if (shift > 0)
28256       {
28257         input.left = ((mask >> (shift+1)) & 1 == 1 ? ON : OFF);
28258         input.right = ((mask >> shift) & 1 == 1 ? ON : OFF);
28259       }
28260       else
28261         input.left = ((mask & 1) == 1 ? ON : OFF);
28262
28263       /* Copy back to user */
28264       phys_copy(vir2phys(&input), user_phys, (phys_bytes) sizeof(input));
28265     }
28266
28267     return OK;
28268   }


28271   /*==========================================================================*
28272    *                              get_set_output                              *
28273    *==========================================================================*/
28274   PRIVATE int get_set_output(m_ptr, flag)
28275   message *m_ptr;
28276   int flag;       /* 0 = get, 1 = set */
28277   {
28278     phys_bytes user_phys;
28279     struct inout_ctrl output;
28280     int output_mask, mask, del_mask, shift;
28281
28282     user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
28283                                           sizeof(struct inout_ctrl));
28284     if (user_phys == 0) return(EFAULT);
28285     phys_copy(user_phys, vir2phys(&output), (phys_bytes) sizeof(output));
28286
28287     mask = mixer_get(MIXER_OUTPUT_CTRL);
28288
28289     switch (output.device)
```

```
28290     {
28291       case Cd:   { shift = 1;
28292                    del_mask = 0x79;
28293                  };break;
28294       case Line: { shift = 3;
28295                    del_mask = 0x67;
28296                  };break;
28297       case Mic:  { shift = 0;
28298                    del_mask = 0x7E;
28299                  };break;
28300       default:   return EINVAL;
28301     }
28302
28303     if (flag)  /* Set input */
28304     {
28305       output_mask =
28306              ((output.left == ON ? 1 : 0) << 1) | (output.right == ON ? 1 : 0);
28307
28308       if (shift > 0)
28309        output_mask <<= shift;
28310       else
28311        output_mask >>= 1;
28312
28313       mask &= del_mask;
28314       mask |= output_mask;
28315
28316       mixer_set(MIXER_OUTPUT_CTRL, mask);
28317     }
28318     else      /* Get input */
28319     {
28320       if (shift > 0)
28321       {
28322         output.left = ((mask >> (shift+1)) & 1 == 1 ? ON : OFF);
28323         output.right = ((mask >> shift) & 1 == 1 ? ON : OFF);
28324       }
28325       else
28326         output.left = ((mask & 1) == 1 ? ON : OFF);
28327
28328       /* Copy back to user */
28329       phys_copy(vir2phys(&output), user_phys, (phys_bytes) sizeof(output));
28330     }
28331
28332     return OK;
28333   }
28334   #endif /* ENABLE_AUDIO */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/start.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

28400    /* This file contains the C startup code for Minix on Intel processors.
28401     * It cooperates with mpx.s to set up a good environment for main().
28402     *
28403     * This code runs in real mode for a 16 bit kernel and may have to switch
28404     * to protected mode for a 286.
28405     *
28406     * For a 32 bit kernel this already runs in protected mode, but the selectors
28407     * are still those given by the BIOS with interrupts disabled, so the
28408     * descriptors need to be reloaded and interrupt descriptors made.
28409     */
28410
28411    #include "kernel.h"
28412    #include <stdlib.h>
28413    #include <minix/boot.h>
28414    #include "protect.h"
28415
28416    PRIVATE char k_environ[256];    /* environment strings passed by loader */
28417
28418    FORWARD _PROTOTYPE( int k_atoi, (char *s) );
28419
28420
28421    /*===========================================================================*
28422     *                                cstart                                     *
28423     *===========================================================================*/
28424    PUBLIC void cstart(cs, ds, mcs, mds, parmoff, parmsize)
28425    U16_t cs, ds;                   /* Kernel code and data segment */
28426    U16_t mcs, mds;                 /* Monitor code and data segment */
28427    U16_t parmoff, parmsize;        /* boot parameters offset and length */
28428    {
28429    /* Perform system initializations prior to calling main(). */
28430
28431      register char *envp;
28432      phys_bytes mcode_base, mdata_base;
28433      unsigned mon_start;
28434
28435      /* Record where the kernel and the monitor are. */
28436      code_base = seg2phys(cs);
28437      data_base = seg2phys(ds);
28438      mcode_base = seg2phys(mcs);
28439      mdata_base = seg2phys(mds);
28440
28441      /* Initialize protected mode descriptors. */
28442      prot_init();
28443
28444      /* Copy the boot parameters to kernel memory. */
28445      if (parmsize > sizeof k_environ - 2) parmsize = sizeof k_environ - 2;
28446      phys_copy(mdata_base + parmoff, vir2phys(k_environ), (phys_bytes) parmsize);
28447
28448      /* Convert important boot environment variables. */
28449      boot_parameters.bp_rootdev = k_atoi(k_getenv("rootdev"));
28450      boot_parameters.bp_ramimagedev = k_atoi(k_getenv("ramimagedev"));
28451      boot_parameters.bp_ramsize = k_atoi(k_getenv("ramsize"));
28452      boot_parameters.bp_processor = k_atoi(k_getenv("processor"));
28453
28454      /* Type of VDU: */
```

```
28455     envp = k_getenv("video");
28456     if (strcmp(envp, "ega") == 0) ega = TRUE;
28457     if (strcmp(envp, "vga") == 0) vga = ega = TRUE;
28458
28459     /* Memory sizes: */
28460     low_memsize = k_atoi(k_getenv("memsize"));
28461     ext_memsize = k_atoi(k_getenv("emssize"));
28462
28463     /* Processor? */
28464     processor = boot_parameters.bp_processor;     /* 86, 186, 286, 386, ... */
28465
28466     /* XT, AT or MCA bus? */
28467     envp = k_getenv("bus");
28468     if (envp == NIL_PTR || strcmp(envp, "at") == 0) {
28469           pc_at = TRUE;
28470     } else
28471     if (strcmp(envp, "mca") == 0) {
28472           pc_at = ps_mca = TRUE;
28473     }
28474
28475     /* Decide if mode is protected. */
28476   #if _WORD_SIZE == 2
28477     protected_mode = processor >= 286;
28478   #endif
28479
28480     /* Is there a monitor to return to?  If so then keep it safe. */
28481     if (!protected_mode) mon_return = 0;
28482     mon_start = mcode_base / 1024;
28483     if (mon_return && low_memsize > mon_start) low_memsize = mon_start;
28484
28485     /* Return to assembler code to switch to protected mode (if 286), reload
28486      * selectors and call main().
28487      */
28488   }


28491   /*===========================================================================*
28492    *                                k_atoi                                     *
28493    *===========================================================================*/
28494   PRIVATE int k_atoi(s)
28495   register char *s;
28496   {
28497   /* Convert string to integer. */
28498
28499     return strtol(s, (char **) NULL, 10);
28500   }


28503   /*===========================================================================*
28504    *                                k_getenv                                   *
28505    *===========================================================================*/
28506   PUBLIC char *k_getenv(name)
28507   char *name;
28508   {
28509   /* Get environment value - kernel version of getenv to avoid setting up the
28510    * usual environment array.
28511    */
28512
28513     register char *namep;
28514     register char *envp;
```

```
28515
28516      for (envp = k_environ; *envp != 0;) {
28517            for (namep = name; *namep != 0 && *namep == *envp; namep++, envp++)
28518                  ;
28519            if (*namep == '\0' && *envp == '=') return(envp + 1);
28520            while (*envp++ != 0)
28521                  ;
28522      }
28523      return(NIL_PTR);
28524    }
```

++
src/kernel/system.c
++

```
28600    /* This task handles the interface between file system and kernel as well as
28601     * between memory manager and kernel.  System services are obtained by sending
28602     * sys_task() a message specifying what is needed.  To make life easier for
28603     * MM and FS, a library is provided with routines whose names are of the
28604     * form sys_xxx, e.g. sys_xit sends the SYS_XIT message to sys_task.  The
28605     * message types and parameters are:
28606     *
28607     *   SYS_FORK    informs kernel that a process has forked
28608     *   SYS_NEWMAP  allows MM to set up a process memory map
28609     *   SYS_GETMAP  allows MM to get a process' memory map
28610     *   SYS_EXEC    sets program counter and stack pointer after EXEC
28611     *   SYS_XIT     informs kernel that a process has exited
28612     *   SYS_GETSP   caller wants to read out some process' stack pointer
28613     *   SYS_TIMES   caller wants to get accounting times for a process
28614     *   SYS_ABORT   MM or FS cannot go on; abort MINIX
28615     *   SYS_FRESH   start with a fresh process image during EXEC (68000 only)
28616     *   SYS_SENDSIG send a signal to a process (POSIX style)
28617     *   SYS_SIGRETURN complete POSIX-style signalling
28618     *   SYS_KILL    cause a signal to be sent via MM
28619     *   SYS_ENDSIG  finish up after SYS_KILL-type signal
28620     *   SYS_COPY    request a block of data to be copied between processes
28621     *   SYS_VCOPY   request a series of data blocks to be copied between procs
28622     *   SYS_GBOOT   copies the boot parameters to a process
28623     *   SYS_MEM     returns the next free chunk of physical memory
28624     *   SYS_UMAP    compute the physical address for a given virtual address
28625     *   SYS_TRACE   request a trace operation
28626     *
28627     * Message types and parameters:
28628     *
28629     *    m_type       PROC1     PROC2      PID     MEM_PTR
28630     * ------------------------------------------------------
28631     * | SYS_FORK   | parent  |  child  |   pid   |         |
28632     * |------------+---------+---------+---------+---------|
28633     * | SYS_NEWMAP | proc nr |         |         | map ptr |
28634     * |------------+---------+---------+---------+---------|
28635     * | SYS_EXEC   | proc nr | traced  | new sp  |         |
28636     * |------------+---------+---------+---------+---------|
28637     * | SYS_XIT    | parent  | exitee  |         |         |
28638     * |------------+---------+---------+---------+---------|
28639     * | SYS_GETSP  | proc nr |         |         |         |
```

```
28640     * |------------+---------+---------+---------+---------|
28641     * | SYS_TIMES  | proc nr |         | buf ptr |         |
28642     * |------------+---------+---------+---------+---------|
28643     * | SYS_ABORT  |         |         |         |         |
28644     * |------------+---------+---------+---------+---------|
28645     * | SYS_FRESH  | proc nr | data_cl |         |         |
28646     * |------------+---------+---------+---------+---------|
28647     * | SYS_GBOOT  | proc nr |         |         | bootptr |
28648     * |------------+---------+---------+---------+---------|
28649     * | SYS_GETMAP | proc nr |         |         | map ptr |
28650     * -----------------------------------------------------
28651     *
28652     *    m_type          m1_i1     m1_i2     m1_i3       m1_p1
28653     * ---------------+---------+---------+---------+-------------
28654     * | SYS_VCOPY     |  src p  |  dst p  | vec siz | vc addr     |
28655     * |---------------+---------+---------+---------+-------------|
28656     * | SYS_SENDSIG   | proc nr |         |         | smp         |
28657     * |---------------+---------+---------+---------+-------------|
28658     * | SYS_SIGRETURN | proc nr |         |         | scp         |
28659     * |---------------+---------+---------+---------+-------------|
28660     * | SYS_ENDSIG    | proc nr |         |         |             |
28661     * ---------------------------------------------------------------
28662     *
28663     *    m_type       m2_i1     m2_i2     m2_l1     m2_l2
28664     * ------------------------------------------------------
28665     * | SYS_TRACE  | proc_nr | request |  addr   |  data   |
28666     * ------------------------------------------------------
28667     *
28668     *
28669     *    m_type       m6_i1     m6_i2     m6_i3     m6_f1
28670     * ------------------------------------------------------
28671     * | SYS_KILL   | proc_nr  |  sig    |         |         |
28672     * ------------------------------------------------------
28673     *
28674     *
28675     *    m_type      m5_c1   m5_i1    m5_l1   m5_c2   m5_i2    m5_l2   m5_l3
28676     * ------------------------------------------------------------------------
28677     * | SYS_COPY   |src seg|src proc|src vir|dst seg|dst proc|dst vir| byte ct |
28678     * ------------------------------------------------------------------------
28679     * | SYS_UMAP   |  seg  |proc nr |vir adr|       |        |       | byte ct |
28680     * ------------------------------------------------------------------------
28681     *
28682     *
28683     *    m_type      m1_i1      m1_i2      m1_i3
28684     * |------------+----------+----------+----------
28685     * | SYS_MEM    | mem base | mem size | tot mem |
28686     * -------------------------------------------
28687     *
28688     * In addition to the main sys_task() entry point, there are 5 other minor
28689     * entry points:
28690     *   cause_sig: take action to cause a signal to occur, sooner or later
28691     *   inform:    tell MM about pending signals
28692     *   numap:     umap D segment starting from process number instead of pointer
28693     *   umap:      compute the physical address for a given virtual address
28694     *   alloc_segments: allocate segments for 8088 or higher processor
28695     */
28696
28697    #include "kernel.h"
28698    #include <signal.h>
28699    #include <unistd.h>
```

```
28700    #include <sys/sigcontext.h>
28701    #include <sys/ptrace.h>
28702    #include <minix/boot.h>
28703    #include <minix/callnr.h>
28704    #include <minix/com.h>
28705    #include "proc.h"
28706    #if (CHIP == INTEL)
28707    #include "protect.h"
28708    #endif
28709
28710    /* PSW masks. */
28711    #define IF_MASK 0x00000200
28712    #define IOPL_MASK 0x003000
28713
28714    PRIVATE message m;
28715
28716    FORWARD _PROTOTYPE( int do_abort, (message *m_ptr) );
28717    FORWARD _PROTOTYPE( int do_copy, (message *m_ptr) );
28718    FORWARD _PROTOTYPE( int do_exec, (message *m_ptr) );
28719    FORWARD _PROTOTYPE( int do_fork, (message *m_ptr) );
28720    FORWARD _PROTOTYPE( int do_gboot, (message *m_ptr) );
28721    FORWARD _PROTOTYPE( int do_getsp, (message *m_ptr) );
28722    FORWARD _PROTOTYPE( int do_kill, (message *m_ptr) );
28723    FORWARD _PROTOTYPE( int do_mem, (message *m_ptr) );
28724    FORWARD _PROTOTYPE( int do_newmap, (message *m_ptr) );
28725    FORWARD _PROTOTYPE( int do_sendsig, (message *m_ptr) );
28726    FORWARD _PROTOTYPE( int do_sigreturn, (message *m_ptr) );
28727    FORWARD _PROTOTYPE( int do_endsig, (message *m_ptr) );
28728    FORWARD _PROTOTYPE( int do_times, (message *m_ptr) );
28729    FORWARD _PROTOTYPE( int do_trace, (message *m_ptr) );
28730    FORWARD _PROTOTYPE( int do_umap, (message *m_ptr) );
28731    FORWARD _PROTOTYPE( int do_xit, (message *m_ptr) );
28732    FORWARD _PROTOTYPE( int do_vcopy, (message *m_ptr) );
28733    FORWARD _PROTOTYPE( int do_getmap, (message *m_ptr) );
28734
28735    #if (SHADOWING == 1)
28736    FORWARD _PROTOTYPE( int do_fresh, (message *m_ptr) );
28737    #endif
28738
28739    /*===========================================================================*
28740     *                              sys_task                                     *
28741     *===========================================================================*/
28742    PUBLIC void sys_task()
28743    {
28744    /* Main entry point of sys_task.  Get the message and dispatch on type. */
28745
28746      register int r;
28747
28748      while (TRUE) {
28749            receive(ANY, &m);
28750
28751            switch (m.m_type) {     /* which system call */
28752                case SYS_FORK:      r = do_fork(&m);        break;
28753                case SYS_NEWMAP:    r = do_newmap(&m);      break;
28754                case SYS_GETMAP:    r = do_getmap(&m);      break;
28755                case SYS_EXEC:      r = do_exec(&m);        break;
28756                case SYS_XIT:       r = do_xit(&m);         break;
28757                case SYS_GETSP:     r = do_getsp(&m);       break;
28758                case SYS_TIMES:     r = do_times(&m);       break;
28759                case SYS_ABORT:     r = do_abort(&m);       break;
```

```
28760   #if (SHADOWING == 1)
28761               case SYS_FRESH:     r = do_fresh(&m);       break;
28762   #endif
28763               case SYS_SENDSIG:   r = do_sendsig(&m);     break;
28764               case SYS_SIGRETURN: r = do_sigreturn(&m);   break;
28765               case SYS_KILL:      r = do_kill(&m);        break;
28766               case SYS_ENDSIG:    r = do_endsig(&m);      break;
28767               case SYS_COPY:      r = do_copy(&m);        break;
28768               case SYS_VCOPY:     r = do_vcopy(&m);       break;
28769               case SYS_GBOOT:     r = do_gboot(&m);       break;
28770               case SYS_MEM:       r = do_mem(&m);         break;
28771               case SYS_UMAP:      r = do_umap(&m);        break;
28772               case SYS_TRACE:     r = do_trace(&m);       break;
28773               default:            r = E_BAD_FCN;
28774           }
28775
28776           m.m_type = r;           /* 'r' reports status of call */
28777           send(m.m_source, &m);   /* send reply to caller */
28778     }
28779   }


28782   /*===========================================================================*
28783    *                                do_fork                                    *
28784    *===========================================================================*/
28785   PRIVATE int do_fork(m_ptr)
28786   register message *m_ptr;        /* pointer to request message */
28787   {
28788   /* Handle sys_fork().  m_ptr->PROC1 has forked.  The child is m_ptr->PROC2. */
28789
28790   #if (CHIP == INTEL)
28791     reg_t old_ldt_sel;
28792   #endif
28793     register struct proc *rpc;
28794     struct proc *rpp;
28795
28796     if (!isoksusern(m_ptr->PROC1) || !isoksusern(m_ptr->PROC2))
28797           return(E_BAD_PROC);
28798     rpp = proc_addr(m_ptr->PROC1);
28799     rpc = proc_addr(m_ptr->PROC2);
28800
28801     /* Copy parent 'proc' struct to child. */
28802   #if (CHIP == INTEL)
28803     old_ldt_sel = rpc->p_ldt_sel; /* stop this being obliterated by copy */
28804   #endif
28805
28806     *rpc = *rpp;                  /* copy 'proc' struct */
28807
28808   #if (CHIP == INTEL)
28809     rpc->p_ldt_sel = old_ldt_sel;
28810   #endif
28811     rpc->p_nr = m_ptr->PROC2;     /* this was obliterated by copy */
28812
28813   #if (SHADOWING == 0)
28814     rpc->p_flags |= NO_MAP;       /* inhibit the process from running */
28815   #endif
28816
28817     rpc->p_flags &= ~(PENDING | SIG_PENDING | P_STOP);
28818
28819     /* Only 1 in group should have PENDING, child does not inherit trace status*/
```

```
28820     sigemptyset(&rpc->p_pending);
28821     rpc->p_pendcount = 0;
28822     rpc->p_pid = m_ptr->PID;      /* install child's pid */
28823     rpc->p_reg.retreg = 0;        /* child sees pid = 0 to know it is child */
28824
28825     rpc->user_time = 0;           /* set all the accounting times to 0 */
28826     rpc->sys_time = 0;
28827     rpc->child_utime = 0;
28828     rpc->child_stime = 0;
28829
28830   #if (SHADOWING == 1)
28831     rpc->p_nflips = 0;
28832     mkshadow(rpp, (phys_clicks)m_ptr->m1_p1);     /* run child first */
28833   #endif
28834
28835     return(OK);
28836   }


28839   /*===========================================================================*
28840    *                                do_newmap                                  *
28841    *===========================================================================*/
28842   PRIVATE int do_newmap(m_ptr)
28843   message *m_ptr;                 /* pointer to request message */
28844   {
28845   /* Handle sys_newmap().  Fetch the memory map from MM. */
28846
28847     register struct proc *rp;
28848     phys_bytes src_phys;
28849     int caller;                   /* whose space has the new map (usually MM) */
28850     int k;                        /* process whose map is to be loaded */
28851     int old_flags;                /* value of flags before modification */
28852     struct mem_map *map_ptr;      /* virtual address of map inside caller (MM) */
28853
28854     /* Extract message parameters and copy new memory map from MM. */
28855     caller = m_ptr->m_source;
28856     k = m_ptr->PROC1;
28857     map_ptr = (struct mem_map *) m_ptr->MEM_PTR;
28858     if (!isokprocn(k)) return(E_BAD_PROC);
28859     rp = proc_addr(k);            /* ptr to entry of user getting new map */
28860
28861     /* Copy the map from MM. */
28862     src_phys = umap(proc_addr(caller), D, (vir_bytes) map_ptr, sizeof(rp->p_map));
28863     if (src_phys == 0) panic("bad call to sys_newmap", NO_NUM);
28864     phys_copy(src_phys, vir2phys(rp->p_map), (phys_bytes) sizeof(rp->p_map));
28865
28866   #if (SHADOWING == 0)
28867   #if (CHIP != M68000)
28868     alloc_segments(rp);
28869   #else
28870     pmmu_init_proc(rp);
28871   #endif
28872     old_flags = rp->p_flags;      /* save the previous value of the flags */
28873     rp->p_flags &= ~NO_MAP;
28874     if (old_flags != 0 && rp->p_flags == 0) lock_ready(rp);
28875   #endif
28876
28877     return(OK);
28878   }
```

```
28881   /*===========================================================================*
28882    *                                do_getmap                                  *
28883    *===========================================================================*/
28884   PRIVATE int do_getmap(m_ptr)
28885   message *m_ptr;                 /* pointer to request message */
28886   {
28887   /* Handle sys_getmap().  Report the memory map to MM. */
28888
28889     register struct proc *rp;
28890     phys_bytes dst_phys;
28891     int caller;                   /* where the map has to be stored */
28892     int k;                        /* process whose map is to be loaded */
28893     struct mem_map *map_ptr;      /* virtual address of map inside caller (MM) */
28894
28895     /* Extract message parameters and copy new memory map to MM. */
28896     caller = m_ptr->m_source;
28897     k = m_ptr->PROC1;
28898     map_ptr = (struct mem_map *) m_ptr->MEM_PTR;
28899
28900     if (!isokprocn(k))
28901           panic("do_getmap got bad proc: ", m_ptr->PROC1);
28902
28903     rp = proc_addr(k);            /* ptr to entry of the map */
28904
28905     /* Copy the map to MM. */
28906     dst_phys = umap(proc_addr(caller), D, (vir_bytes) map_ptr, sizeof(rp->p_map));
28907     if (dst_phys == 0) panic("bad call to sys_getmap", NO_NUM);
28908     phys_copy(vir2phys(rp->p_map), dst_phys, sizeof(rp->p_map));
28909
28910     return(OK);
28911   }


28914   /*===========================================================================*
28915    *                                do_exec                                    *
28916    *===========================================================================*/
28917   PRIVATE int do_exec(m_ptr)
28918   register message *m_ptr;        /* pointer to request message */
28919   {
28920   /* Handle sys_exec().  A process has done a successful EXEC. Patch it up. */
28921
28922     register struct proc *rp;
28923     reg_t sp;                     /* new sp */
28924     phys_bytes phys_name;
28925     char *np;
28926   #define NLEN (sizeof(rp->p_name)-1)
28927
28928     if (!isoksusern(m_ptr->PROC1)) return E_BAD_PROC;
28929     /* PROC2 field is used as flag to indicate process is being traced */
28930     if (m_ptr->PROC2) cause_sig(m_ptr->PROC1, SIGTRAP);
28931     sp = (reg_t) m_ptr->STACK_PTR;
28932     rp = proc_addr(m_ptr->PROC1);
28933     rp->p_reg.sp = sp;            /* set the stack pointer */
28934   #if (CHIP == M68000)
28935     rp->p_splow = sp;             /* set the stack pointer low water */
28936   #ifdef FPP
28937     /* Initialize fpp for this process */
28938     fpp_new_state(rp);
28939   #endif
```

```
28940   #endif
28941     rp->p_reg.pc = (reg_t) m_ptr->IP_PTR; /* set pc */
28942     rp->p_alarm = 0;              /* reset alarm timer */
28943     rp->p_flags &= ~RECEIVING;    /* MM does not reply to EXEC call */
28944     if (rp->p_flags == 0) lock_ready(rp);
28945
28946     /* Save command name for debugging, ps(1) output, etc. */
28947     phys_name = numap(m_ptr->m_source, (vir_bytes) m_ptr->NAME_PTR,
28948                                                     (vir_bytes) NLEN);
28949     if (phys_name != 0) {
28950           phys_copy(phys_name, vir2phys(rp->p_name), (phys_bytes) NLEN);
28951           for (np = rp->p_name; (*np & BYTE) >= ' '; np++) {}
28952           *np = 0;
28953     }
28954     return(OK);
28955   }


28958   /*===========================================================================*
28959    *                              do_xit                                       *
28960    *===========================================================================*/
28961   PRIVATE int do_xit(m_ptr)
28962   message *m_ptr;               /* pointer to request message */
28963   {
28964   /* Handle sys_xit().  A process has exited. */
28965
28966     register struct proc *rp, *rc;
28967     struct proc *np, *xp;
28968     int parent;                   /* number of exiting proc's parent */
28969     int proc_nr;                  /* number of process doing the exit */
28970     phys_clicks base, size;
28971
28972     parent = m_ptr->PROC1;        /* slot number of parent process */
28973     proc_nr = m_ptr->PROC2;       /* slot number of exiting process */
28974     if (!isoksusern(parent) || !isoksusern(proc_nr)) return(E_BAD_PROC);
28975     rp = proc_addr(parent);
28976     rc = proc_addr(proc_nr);
28977     lock();
28978     rp->child_utime += rc->user_time + rc->child_utime;   /* accum child times */
28979     rp->child_stime += rc->sys_time + rc->child_stime;
28980     unlock();
28981     rc->p_alarm = 0;              /* turn off alarm timer */
28982     if (rc->p_flags == 0) lock_unready(rc);
28983
28984   #if (SHADOWING == 1)
28985     rmshadow(rc, &base, &size);
28986     m_ptr->m1_i1 = (int)base;
28987     m_ptr->m1_i2 = (int)size;
28988   #endif
28989
28990     strcpy(rc->p_name, "<noname>");       /* process no longer has a name */
28991
28992     /* If the process being terminated happens to be queued trying to send a
28993      * message (i.e., the process was killed by a signal, rather than it doing an
28994      * EXIT), then it must be removed from the message queues.
28995      */
28996     if (rc->p_flags & SENDING) {
28997           /* Check all proc slots to see if the exiting process is queued. */
28998           for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++) {
28999                   if (rp->p_callerq == NIL_PROC) continue;
```

```
29000                   if (rp->p_callerq == rc) {
29001                           /* Exiting process is on front of this queue. */
29002                           rp->p_callerq = rc->p_sendlink;
29003                           break;
29004                   } else {
29005                           /* See if exiting process is in middle of queue. */
29006                           np = rp->p_callerq;
29007                           while ( ( xp = np->p_sendlink) != NIL_PROC)
29008                                   if (xp == rc) {
29009                                           np->p_sendlink = xp->p_sendlink;
29010                                           break;
29011                                   } else {
29012                                           np = xp;
29013                                   }
29014                   }
29015           }
29016     }
29017   #if (CHIP == M68000) && (SHADOWING == 0)
29018     pmmu_delete(rc);      /* we're done remove tables */
29019   #endif
29020
29021     if (rc->p_flags & PENDING) --sig_procs;
29022     sigemptyset(&rc->p_pending);
29023     rc->p_pendcount = 0;
29024     rc->p_flags = P_SLOT_FREE;
29025     return(OK);
29026   }


29029   /*===========================================================================*
29030    *                                do_getsp                                   *
29031    *===========================================================================*/
29032   PRIVATE int do_getsp(m_ptr)
29033   register message *m_ptr;        /* pointer to request message */
29034   {
29035   /* Handle sys_getsp().  MM wants to know what sp is. */
29036
29037     register struct proc *rp;
29038
29039     if (!isoksusern(m_ptr->PROC1)) return(E_BAD_PROC);
29040     rp = proc_addr(m_ptr->PROC1);
29041     m_ptr->STACK_PTR = (char *) rp->p_reg.sp;     /* return sp here (bad type) */
29042     return(OK);
29043   }


29046   /*===========================================================================*
29047    *                                do_times                                   *
29048    *===========================================================================*/
29049   PRIVATE int do_times(m_ptr)
29050   register message *m_ptr;        /* pointer to request message */
29051   {
29052   /* Handle sys_times().  Retrieve the accounting information. */
29053
29054     register struct proc *rp;
29055
29056     if (!isoksusern(m_ptr->PROC1)) return E_BAD_PROC;
29057     rp = proc_addr(m_ptr->PROC1);
29058
29059     /* Insert the times needed by the TIMES system call in the message. */
```

```
29060     lock();                       /* halt the volatile time counters in rp */
29061     m_ptr->USER_TIME   = rp->user_time;
29062     m_ptr->SYSTEM_TIME = rp->sys_time;
29063     unlock();
29064     m_ptr->CHILD_UTIME = rp->child_utime;
29065     m_ptr->CHILD_STIME = rp->child_stime;
29066     m_ptr->BOOT_TICKS  = get_uptime();
29067     return(OK);
29068   }


29071   /*===========================================================================*
29072    *                              do_abort                                     *
29073    *===========================================================================*/
29074   PRIVATE int do_abort(m_ptr)
29075   message *m_ptr;                 /* pointer to request message */
29076   {
29077   /* Handle sys_abort.  MINIX is unable to continue.  Terminate operation. */
29078     char monitor_code[64];
29079     phys_bytes src_phys;
29080
29081     if (m_ptr->m1_i1 == RBT_MONITOR) {
29082           /* The monitor is to run user specified instructions. */
29083           src_phys = numap(m_ptr->m_source, (vir_bytes) m_ptr->m1_p1,
29084                                       (vir_bytes) sizeof(monitor_code));
29085           if (src_phys == 0) panic("bad monitor code from", m_ptr->m_source);
29086           phys_copy(src_phys, vir2phys(monitor_code),
29087                                       (phys_bytes) sizeof(monitor_code));
29088           reboot_code = vir2phys(monitor_code);
29089     }
29090     wreboot(m_ptr->m1_i1);
29091     return(OK);                   /* pro-forma (really EDISASTER) */
29092   }


29095   #if (SHADOWING == 1)
29096   /*===========================================================================*
29097    *                              do_fresh                                     *
29098    *===========================================================================*/
29099   PRIVATE int do_fresh(m_ptr)     /* for 68000 only */
29100   message *m_ptr;                 /* pointer to request message */
29101   {
29102   /* Handle sys_fresh.  Start with fresh process image during EXEC. */
29103
29104     register struct proc *p;
29105     int proc_nr;                  /* number of process doing the exec */
29106     phys_clicks base, size;
29107     phys_clicks c1, nc;
29108
29109     proc_nr = m_ptr->PROC1;       /* slot number of exec-ing process */
29110     if (!isokprocn(proc_nr)) return(E_BAD_PROC);
29111     p = proc_addr(proc_nr);
29112     rmshadow(p, &base, &size);
29113     do_newmap(m_ptr);
29114     c1 = p->p_map[D].mem_phys;
29115     nc = p->p_map[S].mem_phys - p->p_map[D].mem_phys + p->p_map[S].mem_len;
29116     c1 += m_ptr->m1_i2;
29117     nc -= m_ptr->m1_i2;
29118     zeroclicks(c1, nc);
29119     m_ptr->m1_i1 = (int)base;
```

```
29120       m_ptr->m1_i2 = (int)size;
29121       return(OK);
29122     }
29123     #endif /* (SHADOWING == 1) */
29124
29125
29126     /*===========================================================================*
29127      *                              do_sendsig                                   *
29128      *===========================================================================*/
29129     PRIVATE int do_sendsig(m_ptr)
29130     message *m_ptr;                 /* pointer to request message */
29131     {
29132     /* Handle sys_sendsig, POSIX-style signal */
29133
29134       struct sigmsg smsg;
29135       register struct proc *rp;
29136       phys_bytes src_phys, dst_phys;
29137       struct sigcontext sc, *scp;
29138       struct sigframe fr, *frp;
29139
29140       if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);
29141       rp = proc_addr(m_ptr->PROC1);
29142
29143       /* Get the sigmsg structure into our address space.  */
29144       src_phys = umap(proc_addr(MM_PROC_NR), D, (vir_bytes) m_ptr->SIG_CTXT_PTR,
29145                       (vir_bytes) sizeof(struct sigmsg));
29146       if (src_phys == 0)
29147             panic("do_sendsig can't signal: bad sigmsg address from MM", NO_NUM);
29148       phys_copy(src_phys, vir2phys(&smsg), (phys_bytes) sizeof(struct sigmsg));
29149
29150       /* Compute the usr stack pointer value where sigcontext will be stored. */
29151       scp = (struct sigcontext *) smsg.sm_stkptr - 1;
29152
29153       /* Copy the registers to the sigcontext structure. */
29154       memcpy(&sc.sc_regs, &rp->p_reg, sizeof(struct sigregs));
29155
29156       /* Finish the sigcontext initialization. */
29157       sc.sc_flags = SC_SIGCONTEXT;
29158
29159       sc.sc_mask = smsg.sm_mask;
29160
29161       /* Copy the sigcontext structure to the user's stack. */
29162       dst_phys = umap(rp, D, (vir_bytes) scp,
29163                       (vir_bytes) sizeof(struct sigcontext));
29164       if (dst_phys == 0) return(EFAULT);
29165       phys_copy(vir2phys(&sc), dst_phys, (phys_bytes) sizeof(struct sigcontext));
29166
29167       /* Initialize the sigframe structure. */
29168       frp = (struct sigframe *) scp - 1;
29169       fr.sf_scpcopy = scp;
29170       fr.sf_retadr2= (void (*)()) rp->p_reg.pc;
29171       fr.sf_fp = rp->p_reg.fp;
29172       rp->p_reg.fp = (reg_t) &frp->sf_fp;
29173       fr.sf_scp = scp;
29174       fr.sf_code = 0;       /* XXX - should be used for type of FP exception */
29175       fr.sf_signo = smsg.sm_signo;
29176       fr.sf_retadr = (void (*)()) smsg.sm_sigreturn;
29177
29178       /* Copy the sigframe structure to the user's stack. */
29179       dst_phys = umap(rp, D, (vir_bytes) frp, (vir_bytes) sizeof(struct sigframe));
```

```
29180     if (dst_phys == 0) return(EFAULT);
29181     phys_copy(vir2phys(&fr), dst_phys, (phys_bytes) sizeof(struct sigframe));
29182
29183     /* Reset user registers to execute the signal handler. */
29184     rp->p_reg.sp = (reg_t) frp;
29185     rp->p_reg.pc = (reg_t) smsg.sm_sighandler;
29186
29187     return(OK);
29188   }

29190   /*===========================================================================*
29191    *                              do_sigreturn                                 *
29192    *===========================================================================*/
29193   PRIVATE int do_sigreturn(m_ptr)
29194   register message *m_ptr;
29195   {
29196   /* POSIX style signals require sys_sigreturn to put things in order before the
29197    * signalled process can resume execution
29198    */
29199
29200     struct sigcontext sc;
29201     register struct proc *rp;
29202     phys_bytes src_phys;
29203
29204     if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);
29205     rp = proc_addr(m_ptr->PROC1);
29206
29207     /* Copy in the sigcontext structure. */
29208     src_phys = umap(rp, D, (vir_bytes) m_ptr->SIG_CTXT_PTR,
29209                     (vir_bytes) sizeof(struct sigcontext));
29210     if (src_phys == 0) return(EFAULT);
29211     phys_copy(src_phys, vir2phys(&sc), (phys_bytes) sizeof(struct sigcontext));
29212
29213     /* Make sure that this is not just a jmp_buf. */
29214     if ((sc.sc_flags & SC_SIGCONTEXT) == 0) return(EINVAL);
29215
29216     /* Fix up only certain key registers if the compiler doesn't use
29217      * register variables within functions containing setjmp.
29218      */
29219     if (sc.sc_flags & SC_NOREGLOCALS) {
29220           rp->p_reg.retreg = sc.sc_retreg;
29221           rp->p_reg.fp = sc.sc_fp;
29222           rp->p_reg.pc = sc.sc_pc;
29223           rp->p_reg.sp = sc.sc_sp;
29224           return (OK);
29225     }
29226     sc.sc_psw  = rp->p_reg.psw;
29227
29228   #if (CHIP == INTEL)
29229     /* Don't panic kernel if user gave bad selectors. */
29230     sc.sc_cs = rp->p_reg.cs;
29231     sc.sc_ds = rp->p_reg.ds;
29232     sc.sc_es = rp->p_reg.es;
29233   #if _WORD_SIZE == 4
29234     sc.sc_fs = rp->p_reg.fs;
29235     sc.sc_gs = rp->p_reg.gs;
29236   #endif
29237   #endif
29238
29239     /* Restore the registers. */
```

```
29240     memcpy(&rp->p_reg, (char *)&sc.sc_regs, sizeof(struct sigregs));
29241
29242     return(OK);
29243   }

29245   /*===========================================================================*
29246    *                              do_kill                                      *
29247    *===========================================================================*/
29248   PRIVATE int do_kill(m_ptr)
29249   register message *m_ptr;        /* pointer to request message */
29250   {
29251   /* Handle sys_kill(). Cause a signal to be sent to a process via MM.
29252    * Note that this has nothing to do with the kill (2) system call, this
29253    * is how the FS (and possibly other servers) get access to cause_sig to
29254    * send a KSIG message to MM
29255    */
29256
29257     if (!isokusern(m_ptr->PR)) return(E_BAD_PROC);
29258     cause_sig(m_ptr->PR, m_ptr->SIGNUM);
29259     return(OK);
29260   }


29263   /*===========================================================================*
29264    *                            do_endsig                                      *
29265    *===========================================================================*/
29266   PRIVATE int do_endsig(m_ptr)
29267   register message *m_ptr;        /* pointer to request message */
29268   {
29269   /* Finish up after a KSIG-type signal, caused by a SYS_KILL message or a call
29270    * to cause_sig by a task
29271    */
29272
29273     register struct proc *rp;
29274
29275     if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);
29276     rp = proc_addr(m_ptr->PROC1);
29277
29278     /* MM has finished one KSIG. */
29279     if (rp->p_pendcount != 0 && --rp->p_pendcount == 0
29280         && (rp->p_flags &= ~SIG_PENDING) == 0)
29281           lock_ready(rp);
29282     return(OK);
29283   }

29285   /*===========================================================================*
29286    *                              do_copy                                      *
29287    *===========================================================================*/
29288   PRIVATE int do_copy(m_ptr)
29289   register message *m_ptr;        /* pointer to request message */
29290   {
29291   /* Handle sys_copy().  Copy data for MM or FS. */
29292
29293     int src_proc, dst_proc, src_space, dst_space;
29294     vir_bytes src_vir, dst_vir;
29295     phys_bytes src_phys, dst_phys, bytes;
29296
29297     /* Dismember the command message. */
29298     src_proc = m_ptr->SRC_PROC_NR;
29299     dst_proc = m_ptr->DST_PROC_NR;
```

```
29300     src_space = m_ptr->SRC_SPACE;
29301     dst_space = m_ptr->DST_SPACE;
29302     src_vir = (vir_bytes) m_ptr->SRC_BUFFER;
29303     dst_vir = (vir_bytes) m_ptr->DST_BUFFER;
29304     bytes = (phys_bytes) m_ptr->COPY_BYTES;
29305
29306     /* Compute the source and destination addresses and do the copy. */
29307   #if (SHADOWING == 0)
29308     if (src_proc == ABS)
29309           src_phys = (phys_bytes) m_ptr->SRC_BUFFER;
29310     else {
29311           if (bytes != (vir_bytes) bytes)
29312                   /* This would happen for 64K segments and 16-bit vir_bytes.
29313                    * It would happen a lot for do_fork except MM uses ABS
29314                    * copies for that case.
29315                    */
29316                   panic("overflow in count in do_copy", NO_NUM);
29317   #endif
29318
29319           src_phys = umap(proc_addr(src_proc), src_space, src_vir,
29320                           (vir_bytes) bytes);
29321   #if (SHADOWING == 0)
29322           }
29323   #endif
29324
29325   #if (SHADOWING == 0)
29326     if (dst_proc == ABS)
29327           dst_phys = (phys_bytes) m_ptr->DST_BUFFER;
29328     else
29329   #endif
29330           dst_phys = umap(proc_addr(dst_proc), dst_space, dst_vir,
29331                           (vir_bytes) bytes);
29332
29333     if (src_phys == 0 || dst_phys == 0) return(EFAULT);
29334     phys_copy(src_phys, dst_phys, bytes);
29335     return(OK);
29336   }


29339   /*===========================================================================*
29340    *                               do_vcopy                                    *
29341    *===========================================================================*/
29342   PRIVATE int do_vcopy(m_ptr)
29343   register message *m_ptr;        /* pointer to request message */
29344   {
29345   /* Handle sys_vcopy(). Copy multiple blocks of memory */
29346
29347     int src_proc, dst_proc, vect_s, i;
29348     vir_bytes src_vir, dst_vir, vect_addr;
29349     phys_bytes src_phys, dst_phys, bytes;
29350     cpvec_t cpvec_table[CPVEC_NR];
29351
29352     /* Dismember the command message. */
29353     src_proc = m_ptr->m1_i1;
29354     dst_proc = m_ptr->m1_i2;
29355     vect_s = m_ptr->m1_i3;
29356     vect_addr = (vir_bytes)m_ptr->m1_p1;
29357
29358     if (vect_s > CPVEC_NR) return EDOM;
29359
```

```
29360     src_phys= numap (m_ptr->m_source, vect_addr, vect_s * sizeof(cpvec_t));
29361     if (!src_phys) return EFAULT;
29362     phys_copy(src_phys, vir2phys(cpvec_table),
29363                                 (phys_bytes) (vect_s * sizeof(cpvec_t)));
29364
29365     for (i = 0; i < vect_s; i++) {
29366           src_vir= cpvec_table[i].cpv_src;
29367           dst_vir= cpvec_table[i].cpv_dst;
29368           bytes= cpvec_table[i].cpv_size;
29369           src_phys = numap(src_proc,src_vir,(vir_bytes)bytes);
29370           dst_phys = numap(dst_proc,dst_vir,(vir_bytes)bytes);
29371           if (src_phys == 0 || dst_phys == 0) return(EFAULT);
29372           phys_copy(src_phys, dst_phys, bytes);
29373     }
29374     return(OK);
29375   }


29378   /*============================================================================*
29379    *                               do_gboot                                     *
29380    *============================================================================*/
29381   PUBLIC struct bparam_s boot_parameters;
29382
29383   PRIVATE int do_gboot(m_ptr)
29384   message *m_ptr;                 /* pointer to request message */
29385   {
29386   /* Copy the boot parameters.  Normally only called during fs init. */
29387
29388     phys_bytes dst_phys;
29389
29390     dst_phys = umap(proc_addr(m_ptr->PROC1), D, (vir_bytes) m_ptr->MEM_PTR,
29391                                 (vir_bytes) sizeof(boot_parameters));
29392     if (dst_phys == 0) panic("bad call to SYS_GBOOT", NO_NUM);
29393     phys_copy(vir2phys(&boot_parameters), dst_phys,
29394                                 (phys_bytes) sizeof(boot_parameters));
29395     return(OK);
29396   }


29399   /*============================================================================*
29400    *                               do_mem                                       *
29401    *============================================================================*/
29402   PRIVATE int do_mem(m_ptr)
29403   register message *m_ptr;        /* pointer to request message */
29404   {
29405   /* Return the base and size of the next chunk of memory. */
29406
29407     struct memory *memp;
29408
29409     for (memp = mem; memp < &mem[NR_MEMS]; ++memp) {
29410           m_ptr->m1_i1 = memp->base;
29411           m_ptr->m1_i2 = memp->size;
29412           m_ptr->m1_i3 = tot_mem_size;
29413           memp->size = 0;
29414           if (m_ptr->m1_i2 != 0) break;           /* found a chunk */
29415     }
29416     return(OK);
29417   }
```

```
29420   /*===========================================================================*
29421    *                              do_umap                                      *
29422    *===========================================================================*/
29423   PRIVATE int do_umap(m_ptr)
29424   register message *m_ptr;        /* pointer to request message */
29425   {
29426   /* Same as umap(), for non-kernel processes. */
29427
29428     m_ptr->SRC_BUFFER = umap(proc_addr((int) m_ptr->SRC_PROC_NR),
29429                              (int) m_ptr->SRC_SPACE,
29430                              (vir_bytes) m_ptr->SRC_BUFFER,
29431                              (vir_bytes) m_ptr->COPY_BYTES);
29432     return(OK);
29433   }


29436   /*===========================================================================*
29437    *                              do_trace                                     *
29438    *===========================================================================*/
29439   #define TR_PROCNR       (m_ptr->m2_i1)
29440   #define TR_REQUEST      (m_ptr->m2_i2)
29441   #define TR_ADDR         ((vir_bytes) m_ptr->m2_l1)
29442   #define TR_DATA         (m_ptr->m2_l2)
29443   #define TR_VLSIZE       ((vir_bytes) sizeof(long))
29444
29445   PRIVATE int do_trace(m_ptr)
29446   register message *m_ptr;
29447   {
29448   /* Handle the debugging commands supported by the ptrace system call
29449    * The commands are:
29450    * T_STOP       stop the process
29451    * T_OK         enable tracing by parent for this process
29452    * T_GETINS     return value from instruction space
29453    * T_GETDATA    return value from data space
29454    * T_GETUSER    return value from user process table
29455    * T_SETINS     set value from instruction space
29456    * T_SETDATA    set value from data space
29457    * T_SETUSER    set value in user process table
29458    * T_RESUME     resume execution
29459    * T_EXIT       exit
29460    * T_STEP       set trace bit
29461    *
29462    * The T_OK and T_EXIT commands are handled completely by the memory manager,
29463    * all others come here.
29464    */
29465
29466     register struct proc *rp;
29467     phys_bytes src, dst;
29468     int i;
29469
29470     rp = proc_addr(TR_PROCNR);
29471     if (rp->p_flags & P_SLOT_FREE) return(EIO);
29472     switch (TR_REQUEST) {
29473     case T_STOP:                    /* stop process */
29474           if (rp->p_flags == 0) lock_unready(rp);
29475           rp->p_flags |= P_STOP;
29476           rp->p_reg.psw &= ~TRACEBIT;     /* clear trace bit */
29477           return(OK);
29478
29479     case T_GETINS:                /* return value from instruction space */
```

```
29480           if (rp->p_map[T].mem_len != 0) {
29481                   if ((src = umap(rp, T, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
29482                   phys_copy(src, vir2phys(&TR_DATA), (phys_bytes) sizeof(long));
29483                   break;
29484           }
29485           /* Text space is actually data space - fall through. */
29486
29487       case T_GETDATA:               /* return value from data space */
29488           if ((src = umap(rp, D, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
29489           phys_copy(src, vir2phys(&TR_DATA), (phys_bytes) sizeof(long));
29490           break;
29491
29492       case T_GETUSER:               /* return value from process table */
29493           if ((TR_ADDR & (sizeof(long) - 1)) != 0 ||
29494               TR_ADDR > sizeof(struct proc) - sizeof(long))
29495                   return(EIO);
29496           TR_DATA = *(long *) ((char *) rp + (int) TR_ADDR);
29497           break;
29498
29499       case T_SETINS:                /* set value in instruction space */
29500           if (rp->p_map[T].mem_len != 0) {
29501                   if ((dst = umap(rp, T, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
29502                   phys_copy(vir2phys(&TR_DATA), dst, (phys_bytes) sizeof(long));
29503                   TR_DATA = 0;
29504                   break;
29505           }
29506           /* Text space is actually data space - fall through. */
29507
29508       case T_SETDATA:                       /* set value in data space */
29509           if ((dst = umap(rp, D, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
29510           phys_copy(vir2phys(&TR_DATA), dst, (phys_bytes) sizeof(long));
29511           TR_DATA = 0;
29512           break;
29513
29514       case T_SETUSER:                       /* set value in process table */
29515           if ((TR_ADDR & (sizeof(reg_t) - 1)) != 0 ||
29516                TR_ADDR > sizeof(struct stackframe_s) - sizeof(reg_t))
29517                   return(EIO);
29518           i = (int) TR_ADDR;
29519   #if (CHIP == INTEL)
29520           /* Altering segment registers might crash the kernel when it
29521            * tries to load them prior to restarting a process, so do
29522            * not allow it.
29523            */
29524           if (i == (int) &((struct proc *) 0)->p_reg.cs ||
29525               i == (int) &((struct proc *) 0)->p_reg.ds ||
29526               i == (int) &((struct proc *) 0)->p_reg.es ||
29527   #if _WORD_SIZE == 4
29528               i == (int) &((struct proc *) 0)->p_reg.gs ||
29529               i == (int) &((struct proc *) 0)->p_reg.fs ||
29530   #endif
29531               i == (int) &((struct proc *) 0)->p_reg.ss)
29532                   return(EIO);
29533   #endif
29534           if (i == (int) &((struct proc *) 0)->p_reg.psw)
29535                   /* only selected bits are changeable */
29536                   SETPSW(rp, TR_DATA);
29537           else
29538                   *(reg_t *) ((char *) &rp->p_reg + i) = (reg_t) TR_DATA;
29539           TR_DATA = 0;
```

```
29540           break;
29541
29542     case T_RESUME:                /* resume execution */
29543           rp->p_flags &= ~P_STOP;
29544           if (rp->p_flags == 0) lock_ready(rp);
29545           TR_DATA = 0;
29546           break;
29547
29548     case T_STEP:                  /* set trace bit */
29549           rp->p_reg.psw |= TRACEBIT;
29550           rp->p_flags &= ~P_STOP;
29551           if (rp->p_flags == 0) lock_ready(rp);
29552           TR_DATA = 0;
29553           break;
29554
29555     default:
29556           return(EIO);
29557     }
29558     return(OK);
29559   }

29561   /*===========================================================================*
29562    *                              cause_sig                                    *
29563    *===========================================================================*/
29564   PUBLIC void cause_sig(proc_nr, sig_nr)
29565   int proc_nr;                  /* process to be signalled */
29566   int sig_nr;                   /* signal to be sent, 1 to _NSIG */
29567   {
29568   /* A task wants to send a signal to a process.   Examples of such tasks are:
29569    *   TTY wanting to cause SIGINT upon getting a DEL
29570    *   CLOCK wanting to cause SIGALRM when timer expires
29571    * FS also uses this to send a signal, via the SYS_KILL message.
29572    * Signals are handled by sending a message to MM.  The tasks don't dare do
29573    * that directly, for fear of what would happen if MM were busy.  Instead they
29574    * call cause_sig, which sets bits in p_pending, and then carefully checks to
29575    * see if MM is free.  If so, a message is sent to it.  If not, when it becomes
29576    * free, a message is sent.  The process being signaled is blocked while MM
29577    * has not seen or finished with all signals for it.  These signals are
29578    * counted in p_pendcount, and the SIG_PENDING flag is kept nonzero while
29579    * there are some.  It is not sufficient to ready the process when MM is
29580    * informed, because MM can block waiting for FS to do a core dump.
29581    */
29582
29583     register struct proc *rp, *mmp;
29584
29585     rp = proc_addr(proc_nr);
29586     if (sigismember(&rp->p_pending, sig_nr))
29587           return;                 /* this signal already pending */
29588     sigaddset(&rp->p_pending, sig_nr);
29589     ++rp->p_pendcount;            /* count new signal pending */
29590     if (rp->p_flags & PENDING)
29591           return;                 /* another signal already pending */
29592     if (rp->p_flags == 0) lock_unready(rp);
29593     rp->p_flags |= PENDING | SIG_PENDING;
29594     ++sig_procs;                  /* count new process pending */
29595
29596     mmp = proc_addr(MM_PROC_NR);
29597     if ( ((mmp->p_flags & RECEIVING) == 0) || mmp->p_getfrom != ANY) return;
29598     inform();
29599   }
```

```
29602   /*===========================================================================*
29603    *                              inform                                       *
29604    *===========================================================================*/
29605   PUBLIC void inform()
29606   {
29607   /* When a signal is detected by the kernel (e.g., DEL), or generated by a task
29608    * (e.g. clock task for SIGALRM), cause_sig() is called to set a bit in the
29609    * p_pending field of the process to signal.  Then inform() is called to see
29610    * if MM is idle and can be told about it.  Whenever MM blocks, a check is
29611    * made to see if 'sig_procs' is nonzero; if so, inform() is called.
29612    */
29613
29614     register struct proc *rp;
29615
29616     /* MM is waiting for new input.  Find a process with pending signals. */
29617     for (rp = BEG_SERV_ADDR; rp < END_PROC_ADDR; rp++)
29618           if (rp->p_flags & PENDING) {
29619                   m.m_type = KSIG;
29620                   m.SIG_PROC = proc_number(rp);
29621                   m.SIG_MAP = rp->p_pending;
29622                   sig_procs--;
29623                   if (lock_mini_send(proc_addr(HARDWARE), MM_PROC_NR, &m) != OK)
29624                           panic("can't inform MM", NO_NUM);
29625                   sigemptyset(&rp->p_pending); /* the ball is now in MM's court */
29626                   rp->p_flags &= ~PENDING;/* remains inhibited by SIG_PENDING */
29627                   lock_pick_proc();       /* avoid delay in scheduling MM */
29628                   return;
29629           }
29630   }


29633   /*===========================================================================*
29634    *                              umap                                         *
29635    *===========================================================================*/
29636   PUBLIC phys_bytes umap(rp, seg, vir_addr, bytes)
29637   register struct proc *rp;       /* pointer to proc table entry for process */
29638   int seg;                        /* T, D, or S segment */
29639   vir_bytes vir_addr;             /* virtual address in bytes within the seg */
29640   vir_bytes bytes;                /* # of bytes to be copied */
29641   {
29642   /* Calculate the physical memory address for a given virtual address. */
29643
29644     vir_clicks vc;                /* the virtual address in clicks */
29645     phys_bytes pa;                /* intermediate variables as phys_bytes */
29646   #if (CHIP == INTEL)
29647     phys_bytes seg_base;
29648   #endif
29649
29650     /* If 'seg' is D it could really be S and vice versa.  T really means T.
29651      * If the virtual address falls in the gap,  it causes a problem. On the
29652      * 8088 it is probably a legal stack reference, since "stackfaults" are
29653      * not detected by the hardware.  On 8088s, the gap is called S and
29654      * accepted, but on other machines it is called D and rejected.
29655      * The Atari ST behaves like the 8088 in this respect.
29656      */
29657
29658     if (bytes <= 0) return( (phys_bytes) 0);
29659     vc = (vir_addr + bytes - 1) >> CLICK_SHIFT;   /* last click of data */
```

```
29660
29661   #if (CHIP == INTEL) || (CHIP == M68000)
29662     if (seg != T)
29663           seg = (vc < rp->p_map[D].mem_vir + rp->p_map[D].mem_len ? D : S);
29664   #else
29665     if (seg != T)
29666           seg = (vc < rp->p_map[S].mem_vir ? D : S);
29667   #endif
29668
29669     if((vir_addr>>CLICK_SHIFT) >= rp->p_map[seg].mem_vir+ rp->p_map[seg].mem_len)
29670           return( (phys_bytes) 0 );
29671   #if (CHIP == INTEL)
29672     seg_base = (phys_bytes) rp->p_map[seg].mem_phys;
29673     seg_base = seg_base << CLICK_SHIFT;   /* segment origin in bytes */
29674   #endif
29675     pa = (phys_bytes) vir_addr;
29676   #if (CHIP != M68000)
29677     pa -= rp->p_map[seg].mem_vir << CLICK_SHIFT;
29678     return(seg_base + pa);
29679   #endif
29680   #if (CHIP == M68000)
29681   #if (SHADOWING == 0)
29682     pa -= (phys_bytes)rp->p_map[seg].mem_vir << CLICK_SHIFT;
29683     pa += (phys_bytes)rp->p_map[seg].mem_phys << CLICK_SHIFT;
29684   #else
29685     if (rp->p_shadow && seg != T) {
29686           pa -= (phys_bytes)rp->p_map[D].mem_phys << CLICK_SHIFT;
29687           pa += (phys_bytes)rp->p_shadow << CLICK_SHIFT;
29688     }
29689   #endif
29690     return(pa);
29691   #endif
29692   }


29695   /*===========================================================================*
29696    *                              numap                                        *
29697    *===========================================================================*/
29698   PUBLIC phys_bytes numap(proc_nr, vir_addr, bytes)
29699   int proc_nr;                    /* process number to be mapped */
29700   vir_bytes vir_addr;             /* virtual address in bytes within D seg */
29701   vir_bytes bytes;                /* # of bytes required in segment  */
29702   {
29703   /* Do umap() starting from a process number instead of a pointer.  This
29704    * function is used by device drivers, so they need not know about the
29705    * process table.  To save time, there is no 'seg' parameter. The segment
29706    * is always D.
29707    */
29708
29709     return(umap(proc_addr(proc_nr), D, vir_addr, bytes));
29710   }


29713   #if (CHIP == INTEL)
29714   /*===========================================================================*
29715    *                              alloc_segments                               *
29716    *===========================================================================*/
29717   PUBLIC void alloc_segments(rp)
29718   register struct proc *rp;
29719   {
```

```
29720    /* This is called only by do_newmap, but is broken out as a separate function
29721     * because so much is hardware-dependent.
29722     */
29723
29724      phys_bytes code_bytes;
29725      phys_bytes data_bytes;
29726      int privilege;
29727
29728      if (protected_mode) {
29729            data_bytes = (phys_bytes) (rp->p_map[S].mem_vir + rp->p_map[S].mem_len)
29730                        << CLICK_SHIFT;
29731            if (rp->p_map[T].mem_len == 0)
29732                    code_bytes = data_bytes;        /* common I&D, poor protect */
29733            else
29734                    code_bytes = (phys_bytes) rp->p_map[T].mem_len << CLICK_SHIFT;
29735            privilege = istaskp(rp) ? TASK_PRIVILEGE : USER_PRIVILEGE;
29736            init_codeseg(&rp->p_ldt[CS_LDT_INDEX],
29737                         (phys_bytes) rp->p_map[T].mem_phys << CLICK_SHIFT,
29738                         code_bytes, privilege);
29739            init_dataseg(&rp->p_ldt[DS_LDT_INDEX],
29740                         (phys_bytes) rp->p_map[D].mem_phys << CLICK_SHIFT,
29741                         data_bytes, privilege);
29742            rp->p_reg.cs = (CS_LDT_INDEX * DESC_SIZE) | TI | privilege;
29743    #if _WORD_SIZE == 4
29744            rp->p_reg.gs =
29745            rp->p_reg.fs =
29746    #endif
29747            rp->p_reg.ss =
29748            rp->p_reg.es =
29749            rp->p_reg.ds = (DS_LDT_INDEX*DESC_SIZE) | TI | privilege;
29750      } else {
29751            rp->p_reg.cs = click_to_hclick(rp->p_map[T].mem_phys);
29752            rp->p_reg.ss =
29753            rp->p_reg.es =
29754            rp->p_reg.ds = click_to_hclick(rp->p_map[D].mem_phys);
29755      }
29756    }
29757    #endif /* (CHIP == INTEL) */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/table.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

29800    /* The object file of "table.c" contains all the data.  In the *.h files,
29801     * declared variables appear with EXTERN in front of them, as in
29802     *
29803     *    EXTERN int x;
29804     *
29805     * Normally EXTERN is defined as extern, so when they are included in another
29806     * file, no storage is allocated.  If the EXTERN were not present, but just
29807     * say,
29808     *
29809     *    int x;
29810     *
29811     * then including this file in several source files would cause 'x' to be
29812     * declared several times.  While some linkers accept this, others do not,
29813     * so they are declared extern when included normally.  However, it must
29814     * be declared for real somewhere.  That is done here, by redefining
```

```
29815     * EXTERN as the null string, so the inclusion of all the *.h files in
29816     * table.c actually generates storage for them.  All the initialized
29817     * variables are also declared here, since
29818     *
29819     * extern int x = 4;
29820     *
29821     * is not allowed.  If such variables are shared, they must also be declared
29822     * in one of the *.h files without the initialization.
29823     */
29824
29825    #define _TABLE
29826
29827    #include "kernel.h"
29828    #include <termios.h>
29829    #include <minix/com.h>
29830    #include "proc.h"
29831    #include "tty.h"
29832
29833    /* The startup routine of each task is given below, from -NR_TASKS upwards.
29834     * The order of the names here MUST agree with the numerical values assigned to
29835     * the tasks in <minix/com.h>.
29836     */
29837    #define SMALL_STACK     (128 * sizeof(char *))
29838
29839    #define TTY_STACK       (3 * SMALL_STACK)
29840    #define SYN_ALRM_STACK  SMALL_STACK
29841
29842    #define DP8390_STACK    (SMALL_STACK * ENABLE_NETWORKING)
29843
29844    #if (CHIP == INTEL)
29845    #define IDLE_STACK      ((3+3+4) * sizeof(char *))  /* 3 intr, 3 temps, 4 db */
29846    #else
29847    #define IDLE_STACK      SMALL_STACK
29848    #endif
29849
29850    #define PRINTER_STACK   SMALL_STACK
29851
29852    #if (CHIP == INTEL)
29853    #define WINCH_STACK     (2 * SMALL_STACK * ENABLE_WINI)
29854    #else
29855    #define WINCH_STACK     (3 * SMALL_STACK * ENABLE_WINI)
29856    #endif
29857
29858    #if (MACHINE == ATARI)
29859    #define SCSI_STACK      (3 * SMALL_STACK)
29860    #endif
29861
29862    #if (MACHINE == IBM_PC)
29863    #define SCSI_STACK      (2 * SMALL_STACK * ENABLE_SCSI)
29864    #endif
29865
29866    #define CDROM_STACK     (4 * SMALL_STACK * ENABLE_CDROM)
29867    #define AUDIO_STACK     (4 * SMALL_STACK * ENABLE_AUDIO)
29868    #define MIXER_STACK     (4 * SMALL_STACK * ENABLE_AUDIO)
29869
29870    #define FLOP_STACK      (3 * SMALL_STACK)
29871    #define MEM_STACK       SMALL_STACK
29872    #define CLOCK_STACK     SMALL_STACK
29873    #define SYS_STACK       SMALL_STACK
29874    #define HARDWARE_STACK  0               /* dummy task, uses kernel stack */
```

```
29875
29876
29877   #define TOT_STACK_SPACE         (TTY_STACK + DP8390_STACK + SCSI_STACK + \
29878           SYN_ALRM_STACK + IDLE_STACK + HARDWARE_STACK + PRINTER_STACK + \
29879           WINCH_STACK + FLOP_STACK + MEM_STACK + CLOCK_STACK + SYS_STACK + \
29880           CDROM_STACK + AUDIO_STACK + MIXER_STACK)
29881
29882
29883   /* SCSI, CDROM and AUDIO may in the future have different choices like
29884    * WINCHESTER, but for now the choice is fixed.
29885    */
29886   #define scsi_task       aha_scsi_task
29887   #define cdrom_task      mcd_task
29888   #define audio_task      dsp_task
29889
29890
29891   /*
29892    * Some notes about the following table:
29893    *  1) The tty_task should always be first so that other tasks can use printf
29894    *     if their initialisation has problems.
29895    *  2) If you add a new kernel task, add it before the printer task.
29896    *  3) The task name is used for the process name (p_name).
29897    */
29898
29899   PUBLIC struct tasktab tasktab[] = {
29900           { tty_task,             TTY_STACK,      "TTY"           },
29901   #if ENABLE_NETWORKING
29902           { dp8390_task,          DP8390_STACK,   "DP8390"        },
29903   #endif
29904   #if ENABLE_CDROM
29905           { cdrom_task,           CDROM_STACK,    "CDROM"         },
29906   #endif
29907   #if ENABLE_AUDIO
29908           { audio_task,           AUDIO_STACK,    "AUDIO"         },
29909           { mixer_task,           MIXER_STACK,    "MIXER"         },
29910   #endif
29911   #if ENABLE_SCSI
29912           { scsi_task,            SCSI_STACK,     "SCSI"          },
29913   #endif
29914   #if ENABLE_WINI
29915           { winchester_task,      WINCH_STACK,    "WINCH"         },
29916   #endif
29917           { syn_alrm_task,        SYN_ALRM_STACK, "SYN_AL"        },
29918           { idle_task,            IDLE_STACK,     "IDLE"          },
29919           { printer_task,         PRINTER_STACK,  "PRINTER"       },
29920           { floppy_task,          FLOP_STACK,     "FLOPPY"        },
29921           { mem_task,             MEM_STACK,      "MEMORY"        },
29922           { clock_task,           CLOCK_STACK,    "CLOCK"         },
29923           { sys_task,             SYS_STACK,      "SYS"           },
29924           { 0,                    HARDWARE_STACK, "HARDWAR"       },
29925           { 0,                    0,              "MM"            },
29926           { 0,                    0,              "FS"            },
29927   #if ENABLE_NETWORKING
29928           { 0,                    0,              "INET"          },
29929   #endif
29930           { 0,                    0,              "INIT"          },
29931   };
29932
29933   /* Stack space for all the task stacks.  (Declared as (char *) to align it.) */
29934   PUBLIC char *t_stack[TOT_STACK_SPACE / sizeof(char *)];
```

```
29935
29936   /*
29937    * The number of kernel tasks must be the same as NR_TASKS.
29938    * If NR_TASKS is not correct then you will get the compile error:
29939    *   "array size is negative"
29940    */
29941
29942   #define NKT (sizeof tasktab / sizeof (struct tasktab) - (INIT_PROC_NR + 1))
29943
29944   extern int dummy_tasktab_check[NR_TASKS == NKT ? 1 : -1];
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/tty.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

30000   /* This file contains the terminal driver, both for the IBM console and regular
30001    * ASCII terminals.  It handles only the device-independent part of a TTY, the
30002    * device dependent parts are in console.c, rs232.c, etc.  This file contains
30003    * two main entry points, tty_task() and tty_wakeup(), and several minor entry
30004    * points for use by the device-dependent code.
30005    *
30006    * The device-independent part accepts "keyboard" input from the device-
30007    * dependent part, performs input processing (special key interpretation),
30008    * and sends the input to a process reading from the TTY.  Output to a TTY
30009    * is sent to the device-dependent code for output processing and "screen"
30010    * display.  Input processing is done by the device by calling 'in_process'
30011    * on the input characters, output processing may be done by the device itself
30012    * or by calling 'out_process'.  The TTY takes care of input queuing, the
30013    * device does the output queuing.  If a device receives an external signal,
30014    * like an interrupt, then it causes tty_wakeup() to be run by the CLOCK task
30015    * to, you guessed it, wake up the TTY to check if input or output can
30016    * continue.
30017    *
30018    * The valid messages and their parameters are:
30019    *
30020    *   HARD_INT:     output has been completed or input has arrived
30021    *   DEV_READ:     a process wants to read from a terminal
30022    *   DEV_WRITE:    a process wants to write on a terminal
30023    *   DEV_IOCTL:    a process wants to change a terminal's parameters
30024    *   DEV_OPEN:     a tty line has been opened
30025    *   DEV_CLOSE:    a tty line has been closed
30026    *   CANCEL:       terminate a previous incomplete system call immediately
30027    *
30028    *    m_type      TTY_LINE   PROC_NR    COUNT   TTY_SPEK  TTY_FLAGS  ADDRESS
30029    * ---------------------------------------------------------------------------
30030    * | HARD_INT    |         |         |         |         |         |         |
30031    * |-------------+---------+---------+---------+---------+---------+---------|
30032    * | DEV_READ    |minor dev| proc nr |  count  |         |O_NONBLOCK| buf ptr |
30033    * |-------------+---------+---------+---------+---------+---------+---------|
30034    * | DEV_WRITE   |minor dev| proc nr |  count  |         |         | buf ptr |
30035    * |-------------+---------+---------+---------+---------+---------+---------|
30036    * | DEV_IOCTL   |minor dev| proc nr |func code|erase etc|  flags  |         |
30037    * |-------------+---------+---------+---------+---------+---------+---------|
30038    * | DEV_OPEN    |minor dev| proc nr | O_NOCTTY|         |         |         |
30039    * |-------------+---------+---------+---------+---------+---------+---------|
```

```
30040     * | DEV_CLOSE   |minor dev| proc nr |         |         |         |         |
30041     * |-------------+---------+---------+---------+---------+---------+---------|
30042     * | CANCEL      |minor dev| proc nr |         |         |         |         |
30043     * ---------------------------------------------------------------------------
30044     */
30045
30046    #include "kernel.h"
30047    #include <termios.h>
30048    #if ENABLE_SRCCOMPAT || ENABLE_BINCOMPAT
30049    #include <sgtty.h>
30050    #endif
30051    #include <sys/ioctl.h>
30052    #include <signal.h>
30053    #include <minix/callnr.h>
30054    #include <minix/com.h>
30055    #if (CHIP == INTEL)
30056    #include <minix/keymap.h>
30057    #endif
30058    #include "tty.h"
30059    #include "proc.h"
30060
30061    /* Address of a tty structure. */
30062    #define tty_addr(line)  (&tty_table[line])
30063
30064    /* First minor numbers for the various classes of TTY devices. */
30065    #define CONS_MINOR        0
30066    #define LOG_MINOR        15
30067    #define RS232_MINOR      16
30068    #define TTYPX_MINOR     128
30069    #define PTYPX_MINOR     192
30070
30071    /* Macros for magic tty types. */
30072    #define isconsole(tp)   ((tp) < tty_addr(NR_CONS))
30073
30074    /* Macros for magic tty structure pointers. */
30075    #define FIRST_TTY       tty_addr(0)
30076    #define END_TTY         tty_addr(sizeof(tty_table) / sizeof(tty_table[0]))
30077
30078    /* A device exists if at least its 'devread' function is defined. */
30079    #define tty_active(tp)  ((tp)->tty_devread != NULL)
30080
30081    /* RS232 lines or pseudo terminals can be completely configured out. */
30082    #if NR_RS_LINES == 0
30083    #define rs_init(tp)     ((void) 0)
30084    #endif
30085    #if NR_PTYS == 0
30086    #define pty_init(tp)    ((void) 0)
30087    #define do_pty(tp, mp)  ((void) 0)
30088    #endif
30089
30090    FORWARD _PROTOTYPE( void do_cancel, (tty_t *tp, message *m_ptr)         );
30091    FORWARD _PROTOTYPE( void do_ioctl, (tty_t *tp, message *m_ptr)          );
30092    FORWARD _PROTOTYPE( void do_open, (tty_t *tp, message *m_ptr)           );
30093    FORWARD _PROTOTYPE( void do_close, (tty_t *tp, message *m_ptr)          );
30094    FORWARD _PROTOTYPE( void do_read, (tty_t *tp, message *m_ptr)           );
30095    FORWARD _PROTOTYPE( void do_write, (tty_t *tp, message *m_ptr)          );
30096    FORWARD _PROTOTYPE( void in_transfer, (tty_t *tp)                       );
30097    FORWARD _PROTOTYPE( int echo, (tty_t *tp, int ch)                       );
30098    FORWARD _PROTOTYPE( void rawecho, (tty_t *tp, int ch)                   );
30099    FORWARD _PROTOTYPE( int back_over, (tty_t *tp)                          );
```

```
30100   FORWARD _PROTOTYPE( void reprint, (tty_t *tp)                          );
30101   FORWARD _PROTOTYPE( void dev_ioctl, (tty_t *tp)                        );
30102   FORWARD _PROTOTYPE( void setattr, (tty_t *tp)                          );
30103   FORWARD _PROTOTYPE( void tty_icancel, (tty_t *tp)                      );
30104   FORWARD _PROTOTYPE( void tty_init, (tty_t *tp)                         );
30105   FORWARD _PROTOTYPE( void settimer, (tty_t *tp, int on)                 );
30106   #if ENABLE_SRCCOMPAT || ENABLE_BINCOMPAT
30107   FORWARD _PROTOTYPE( int compat_getp, (tty_t *tp, struct sgttyb *sg)    );
30108   FORWARD _PROTOTYPE( int compat_getc, (tty_t *tp, struct tchars *sg)    );
30109   FORWARD _PROTOTYPE( int compat_setp, (tty_t *tp, struct sgttyb *sg)    );
30110   FORWARD _PROTOTYPE( int compat_setc, (tty_t *tp, struct tchars *sg)    );
30111   FORWARD _PROTOTYPE( int tspd2sgspd, (speed_t tspd)                     );
30112   FORWARD _PROTOTYPE( speed_t sgspd2tspd, (int sgspd)                    );
30113   #if ENABLE_BINCOMPAT
30114   FORWARD _PROTOTYPE( void do_ioctl_compat, (tty_t *tp, message *m_ptr)  );
30115   #endif
30116   #endif
30117
30118   /* Default attributes. */
30119   PRIVATE struct termios termios_defaults = {
30120     TINPUT_DEF, TOUTPUT_DEF, TCTRL_DEF, TLOCAL_DEF, TSPEED_DEF, TSPEED_DEF,
30121     {
30122           TEOF_DEF, TEOL_DEF, TERASE_DEF, TINTR_DEF, TKILL_DEF, TMIN_DEF,
30123           TQUIT_DEF, TTIME_DEF, TSUSP_DEF, TSTART_DEF, TSTOP_DEF,
30124           TREPRINT_DEF, TLNEXT_DEF, TDISCARD_DEF,
30125     },
30126   };
30127   PRIVATE struct winsize winsize_defaults;        /* = all zeroes */
30128
30129
30130   /*===========================================================================*
30131    *                               tty_task                                    *
30132    *===========================================================================*/
30133   PUBLIC void tty_task()
30134   {
30135   /* Main routine of the terminal task. */
30136
30137     message tty_mess;             /* buffer for all incoming messages */
30138     register tty_t *tp;
30139     unsigned line;
30140
30141     /* Initialize the terminal lines. */
30142     for (tp = FIRST_TTY; tp < END_TTY; tp++) tty_init(tp);
30143
30144     /* Display the Minix startup banner. */
30145     printf("Minix %s.%s  Copyright 1997 Prentice-Hall, Inc.\n\n",
30146                                             OS_RELEASE, OS_VERSION);
30147
30148   #if (CHIP == INTEL)
30149     /* Real mode, or 16/32-bit protected mode? */
30150   #if _WORD_SIZE == 4
30151     printf("Executing in 32-bit protected mode\n\n");
30152   #else
30153     printf("Executing in %s mode\n\n",
30154           protected_mode ? "16-bit protected" : "real");
30155   #endif
30156   #endif
30157
30158     while (TRUE) {
30159           /* Handle any events on any of the ttys. */
```

```
30160           for (tp = FIRST_TTY; tp < END_TTY; tp++) {
30161                   if (tp->tty_events) handle_events(tp);
30162           }
30163
30164           receive(ANY, &tty_mess);
30165
30166           /* A hardware interrupt is an invitation to check for events. */
30167           if (tty_mess.m_type == HARD_INT) continue;
30168
30169           /* Check the minor device number. */
30170           line = tty_mess.TTY_LINE;
30171           if ((line - CONS_MINOR) < NR_CONS) {
30172                   tp = tty_addr(line - CONS_MINOR);
30173           } else
30174           if (line == LOG_MINOR) {
30175                   tp = tty_addr(0);
30176           } else
30177           if ((line - RS232_MINOR) < NR_RS_LINES) {
30178                   tp = tty_addr(line - RS232_MINOR + NR_CONS);
30179           } else
30180           if ((line - TTYPX_MINOR) < NR_PTYS) {
30181                   tp = tty_addr(line - TTYPX_MINOR + NR_CONS + NR_RS_LINES);
30182           } else
30183           if ((line - PTYPX_MINOR) < NR_PTYS) {
30184                   tp = tty_addr(line - PTYPX_MINOR + NR_CONS + NR_RS_LINES);
30185                   do_pty(tp, &tty_mess);
30186                   continue;                       /* this is a pty, not a tty */
30187           } else {
30188                   tp = NULL;
30189           }
30190
30191           /* If the device doesn't exist or is not configured return ENXIO. */
30192           if (tp == NULL || !tty_active(tp)) {
30193                   tty_reply(TASK_REPLY, tty_mess.m_source,
30194                                                   tty_mess.PROC_NR, ENXIO);
30195                   continue;
30196           }
30197
30198           /* Execute the requested function. */
30199           switch (tty_mess.m_type) {
30200               case DEV_READ:      do_read(tp, &tty_mess);         break;
30201               case DEV_WRITE:     do_write(tp, &tty_mess);        break;
30202               case DEV_IOCTL:     do_ioctl(tp, &tty_mess);        break;
30203               case DEV_OPEN:      do_open(tp, &tty_mess);         break;
30204               case DEV_CLOSE:     do_close(tp, &tty_mess);        break;
30205               case CANCEL:        do_cancel(tp, &tty_mess);       break;
30206               default:            tty_reply(TASK_REPLY, tty_mess.m_source,
30207                                                   tty_mess.PROC_NR, EINVAL);
30208           }
30209       }
30210   }


30213   /*===========================================================================*
30214    *                              do_read                                      *
30215    *===========================================================================*/
30216   PRIVATE void do_read(tp, m_ptr)
30217   register tty_t *tp;             /* pointer to tty struct */
30218   message *m_ptr;                 /* pointer to message sent to the task */
30219   {
```

```
30220   /* A process wants to read from a terminal. */
30221     int r;
30222
30223     /* Check if there is already a process hanging in a read, check if the
30224      * parameters are correct, do I/O.
30225      */
30226     if (tp->tty_inleft > 0) {
30227           r = EIO;
30228     } else
30229     if (m_ptr->COUNT <= 0) {
30230           r = EINVAL;
30231     } else
30232     if (numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, m_ptr->COUNT) == 0) {
30233           r = EFAULT;
30234     } else {
30235           /* Copy information from the message to the tty struct. */
30236           tp->tty_inrepcode = TASK_REPLY;
30237           tp->tty_incaller = m_ptr->m_source;
30238           tp->tty_inproc = m_ptr->PROC_NR;
30239           tp->tty_in_vir = (vir_bytes) m_ptr->ADDRESS;
30240           tp->tty_inleft = m_ptr->COUNT;
30241
30242           if (!(tp->tty_termios.c_lflag & ICANON)
30243                                       && tp->tty_termios.c_cc[VTIME] > 0) {
30244                   if (tp->tty_termios.c_cc[VMIN] == 0) {
30245                           /* MIN & TIME specify a read timer that finishes the
30246                            * read in TIME/10 seconds if no bytes are available.
30247                            */
30248                           lock();
30249                           settimer(tp, TRUE);
30250                           tp->tty_min = 1;
30251                           unlock();
30252                   } else {
30253                           /* MIN & TIME specify an inter-byte timer that may
30254                            * have to be cancelled if there are no bytes yet.
30255                            */
30256                           if (tp->tty_eotct == 0) {
30257                                   lock();
30258                                   settimer(tp, FALSE);
30259                                   unlock();
30260                                   tp->tty_min = tp->tty_termios.c_cc[VMIN];
30261                           }
30262                   }
30263           }
30264
30265           /* Anything waiting in the input buffer? Clear it out... */
30266           in_transfer(tp);
30267           /* ...then go back for more */
30268           handle_events(tp);
30269           if (tp->tty_inleft == 0) return;                /* already done */
30270
30271           /* There were no bytes in the input queue available, so either suspend
30272            * the caller or break off the read if nonblocking.
30273            */
30274           if (m_ptr->TTY_FLAGS & O_NONBLOCK) {
30275                   r = EAGAIN;                             /* cancel the read */
30276                   tp->tty_inleft = tp->tty_incum = 0;
30277           } else {
30278                   r = SUSPEND;                            /* suspend the caller */
30279                   tp->tty_inrepcode = REVIVE;
```

```
30280           }
30281     }
30282     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
30283   }


30286   /*===========================================================================*
30287    *                              do_write                                     *
30288    *===========================================================================*/
30289   PRIVATE void do_write(tp, m_ptr)
30290   register tty_t *tp;
30291   register message *m_ptr;        /* pointer to message sent to the task */
30292   {
30293   /* A process wants to write on a terminal. */
30294     int r;
30295
30296     /* Check if there is already a process hanging in a write, check if the
30297      * parameters are correct, do I/O.
30298      */
30299     if (tp->tty_outleft > 0) {
30300           r = EIO;
30301     } else
30302     if (m_ptr->COUNT <= 0) {
30303           r = EINVAL;
30304     } else
30305     if (numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, m_ptr->COUNT) == 0) {
30306           r = EFAULT;
30307     } else {
30308           /* Copy message parameters to the tty structure. */
30309           tp->tty_outrepcode = TASK_REPLY;
30310           tp->tty_outcaller = m_ptr->m_source;
30311           tp->tty_outproc = m_ptr->PROC_NR;
30312           tp->tty_out_vir = (vir_bytes) m_ptr->ADDRESS;
30313           tp->tty_outleft = m_ptr->COUNT;
30314
30315           /* Try to write. */
30316           handle_events(tp);
30317           if (tp->tty_outleft == 0) return;               /* already done */
30318
30319           /* None or not all the bytes could be written, so either suspend the
30320            * caller or break off the write if nonblocking.
30321            */
30322           if (m_ptr->TTY_FLAGS & O_NONBLOCK) {            /* cancel the write */
30323                   r = tp->tty_outcum > 0 ? tp->tty_outcum : EAGAIN;
30324                   tp->tty_outleft = tp->tty_outcum = 0;
30325           } else {
30326                   r = SUSPEND;                            /* suspend the caller */
30327                   tp->tty_outrepcode = REVIVE;
30328           }
30329     }
30330     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
30331   }


30334   /*===========================================================================*
30335    *                              do_ioctl                                     *
30336    *===========================================================================*/
30337   PRIVATE void do_ioctl(tp, m_ptr)
30338   register tty_t *tp;
30339   message *m_ptr;                 /* pointer to message sent to task */
```

```
30340   {
30341   /* Perform an IOCTL on this terminal. Posix termios calls are handled
30342    * by the IOCTL system call
30343    */
30344
30345     int r;
30346     union {
30347           int i;
30348   #if ENABLE_SRCCOMPAT
30349           struct sgttyb sg;
30350           struct tchars tc;
30351   #endif
30352     } param;
30353     phys_bytes user_phys;
30354     size_t size;
30355
30356     /* Size of the ioctl parameter. */
30357     switch (m_ptr->TTY_REQUEST) {
30358       case TCGETS:        /* Posix tcgetattr function */
30359       case TCSETS:        /* Posix tcsetattr function, TCSANOW option */
30360       case TCSETSW:       /* Posix tcsetattr function, TCSADRAIN option */
30361       case TCSETSF:       /* Posix tcsetattr function, TCSAFLUSH option */
30362           size = sizeof(struct termios);
30363           break;
30364
30365       case TCSBRK:        /* Posix tcsendbreak function */
30366       case TCFLOW:        /* Posix tcflow function */
30367       case TCFLSH:        /* Posix tcflush function */
30368       case TIOCGPGRP:     /* Posix tcgetpgrp function */
30369       case TIOCSPGRP:     /* Posix tcsetpgrp function */
30370           size = sizeof(int);
30371           break;
30372
30373       case TIOCGWINSZ:    /* get window size (not Posix) */
30374       case TIOCSWINSZ:    /* set window size (not Posix) */
30375           size = sizeof(struct winsize);
30376           break;
30377
30378   #if ENABLE_SRCCOMPAT
30379       case TIOCGETP:      /* BSD-style get terminal properties */
30380       case TIOCSETP:      /* BSD-style set terminal properties */
30381           size = sizeof(struct sgttyb);
30382           break;
30383
30384       case TIOCGETC:      /* BSD-style get terminal special characters */
30385       case TIOCSETC:      /* BSD-style get terminal special characters */
30386           size = sizeof(struct tchars);
30387           break;
30388   #endif
30389   #if (MACHINE == IBM_PC)
30390       case KIOCSMAP:      /* load keymap (Minix extension) */
30391           size = sizeof(keymap_t);
30392           break;
30393
30394       case TIOCSFON:      /* load font (Minix extension) */
30395           size = sizeof(u8_t [8192]);
30396           break;
30397
30398   #endif
30399       case TCDRAIN:       /* Posix tcdrain function -- no parameter */
```

```
30400         default:            size = 0;
30401       }
30402
30403       if (size != 0) {
30404             user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, size);
30405             if (user_phys == 0) {
30406                     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, EFAULT);
30407                     return;
30408             }
30409       }
30410
30411       r = OK;
30412       switch (m_ptr->TTY_REQUEST) {
30413         case TCGETS:
30414             /* Get the termios attributes. */
30415             phys_copy(vir2phys(&tp->tty_termios), user_phys, (phys_bytes) size);
30416             break;
30417
30418         case TCSETSW:
30419         case TCSETSF:
30420         case TCDRAIN:
30421             if (tp->tty_outleft > 0) {
30422                     /* Wait for all ongoing output processing to finish. */
30423                     tp->tty_iocaller = m_ptr->m_source;
30424                     tp->tty_ioproc = m_ptr->PROC_NR;
30425                     tp->tty_ioreq = m_ptr->REQUEST;
30426                     tp->tty_iovir = (vir_bytes) m_ptr->ADDRESS;
30427                     r = SUSPEND;
30428                     break;
30429             }
30430             if (m_ptr->TTY_REQUEST == TCDRAIN) break;
30431             if (m_ptr->TTY_REQUEST == TCSETSF) tty_icancel(tp);
30432             /*FALL THROUGH*/
30433         case TCSETS:
30434             /* Set the termios attributes. */
30435             phys_copy(user_phys, vir2phys(&tp->tty_termios), (phys_bytes) size);
30436             setattr(tp);
30437             break;
30438
30439         case TCFLSH:
30440             phys_copy(user_phys, vir2phys(&param.i), (phys_bytes) size);
30441             switch (param.i) {
30442                 case TCIFLUSH:      tty_icancel(tp);                          break;
30443                 case TCOFLUSH:      (*tp->tty_ocancel)(tp);                   break;
30444                 case TCIOFLUSH:     tty_icancel(tp); (*tp->tty_ocancel)(tp);break;
30445                 default:            r = EINVAL;
30446             }
30447             break;
30448
30449         case TCFLOW:
30450             phys_copy(user_phys, vir2phys(&param.i), (phys_bytes) size);
30451             switch (param.i) {
30452                 case TCOOFF:
30453                 case TCOON:
30454                     tp->tty_inhibited = (param.i == TCOOFF);
30455                     tp->tty_events = 1;
30456                     break;
30457                 case TCIOFF:
30458                     (*tp->tty_echo)(tp, tp->tty_termios.c_cc[VSTOP]);
30459                     break;
```

```
30460               case TCION:
30461                   (*tp->tty_echo)(tp, tp->tty_termios.c_cc[VSTART]);
30462                   break;
30463               default:
30464                   r = EINVAL;
30465           }
30466           break;
30467
30468       case TCSBRK:
30469           if (tp->tty_break != NULL) (*tp->tty_break)(tp);
30470           break;
30471
30472       case TIOCGWINSZ:
30473           phys_copy(vir2phys(&tp->tty_winsize), user_phys, (phys_bytes) size);
30474           break;
30475
30476       case TIOCSWINSZ:
30477           phys_copy(user_phys, vir2phys(&tp->tty_winsize), (phys_bytes) size);
30478           /* SIGWINCH... */
30479           break;
30480
30481   #if ENABLE_SRCCOMPAT
30482       case TIOCGETP:
30483           compat_getp(tp, &param.sg);
30484           phys_copy(vir2phys(&param.sg), user_phys, (phys_bytes) size);
30485           break;
30486
30487       case TIOCSETP:
30488           phys_copy(user_phys, vir2phys(&param.sg), (phys_bytes) size);
30489           compat_setp(tp, &param.sg);
30490           break;
30491
30492       case TIOCGETC:
30493           compat_getc(tp, &param.tc);
30494           phys_copy(vir2phys(&param.tc), user_phys, (phys_bytes) size);
30495           break;
30496
30497       case TIOCSETC:
30498           phys_copy(user_phys, vir2phys(&param.tc), (phys_bytes) size);
30499           compat_setc(tp, &param.tc);
30500           break;
30501   #endif
30502
30503   #if (MACHINE == IBM_PC)
30504       case KIOCSMAP:
30505           /* Load a new keymap (only /dev/console). */
30506           if (isconsole(tp)) r = kbd_loadmap(user_phys);
30507           break;
30508
30509       case TIOCSFON:
30510           /* Load a font into an EGA or VGA card (hs@hck.hr) */
30511           if (isconsole(tp)) r = con_loadfont(user_phys);
30512           break;
30513   #endif
30514
30515   #if (MACHINE == ATARI)
30516       case VDU_LOADFONT:
30517           r = vdu_loadfont(m_ptr);
30518           break;
30519   #endif
```

```
30520
30521   /* These Posix functions are allowed to fail if _POSIX_JOB_CONTROL is
30522    * not defined.
30523    */
30524       case TIOCGPGRP:
30525       case TIOCSPGRP:
30526       default:
30527   #if ENABLE_BINCOMPAT
30528           do_ioctl_compat(tp, m_ptr);
30529           return;
30530   #else
30531           r = ENOTTY;
30532   #endif
30533     }
30534
30535     /* Send the reply. */
30536     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
30537   }


30540   /*===========================================================================*
30541    *                              do_open                                      *
30542    *===========================================================================*/
30543   PRIVATE void do_open(tp, m_ptr)
30544   register tty_t *tp;
30545   message *m_ptr;                 /* pointer to message sent to task */
30546   {
30547   /* A tty line has been opened.  Make it the callers controlling tty if
30548    * O_NOCTTY is *not* set and it is not the log device.  1 is returned if
30549    * the tty is made the controlling tty, otherwise OK or an error code.
30550    */
30551     int r = OK;
30552
30553     if (m_ptr->TTY_LINE == LOG_MINOR) {
30554           /* The log device is a write-only diagnostics device. */
30555           if (m_ptr->COUNT & R_BIT) r = EACCES;
30556     } else {
30557           if (!(m_ptr->COUNT & O_NOCTTY)) {
30558                   tp->tty_pgrp = m_ptr->PROC_NR;
30559                   r = 1;
30560           }
30561           tp->tty_openct++;
30562     }
30563     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
30564   }


30567   /*===========================================================================*
30568    *                              do_close                                     *
30569    *===========================================================================*/
30570   PRIVATE void do_close(tp, m_ptr)
30571   register tty_t *tp;
30572   message *m_ptr;                 /* pointer to message sent to task */
30573   {
30574   /* A tty line has been closed.  Clean up the line if it is the last close. */
30575
30576     if (m_ptr->TTY_LINE != LOG_MINOR && --tp->tty_openct == 0) {
30577           tp->tty_pgrp = 0;
30578           tty_icancel(tp);
30579           (*tp->tty_ocancel)(tp);
```

```
30580           (*tp->tty_close)(tp);
30581           tp->tty_termios = termios_defaults;
30582           tp->tty_winsize = winsize_defaults;
30583           setattr(tp);
30584     }
30585     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, OK);
30586   }


30589   /*===========================================================================*
30590    *                              do_cancel                                    *
30591    *===========================================================================*/
30592   PRIVATE void do_cancel(tp, m_ptr)
30593   register tty_t *tp;
30594   message *m_ptr;               /* pointer to message sent to task */
30595   {
30596   /* A signal has been sent to a process that is hanging trying to read or write.
30597    * The pending read or write must be finished off immediately.
30598    */
30599
30600     int proc_nr;
30601     int mode;
30602
30603     /* Check the parameters carefully, to avoid cancelling twice. */
30604     proc_nr = m_ptr->PROC_NR;
30605     mode = m_ptr->COUNT;
30606     if ((mode & R_BIT) && tp->tty_inleft != 0 && proc_nr == tp->tty_inproc) {
30607           /* Process was reading when killed.  Clean up input. */
30608           tty_icancel(tp);
30609           tp->tty_inleft = tp->tty_incum = 0;
30610     }
30611     if ((mode & W_BIT) && tp->tty_outleft != 0 && proc_nr == tp->tty_outproc) {
30612           /* Process was writing when killed.  Clean up output. */
30613           (*tp->tty_ocancel)(tp);
30614           tp->tty_outleft = tp->tty_outcum = 0;
30615     }
30616     if (tp->tty_ioreq != 0 && proc_nr == tp->tty_ioproc) {
30617           /* Process was waiting for output to drain. */
30618           tp->tty_ioreq = 0;
30619     }
30620     tp->tty_events = 1;
30621     tty_reply(TASK_REPLY, m_ptr->m_source, proc_nr, EINTR);
30622   }


30625   /*===========================================================================*
30626    *                              handle_events                                *
30627    *===========================================================================*/
30628   PUBLIC void handle_events(tp)
30629   tty_t *tp;                    /* TTY to check for events. */
30630   {
30631   /* Handle any events pending on a TTY.  These events are usually device
30632    * interrupts.
30633    *
30634    * Two kinds of events are prominent:
30635    *      - a character has been received from the console or an RS232 line.
30636    *      - an RS232 line has completed a write request (on behalf of a user).
30637    * The interrupt handler may delay the interrupt message at its discretion
30638    * to avoid swamping the TTY task.  Messages may be overwritten when the
30639    * lines are fast or when there are races between different lines, input
```

```
30640     * and output, because MINIX only provides single buffering for interrupt
30641     * messages (in proc.c).  This is handled by explicitly checking each line
30642     * for fresh input and completed output on each interrupt.
30643     */
30644      char *buf;
30645      unsigned count;
30646
30647      do {
30648            tp->tty_events = 0;
30649
30650            /* Read input and perform input processing. */
30651            (*tp->tty_devread)(tp);
30652
30653            /* Perform output processing and write output. */
30654            (*tp->tty_devwrite)(tp);
30655
30656            /* Ioctl waiting for some event? */
30657            if (tp->tty_ioreq != 0) dev_ioctl(tp);
30658      } while (tp->tty_events);
30659
30660      /* Transfer characters from the input queue to a waiting process. */
30661      in_transfer(tp);
30662
30663      /* Reply if enough bytes are available. */
30664      if (tp->tty_incum >= tp->tty_min && tp->tty_inleft > 0) {
30665            tty_reply(tp->tty_inrepcode, tp->tty_incaller, tp->tty_inproc,
30666                                                                  tp->tty_incum);
30667            tp->tty_inleft = tp->tty_incum = 0;
30668      }
30669    }


30672    /*===========================================================================*
30673     *                              in_transfer                                  *
30674     *===========================================================================*/
30675    PRIVATE void in_transfer(tp)
30676    register tty_t *tp;             /* pointer to terminal to read from */
30677    {
30678    /* Transfer bytes from the input queue to a process reading from a terminal. */
30679
30680      int ch;
30681      int count;
30682      phys_bytes buf_phys, user_base;
30683      char buf[64], *bp;
30684
30685      /* Anything to do? */
30686      if (tp->tty_inleft == 0 || tp->tty_eotct < tp->tty_min) return;
30687
30688      buf_phys = vir2phys(buf);
30689      user_base = proc_vir2phys(proc_addr(tp->tty_inproc), 0);
30690      bp = buf;
30691      while (tp->tty_inleft > 0 && tp->tty_eotct > 0) {
30692            ch = *tp->tty_intail;
30693
30694            if (!(ch & IN_EOF)) {
30695                    /* One character to be delivered to the user. */
30696                    *bp = ch & IN_CHAR;
30697                    tp->tty_inleft--;
30698                    if (++bp == bufend(buf)) {
30699                            /* Temp buffer full, copy to user space. */
```

```
30700                           phys_copy(buf_phys, user_base + tp->tty_in_vir,
30701                                                   (phys_bytes) buflen(buf));
30702                           tp->tty_in_vir += buflen(buf);
30703                           tp->tty_incum += buflen(buf);
30704                           bp = buf;
30705                   }
30706           }
30707
30708           /* Remove the character from the input queue. */
30709           if (++tp->tty_intail == bufend(tp->tty_inbuf))
30710                   tp->tty_intail = tp->tty_inbuf;
30711           tp->tty_incount--;
30712           if (ch & IN_EOT) {
30713                   tp->tty_eotct--;
30714                   /* Don't read past a line break in canonical mode. */
30715                   if (tp->tty_termios.c_lflag & ICANON) tp->tty_inleft = 0;
30716           }
30717     }
30718
30719     if (bp > buf) {
30720           /* Leftover characters in the buffer. */
30721           count = bp - buf;
30722           phys_copy(buf_phys, user_base + tp->tty_in_vir, (phys_bytes) count);
30723           tp->tty_in_vir += count;
30724           tp->tty_incum += count;
30725     }
30726
30727     /* Usually reply to the reader, possibly even if incum == 0 (EOF). */
30728     if (tp->tty_inleft == 0) {
30729           tty_reply(tp->tty_inrepcode, tp->tty_incaller, tp->tty_inproc,
30730                                                                   tp->tty_incum);
30731           tp->tty_inleft = tp->tty_incum = 0;
30732     }
30733   }


30736   /*===========================================================================*
30737    *                                in_process                                 *
30738    *===========================================================================*/
30739   PUBLIC int in_process(tp, buf, count)
30740   register tty_t *tp;             /* terminal on which character has arrived */
30741   char *buf;                      /* buffer with input characters */
30742   int count;                      /* number of input characters */
30743   {
30744   /* Characters have just been typed in.  Process, save, and echo them.  Return
30745    * the number of characters processed.
30746    */
30747
30748     int ch, sig, ct;
30749     int timeset = FALSE;
30750     static unsigned char csize_mask[] = { 0x1F, 0x3F, 0x7F, 0xFF };
30751
30752     for (ct = 0; ct < count; ct++) {
30753           /* Take one character. */
30754           ch = *buf++ & BYTE;
30755
30756           /* Strip to seven bits? */
30757           if (tp->tty_termios.c_iflag & ISTRIP) ch &= 0x7F;
30758
30759           /* Input extensions? */
```

```
30760           if (tp->tty_termios.c_lflag & IEXTEN) {
30761
30762                   /* Previous character was a character escape? */
30763                   if (tp->tty_escaped) {
30764                           tp->tty_escaped = NOT_ESCAPED;
30765                           ch |= IN_ESC;   /* protect character */
30766                   }
30767
30768                   /* LNEXT (^V) to escape the next character? */
30769                   if (ch == tp->tty_termios.c_cc[VLNEXT]) {
30770                           tp->tty_escaped = ESCAPED;
30771                           rawecho(tp, '^');
30772                           rawecho(tp, '\b');
30773                           continue;       /* do not store the escape */
30774                   }
30775
30776                   /* REPRINT (^R) to reprint echoed characters? */
30777                   if (ch == tp->tty_termios.c_cc[VREPRINT]) {
30778                           reprint(tp);
30779                           continue;
30780                   }
30781           }
30782
30783           /* _POSIX_VDISABLE is a normal character value, so better escape it. */
30784           if (ch == _POSIX_VDISABLE) ch |= IN_ESC;
30785
30786           /* Map CR to LF, ignore CR, or map LF to CR. */
30787           if (ch == '\r') {
30788                   if (tp->tty_termios.c_iflag & IGNCR) continue;
30789                   if (tp->tty_termios.c_iflag & ICRNL) ch = '\n';
30790           } else
30791           if (ch == '\n') {
30792                   if (tp->tty_termios.c_iflag & INLCR) ch = '\r';
30793           }
30794
30795           /* Canonical mode? */
30796           if (tp->tty_termios.c_lflag & ICANON) {
30797
30798                   /* Erase processing (rub out of last character). */
30799                   if (ch == tp->tty_termios.c_cc[VERASE]) {
30800                           (void) back_over(tp);
30801                           if (!(tp->tty_termios.c_lflag & ECHOE)) {
30802                                   (void) echo(tp, ch);
30803                           }
30804                           continue;
30805                   }
30806
30807                   /* Kill processing (remove current line). */
30808                   if (ch == tp->tty_termios.c_cc[VKILL]) {
30809                           while (back_over(tp)) {}
30810                           if (!(tp->tty_termios.c_lflag & ECHOE)) {
30811                                   (void) echo(tp, ch);
30812                                   if (tp->tty_termios.c_lflag & ECHOK)
30813                                           rawecho(tp, '\n');
30814                           }
30815                           continue;
30816                   }
30817
30818                   /* EOF (^D) means end-of-file, an invisible "line break". */
30819                   if (ch == tp->tty_termios.c_cc[VEOF]) ch |= IN_EOT | IN_EOF;
```

```
30820
30821                   /* The line may be returned to the user after an LF. */
30822                   if (ch == '\n') ch |= IN_EOT;
30823
30824                   /* Same thing with EOL, whatever it may be. */
30825                   if (ch == tp->tty_termios.c_cc[VEOL]) ch |= IN_EOT;
30826           }
30827
30828           /* Start/stop input control? */
30829           if (tp->tty_termios.c_iflag & IXON) {
30830
30831                   /* Output stops on STOP (^S). */
30832                   if (ch == tp->tty_termios.c_cc[VSTOP]) {
30833                           tp->tty_inhibited = STOPPED;
30834                           tp->tty_events = 1;
30835                           continue;
30836                   }
30837
30838                   /* Output restarts on START (^Q) or any character if IXANY. */
30839                   if (tp->tty_inhibited) {
30840                           if (ch == tp->tty_termios.c_cc[VSTART]
30841                                           || (tp->tty_termios.c_iflag & IXANY)) {
30842                                   tp->tty_inhibited = RUNNING;
30843                                   tp->tty_events = 1;
30844                                   if (ch == tp->tty_termios.c_cc[VSTART])
30845                                           continue;
30846                           }
30847                   }
30848           }
30849
30850           if (tp->tty_termios.c_lflag & ISIG) {
30851                   /* Check for INTR (^?) and QUIT (^\) characters. */
30852                   if (ch == tp->tty_termios.c_cc[VINTR]
30853                                           || ch == tp->tty_termios.c_cc[VQUIT]) {
30854                           sig = SIGINT;
30855                           if (ch == tp->tty_termios.c_cc[VQUIT]) sig = SIGQUIT;
30856                           sigchar(tp, sig);
30857                           (void) echo(tp, ch);
30858                           continue;
30859                   }
30860           }
30861
30862           /* Is there space in the input buffer? */
30863           if (tp->tty_incount == buflen(tp->tty_inbuf)) {
30864                   /* No space; discard in canonical mode, keep in raw mode. */
30865                   if (tp->tty_termios.c_lflag & ICANON) continue;
30866                   break;
30867           }
30868
30869           if (!(tp->tty_termios.c_lflag & ICANON)) {
30870                   /* In raw mode all characters are "line breaks". */
30871                   ch |= IN_EOT;
30872
30873                   /* Start an inter-byte timer? */
30874                   if (!timeset && tp->tty_termios.c_cc[VMIN] > 0
30875                                   && tp->tty_termios.c_cc[VTIME] > 0) {
30876                           lock();
30877                           settimer(tp, TRUE);
30878                           unlock();
30879                           timeset = TRUE;
```

```
30880                   }
30881           }
30882
30883           /* Perform the intricate function of echoing. */
30884           if (tp->tty_termios.c_lflag & (ECHO|ECHONL)) ch = echo(tp, ch);
30885
30886           /* Save the character in the input queue. */
30887           *tp->tty_inhead++ = ch;
30888           if (tp->tty_inhead == bufend(tp->tty_inbuf))
30889                   tp->tty_inhead = tp->tty_inbuf;
30890           tp->tty_incount++;
30891           if (ch & IN_EOT) tp->tty_eotct++;
30892
30893           /* Try to finish input if the queue threatens to overflow. */
30894           if (tp->tty_incount == buflen(tp->tty_inbuf)) in_transfer(tp);
30895     }
30896     return ct;
30897   }


30900   /*===========================================================================*
30901    *                                echo                                       *
30902    *===========================================================================*/
30903   PRIVATE int echo(tp, ch)
30904   register tty_t *tp;             /* terminal on which to echo */
30905   register int ch;                /* pointer to character to echo */
30906   {
30907   /* Echo the character if echoing is on.  Some control characters are echoed
30908    * with their normal effect, other control characters are echoed as "^X",
30909    * normal characters are echoed normally.  EOF (^D) is echoed, but immediately
30910    * backspaced over.  Return the character with the echoed length added to its
30911    * attributes.
30912    */
30913     int len, rp;
30914
30915     ch &= ~IN_LEN;
30916     if (!(tp->tty_termios.c_lflag & ECHO)) {
30917           if (ch == ('\n' | IN_EOT) && (tp->tty_termios.c_lflag
30918                                         & (ICANON|ECHONL)) == (ICANON|ECHONL))
30919                   (*tp->tty_echo)(tp, '\n');
30920           return(ch);
30921     }
30922
30923     /* "Reprint" tells if the echo output has been messed up by other output. */
30924     rp = tp->tty_incount == 0 ? FALSE : tp->tty_reprint;
30925
30926     if ((ch & IN_CHAR) < ' ') {
30927           switch (ch & (IN_ESC|IN_EOF|IN_EOT|IN_CHAR)) {
30928               case '\t':
30929                   len = 0;
30930                   do {
30931                           (*tp->tty_echo)(tp, ' ');
30932                           len++;
30933                   } while (len < TAB_SIZE && (tp->tty_position & TAB_MASK) != 0);
30934                   break;
30935               case '\r' | IN_EOT:
30936               case '\n' | IN_EOT:
30937                   (*tp->tty_echo)(tp, ch & IN_CHAR);
30938                   len = 0;
30939                   break;
```

```
30940               default:
30941                   (*tp->tty_echo)(tp, '^');
30942                   (*tp->tty_echo)(tp, '@' + (ch & IN_CHAR));
30943                   len = 2;
30944           }
30945     } else
30946     if ((ch & IN_CHAR) == '\177') {
30947           /* A DEL prints as "^?". */
30948           (*tp->tty_echo)(tp, '^');
30949           (*tp->tty_echo)(tp, '?');
30950           len = 2;
30951     } else {
30952           (*tp->tty_echo)(tp, ch & IN_CHAR);
30953           len = 1;
30954     }
30955     if (ch & IN_EOF) while (len > 0) { (*tp->tty_echo)(tp, '\b'); len--; }
30956
30957     tp->tty_reprint = rp;
30958     return(ch | (len << IN_LSHIFT));
30959   }


30962   /*===========================================================================*
30963    *                                rawecho                                    *
30964    *===========================================================================*/
30965   PRIVATE void rawecho(tp, ch)
30966   register tty_t *tp;
30967   int ch;
30968   {
30969   /* Echo without interpretation if ECHO is set. */
30970     int rp = tp->tty_reprint;
30971     if (tp->tty_termios.c_lflag & ECHO) (*tp->tty_echo)(tp, ch);
30972     tp->tty_reprint = rp;
30973   }


30976   /*===========================================================================*
30977    *                                back_over                                  *
30978    *===========================================================================*/
30979   PRIVATE int back_over(tp)
30980   register tty_t *tp;
30981   {
30982   /* Backspace to previous character on screen and erase it. */
30983     u16_t *head;
30984     int len;
30985
30986     if (tp->tty_incount == 0) return(0);  /* queue empty */
30987     head = tp->tty_inhead;
30988     if (head == tp->tty_inbuf) head = bufend(tp->tty_inbuf);
30989     if (*--head & IN_EOT) return(0);              /* can't erase "line breaks" */
30990     if (tp->tty_reprint) reprint(tp);             /* reprint if messed up */
30991     tp->tty_inhead = head;
30992     tp->tty_incount--;
30993     if (tp->tty_termios.c_lflag & ECHOE) {
30994           len = (*head & IN_LEN) >> IN_LSHIFT;
30995           while (len > 0) {
30996                   rawecho(tp, '\b');
30997                   rawecho(tp, ' ');
30998                   rawecho(tp, '\b');
30999                   len--;
```

```
31000           }
31001       }
31002       return(1);                          /* one character erased */
31003   }


31006   /*===========================================================================*
31007    *                              reprint                                      *
31008    *===========================================================================*/
31009   PRIVATE void reprint(tp)
31010   register tty_t *tp;             /* pointer to tty struct */
31011   {
31012   /* Restore what has been echoed to screen before if the user input has been
31013    * messed up by output, or if REPRINT (^R) is typed.
31014    */
31015     int count;
31016     u16_t *head;
31017
31018     tp->tty_reprint = FALSE;
31019
31020     /* Find the last line break in the input. */
31021     head = tp->tty_inhead;
31022     count = tp->tty_incount;
31023     while (count > 0) {
31024           if (head == tp->tty_inbuf) head = bufend(tp->tty_inbuf);
31025           if (head[-1] & IN_EOT) break;
31026           head--;
31027           count--;
31028     }
31029     if (count == tp->tty_incount) return;         /* no reason to reprint */
31030
31031     /* Show REPRINT (^R) and move to a new line. */
31032     (void) echo(tp, tp->tty_termios.c_cc[VREPRINT] | IN_ESC);
31033     rawecho(tp, '\r');
31034     rawecho(tp, '\n');
31035
31036     /* Reprint from the last break onwards. */
31037     do {
31038           if (head == bufend(tp->tty_inbuf)) head = tp->tty_inbuf;
31039           *head = echo(tp, *head);
31040           head++;
31041           count++;
31042     } while (count < tp->tty_incount);
31043   }


31046   /*===========================================================================*
31047    *                              out_process                                  *
31048    *===========================================================================*/
31049   PUBLIC void out_process(tp, bstart, bpos, bend, icount, ocount)
31050   tty_t *tp;
31051   char *bstart, *bpos, *bend;     /* start/pos/end of circular buffer */
31052   int *icount;                    /* # input chars / input chars used */
31053   int *ocount;                    /* max output chars / output chars used */
31054   {
31055   /* Perform output processing on a circular buffer.  *icount is the number of
31056    * bytes to process, and the number of bytes actually processed on return.
31057    * *ocount is the space available on input and the space used on output.
31058    * (Naturally *icount < *ocount.)  The column position is updated modulo
31059    * the TAB size, because we really only need it for tabs.
```

```
31060     */
31061
31062      int tablen;
31063      int ict = *icount;
31064      int oct = *ocount;
31065      int pos = tp->tty_position;
31066
31067      while (ict > 0) {
31068            switch (*bpos) {
31069            case '\7':
31070                    break;
31071            case '\b':
31072                    pos--;
31073                    break;
31074            case '\r':
31075                    pos = 0;
31076                    break;
31077            case '\n':
31078                    if ((tp->tty_termios.c_oflag & (OPOST|ONLCR))
31079                                                    == (OPOST|ONLCR)) {
31080                            /* Map LF to CR+LF if there is space.  Note that the
31081                             * next character in the buffer is overwritten, so
31082                             * we stop at this point.
31083                             */
31084                            if (oct >= 2) {
31085                                    *bpos = '\r';
31086                                    if (++bpos == bend) bpos = bstart;
31087                                    *bpos = '\n';
31088                                    pos = 0;
31089                                    ict--;
31090                                    oct -= 2;
31091                            }
31092                            goto out_done;  /* no space or buffer got changed */
31093                    }
31094                    break;
31095            case '\t':
31096                    /* Best guess for the tab length. */
31097                    tablen = TAB_SIZE - (pos & TAB_MASK);
31098
31099                    if ((tp->tty_termios.c_oflag & (OPOST|XTABS))
31100                                                    == (OPOST|XTABS)) {
31101                            /* Tabs must be expanded. */
31102                            if (oct >= tablen) {
31103                                    pos += tablen;
31104                                    ict--;
31105                                    oct -= tablen;
31106                                    do {
31107                                            *bpos = ' ';
31108                                            if (++bpos == bend) bpos = bstart;
31109                                    } while (--tablen != 0);
31110                            }
31111                            goto out_done;
31112                    }
31113                    /* Tabs are output directly. */
31114                    pos += tablen;
31115                    break;
31116            default:
31117                    /* Assume any other character prints as one character. */
31118                    pos++;
31119            }
```

```
31120           if (++bpos == bend) bpos = bstart;
31121           ict--;
31122           oct--;
31123     }
31124   out_done:
31125     tp->tty_position = pos & TAB_MASK;
31126
31127     *icount -= ict;       /* [io]ct are the number of chars not used */
31128     *ocount -= oct;       /* *[io]count are the number of chars that are used */
31129   }


31132   /*===========================================================================*
31133    *                              dev_ioctl                                    *
31134    *===========================================================================*/
31135   PRIVATE void dev_ioctl(tp)
31136   tty_t *tp;
31137   {
31138   /* The ioctl's TCSETSW, TCSETSF and TCDRAIN wait for output to finish to make
31139    * sure that an attribute change doesn't affect the processing of current
31140    * output.  Once output finishes the ioctl is executed as in do_ioctl().
31141    */
31142     phys_bytes user_phys;
31143
31144     if (tp->tty_outleft > 0) return;              /* output not finished */
31145
31146     if (tp->tty_ioreq != TCDRAIN) {
31147           if (tp->tty_ioreq == TCSETSF) tty_icancel(tp);
31148           user_phys = proc_vir2phys(proc_addr(tp->tty_ioproc), tp->tty_iovir);
31149           phys_copy(user_phys, vir2phys(&tp->tty_termios),
31150                                       (phys_bytes) sizeof(tp->tty_termios));
31151           setattr(tp);
31152     }
31153     tp->tty_ioreq = 0;
31154     tty_reply(REVIVE, tp->tty_iocaller, tp->tty_ioproc, OK);
31155   }


31158   /*===========================================================================*
31159    *                              setattr                                      *
31160    *===========================================================================*/
31161   PRIVATE void setattr(tp)
31162   tty_t *tp;
31163   {
31164   /* Apply the new line attributes (raw/canonical, line speed, etc.) */
31165     u16_t *inp;
31166     int count;
31167
31168     if (!(tp->tty_termios.c_lflag & ICANON)) {
31169           /* Raw mode; put a "line break" on all characters in the input queue.
31170            * It is undefined what happens to the input queue when ICANON is
31171            * switched off, a process should use TCSAFLUSH to flush the queue.
31172            * Keeping the queue to preserve typeahead is the Right Thing, however
31173            * when a process does use TCSANOW to switch to raw mode.
31174            */
31175           count = tp->tty_eotct = tp->tty_incount;
31176           inp = tp->tty_intail;
31177           while (count > 0) {
31178                   *inp |= IN_EOT;
31179                   if (++inp == bufend(tp->tty_inbuf)) inp = tp->tty_inbuf;
```

```
31180                   --count;
31181           }
31182       }
31183
31184       /* Inspect MIN and TIME. */
31185       lock();
31186       settimer(tp, FALSE);
31187       unlock();
31188       if (tp->tty_termios.c_lflag & ICANON) {
31189             /* No MIN & TIME in canonical mode. */
31190             tp->tty_min = 1;
31191       } else {
31192             /* In raw mode MIN is the number of chars wanted, and TIME how long
31193              * to wait for them.  With interesting exceptions if either is zero.
31194              */
31195             tp->tty_min = tp->tty_termios.c_cc[VMIN];
31196             if (tp->tty_min == 0 && tp->tty_termios.c_cc[VTIME] > 0)
31197                     tp->tty_min = 1;
31198       }
31199
31200       if (!(tp->tty_termios.c_iflag & IXON)) {
31201             /* No start/stop output control, so don't leave output inhibited. */
31202             tp->tty_inhibited = RUNNING;
31203             tp->tty_events = 1;
31204       }
31205
31206       /* Setting the output speed to zero hangs up the phone. */
31207       if (tp->tty_termios.c_ospeed == B0) sigchar(tp, SIGHUP);
31208
31209       /* Set new line speed, character size, etc at the device level. */
31210       (*tp->tty_ioctl)(tp);
31211   }


31214   /*===========================================================================*
31215    *                               tty_reply                                   *
31216    *===========================================================================*/
31217   PUBLIC void tty_reply(code, replyee, proc_nr, status)
31218   int code;                     /* TASK_REPLY or REVIVE */
31219   int replyee;                  /* destination address for the reply */
31220   int proc_nr;                  /* to whom should the reply go? */
31221   int status;                   /* reply code */
31222   {
31223   /* Send a reply to a process that wanted to read or write data. */
31224
31225     message tty_mess;
31226
31227     tty_mess.m_type = code;
31228     tty_mess.REP_PROC_NR = proc_nr;
31229     tty_mess.REP_STATUS = status;
31230     if ((status = send(replyee, &tty_mess)) != OK)
31231           panic("tty_reply failed, status\n", status);
31232   }


31235   /*===========================================================================*
31236    *                               sigchar                                     *
31237    *===========================================================================*/
31238   PUBLIC void sigchar(tp, sig)
31239   register tty_t *tp;
```

```
31240   int sig;                        /* SIGINT, SIGQUIT, SIGKILL or SIGHUP */
31241   {
31242   /* Process a SIGINT, SIGQUIT or SIGKILL char from the keyboard or SIGHUP from
31243    * a tty close, "stty 0", or a real RS-232 hangup.  MM will send the signal to
31244    * the process group (INT, QUIT), all processes (KILL), or the session leader
31245    * (HUP).
31246    */
31247
31248     if (tp->tty_pgrp != 0) cause_sig(tp->tty_pgrp, sig);
31249
31250     if (!(tp->tty_termios.c_lflag & NOFLSH)) {
31251           tp->tty_incount = tp->tty_eotct = 0;    /* kill earlier input */
31252           tp->tty_intail = tp->tty_inhead;
31253           (*tp->tty_ocancel)(tp);                 /* kill all output */
31254           tp->tty_inhibited = RUNNING;
31255           tp->tty_events = 1;
31256     }
31257   }


31260   /*===========================================================================*
31261    *                               tty_icancel                                 *
31262    *===========================================================================*/
31263   PRIVATE void tty_icancel(tp)
31264   register tty_t *tp;
31265   {
31266   /* Discard all pending input, tty buffer or device. */
31267
31268     tp->tty_incount = tp->tty_eotct = 0;
31269     tp->tty_intail = tp->tty_inhead;
31270     (*tp->tty_icancel)(tp);
31271   }


31274   /*===========================================================================*
31275    *                               tty_init                                    *
31276    *===========================================================================*/
31277   PRIVATE void tty_init(tp)
31278   tty_t *tp;                      /* TTY line to initialize. */
31279   {
31280   /* Initialize tty structure and call device initialization routines. */
31281
31282     tp->tty_intail = tp->tty_inhead = tp->tty_inbuf;
31283     tp->tty_min = 1;
31284     tp->tty_termios = termios_defaults;
31285     tp->tty_icancel = tp->tty_ocancel = tp->tty_ioctl = tp->tty_close =
31286                                                             tty_devnop;
31287     if (tp < tty_addr(NR_CONS)) {
31288           scr_init(tp);
31289     } else
31290     if (tp < tty_addr(NR_CONS+NR_RS_LINES)) {
31291           rs_init(tp);
31292     } else {
31293           pty_init(tp);
31294     }
31295   }


31298   /*===========================================================================*
31299    *                               tty_wakeup                                  *
```

```
31300     *===========================================================================*/
31301    PUBLIC void tty_wakeup(now)
31302    clock_t now;                            /* current time */
31303    {
31304    /* Wake up TTY when something interesting is happening on one of the terminal
31305     * lines, like a character arriving on an RS232 line, a key being typed, or
31306     * a timer on a line expiring by TIME.
31307     */
31308      tty_t *tp;
31309
31310      /* Scan the timerlist for expired timers and compute the next timeout time. */
31311      tty_timeout = TIME_NEVER;
31312      while ((tp = tty_timelist) != NULL) {
31313            if (tp->tty_time > now) {
31314                    tty_timeout = tp->tty_time;     /* this timer is next */
31315                    break;
31316            }
31317            tp->tty_min = 0;                        /* force read to succeed */
31318            tp->tty_events = 1;
31319            tty_timelist = tp->tty_timenext;
31320      }
31321
31322      /* Let TTY know there is something afoot. */
31323      interrupt(TTY);
31324    }


31327    /*===========================================================================*
31328     *                              settimer                                     *
31329     *===========================================================================*/
31330    PRIVATE void settimer(tp, on)
31331    tty_t *tp;                      /* line to set or unset a timer on */
31332    int on;                         /* set timer if true, otherwise unset */
31333    {
31334    /* Set or unset a TIME inspired timer.  This function is interrupt sensitive
31335     * due to tty_wakeup(), so it must be called from within lock()/unlock().
31336     */
31337      tty_t **ptp;
31338
31339      /* Take tp out of the timerlist if present. */
31340      for (ptp = &tty_timelist; *ptp != NULL; ptp = &(*ptp)->tty_timenext) {
31341            if (tp == *ptp) {
31342                    *ptp = tp->tty_timenext;        /* take tp out of the list */
31343                    break;
31344            }
31345      }
31346      if (!on) return;                              /* unsetting it is enough */
31347
31348      /* Timeout occurs TIME deciseconds from now. */
31349      tp->tty_time = get_uptime() + tp->tty_termios.c_cc[VTIME] * (HZ/10);
31350
31351      /* Find a new place in the list. */
31352      for (ptp = &tty_timelist; *ptp != NULL; ptp = &(*ptp)->tty_timenext) {
31353            if (tp->tty_time <= (*ptp)->tty_time) break;
31354      }
31355      tp->tty_timenext = *ptp;
31356      *ptp = tp;
31357      if (tp->tty_time < tty_timeout) tty_timeout = tp->tty_time;
31358    }
```

```
31361   /*===========================================================================*
31362    *                              tty_devnop                                   *
31363    *===========================================================================*/
31364   PUBLIC void tty_devnop(tp)
31365   tty_t *tp;
31366   {
31367     /* Some functions need not be implemented at the device level. */
31368   }


31371   #if ENABLE_SRCCOMPAT || ENABLE_BINCOMPAT
31372   /*===========================================================================*
31373    *                              compat_getp                                  *
31374    *===========================================================================*/
31375   PRIVATE int compat_getp(tp, sg)
31376   tty_t *tp;
31377   struct sgttyb *sg;
31378   {
31379   /* Translate an old TIOCGETP to the termios equivalent. */
31380     int flgs;
31381
31382     sg->sg_erase = tp->tty_termios.c_cc[VERASE];
31383     sg->sg_kill = tp->tty_termios.c_cc[VKILL];
31384     sg->sg_ospeed = tspd2sgspd(cfgetospeed(&tp->tty_termios));
31385     sg->sg_ispeed = tspd2sgspd(cfgetispeed(&tp->tty_termios));
31386
31387     flgs = 0;
31388
31389     /* XTABS      - if OPOST and XTABS */
31390     if ((tp->tty_termios.c_oflag & (OPOST|XTABS)) == (OPOST|XTABS))
31391           flgs |= 0006000;
31392
31393     /* BITS5..BITS8  - map directly to CS5..CS8 */
31394     flgs |= (tp->tty_termios.c_cflag & CSIZE) << (8-2);
31395
31396     /* EVENP      - if PARENB and not PARODD */
31397     if ((tp->tty_termios.c_cflag & (PARENB|PARODD)) == PARENB)
31398           flgs |= 0000200;
31399
31400     /* ODDP       - if PARENB and PARODD */
31401     if ((tp->tty_termios.c_cflag & (PARENB|PARODD)) == (PARENB|PARODD))
31402           flgs |= 0000100;
31403
31404     /* RAW        - if not ICANON and not ISIG */
31405     if (!(tp->tty_termios.c_lflag & (ICANON|ISIG)))
31406           flgs |= 0000040;
31407
31408     /* CRMOD      - if ICRNL */
31409     if (tp->tty_termios.c_iflag & ICRNL)
31410           flgs |= 0000020;
31411
31412     /* ECHO       - if ECHO */
31413     if (tp->tty_termios.c_lflag & ECHO)
31414           flgs |= 0000010;
31415
31416     /* CBREAK     - if not ICANON and ISIG */
31417     if ((tp->tty_termios.c_lflag & (ICANON|ISIG)) == ISIG)
31418           flgs |= 0000002;
31419
```

```
31420     sg->sg_flags = flgs;
31421     return(OK);
31422   }


31425   /*===========================================================================*
31426    *                              compat_getc                                  *
31427    *===========================================================================*/
31428   PRIVATE int compat_getc(tp, tc)
31429   tty_t *tp;
31430   struct tchars *tc;
31431   {
31432   /* Translate an old TIOCGETC to the termios equivalent. */
31433
31434     tc->t_intrc = tp->tty_termios.c_cc[VINTR];
31435     tc->t_quitc = tp->tty_termios.c_cc[VQUIT];
31436     tc->t_startc = tp->tty_termios.c_cc[VSTART];
31437     tc->t_stopc = tp->tty_termios.c_cc[VSTOP];
31438     tc->t_brkc = tp->tty_termios.c_cc[VEOL];
31439     tc->t_eofc = tp->tty_termios.c_cc[VEOF];
31440     return(OK);
31441   }


31444   /*===========================================================================*
31445    *                              compat_setp                                  *
31446    *===========================================================================*/
31447   PRIVATE int compat_setp(tp, sg)
31448   tty_t *tp;
31449   struct sgttyb *sg;
31450   {
31451   /* Translate an old TIOCSETP to the termios equivalent. */
31452     struct termios termios;
31453     int flags;
31454
31455     termios = tp->tty_termios;
31456
31457     termios.c_cc[VERASE] = sg->sg_erase;
31458     termios.c_cc[VKILL] = sg->sg_kill;
31459     cfsetispeed(&termios, sgspd2tspd(sg->sg_ispeed & BYTE));
31460     cfsetospeed(&termios, sgspd2tspd(sg->sg_ospeed & BYTE));
31461     flags = sg->sg_flags;
31462
31463     /* Input flags */
31464
31465     /* BRKINT     - not changed */
31466     /* ICRNL      - set if CRMOD is set and not RAW */
31467     /*              (CRMOD also controls output) */
31468     termios.c_iflag &= ~ICRNL;
31469     if ((flags & 0000020) && !(flags & 0000040))
31470           termios.c_iflag |= ICRNL;
31471
31472     /* IGNBRK     - not changed */
31473     /* IGNCR      - forced off (ignoring cr's is not supported) */
31474     termios.c_iflag &= ~IGNCR;
31475
31476     /* IGNPAR     - not changed */
31477     /* INLCR      - forced off (mapping nl's to cr's is not supported) */
31478     termios.c_iflag &= ~INLCR;
31479
```

```
31480      /* INPCK      - not changed */
31481      /* ISTRIP     - not changed */
31482      /* IXOFF      - not changed */
31483      /* IXON       - forced on if not RAW */
31484      termios.c_iflag &= ~IXON;
31485      if (!(flags & 0000040))
31486            termios.c_iflag |= IXON;
31487
31488      /* PARMRK     - not changed */
31489
31490      /* Output flags */
31491
31492      /* OPOST      - forced on if not RAW */
31493      termios.c_oflag &= ~OPOST;
31494      if (!(flags & 0000040))
31495            termios.c_oflag |= OPOST;
31496
31497      /* ONLCR      - forced on if CRMOD */
31498      termios.c_oflag &= ~ONLCR;
31499      if (flags & 0000020)
31500            termios.c_oflag |= ONLCR;
31501
31502      /* XTABS      - forced on if XTABS */
31503      termios.c_oflag &= ~XTABS;
31504      if (flags & 0006000)
31505            termios.c_oflag |= XTABS;
31506
31507      /* CLOCAL     - not changed */
31508      /* CREAD      - forced on (receiver is always enabled) */
31509      termios.c_cflag |= CREAD;
31510
31511      /* CSIZE      - CS5-CS8 correspond directly to BITS5-BITS8 */
31512      termios.c_cflag = (termios.c_cflag & ~CSIZE) | ((flags & 0001400) >> (8-2));
31513
31514      /* CSTOPB     - not changed */
31515      /* HUPCL      - not changed */
31516      /* PARENB     - set if EVENP or ODDP is set */
31517      termios.c_cflag &= ~PARENB;
31518      if (flags & (0000200|0000100))
31519            termios.c_cflag |= PARENB;
31520
31521      /* PARODD     - set if ODDP is set */
31522      termios.c_cflag &= ~PARODD;
31523      if (flags & 0000100)
31524            termios.c_cflag |= PARODD;
31525
31526      /* Local flags */
31527
31528      /* ECHO               - set if ECHO is set */
31529      termios.c_lflag &= ~ECHO;
31530      if (flags & 0000010)
31531            termios.c_lflag |= ECHO;
31532
31533      /* ECHOE      - not changed */
31534      /* ECHOK      - not changed */
31535      /* ECHONL     - not changed */
31536      /* ICANON     - set if neither CBREAK nor RAW */
31537      termios.c_lflag &= ~ICANON;
31538      if (!(flags & (0000002|0000040)))
31539            termios.c_lflag |= ICANON;
```

```
31540
31541     /* IEXTEN     - set if not RAW */
31542     /* ISIG       - set if not RAW */
31543     termios.c_lflag &= ~(IEXTEN|ISIG);
31544     if (!(flags & 0000040))
31545           termios.c_lflag |= (IEXTEN|ISIG);
31546
31547     /* NOFLSH     - not changed */
31548     /* TOSTOP     - not changed */
31549
31550     tp->tty_termios = termios;
31551     setattr(tp);
31552     return(OK);
31553   }


31556   /*===========================================================================*
31557    *                              compat_setc                                  *
31558    *===========================================================================*/
31559   PRIVATE int compat_setc(tp, tc)
31560   tty_t *tp;
31561   struct tchars *tc;
31562   {
31563   /* Translate an old TIOCSETC to the termios equivalent. */
31564     struct termios termios;
31565
31566     termios = tp->tty_termios;
31567
31568     termios.c_cc[VINTR] = tc->t_intrc;
31569     termios.c_cc[VQUIT] = tc->t_quitc;
31570     termios.c_cc[VSTART] = tc->t_startc;
31571     termios.c_cc[VSTOP] = tc->t_stopc;
31572     termios.c_cc[VEOL] = tc->t_brkc;
31573     termios.c_cc[VEOF] = tc->t_eofc;
31574
31575     tp->tty_termios = termios;
31576     setattr(tp);
31577     return(OK);
31578   }


31581   /* Table of termios line speed to sgtty line speed translations.   All termios
31582    * speeds are present even if sgtty didn't know about them.  (Now it does.)
31583    */
31584   PRIVATE struct s2s {
31585     speed_t       tspd;
31586     u8_t          sgspd;
31587   } ts2sgs[] = {
31588     { B0,           0 },
31589     { B50,         50 },
31590     { B75,         75 },
31591     { B110,         1 },
31592     { B134,       134 },
31593     { B200,         2 },
31594     { B300,         3 },
31595     { B600,         6 },
31596     { B1200,       12 },
31597     { B1800,       18 },
31598     { B2400,       24 },
31599     { B4800,       48 },
```

```
31600     { B9600,       96 },
31601     { B19200,     192 },
31602     { B38400,     195 },
31603     { B57600,     194 },
31604     { B115200,    193 },
31605   };
31606
31607   /*==========================================================================*
31608    *                              tspd2sgspd                                  *
31609    *==========================================================================*/
31610   PRIVATE int tspd2sgspd(tspd)
31611   speed_t tspd;
31612   {
31613   /* Translate a termios speed to sgtty speed. */
31614     struct s2s *s;
31615
31616     for (s = ts2sgs; s < ts2sgs + sizeof(ts2sgs)/sizeof(ts2sgs[0]); s++) {
31617           if (s->tspd == tspd) return(s->sgspd);
31618     }
31619     return 96;
31620   }


31623   /*==========================================================================*
31624    *                              sgspd2tspd                                  *
31625    *==========================================================================*/
31626   PRIVATE speed_t sgspd2tspd(sgspd)
31627   int sgspd;
31628   {
31629   /* Translate a sgtty speed to termios speed. */
31630     struct s2s *s;
31631
31632     for (s = ts2sgs; s < ts2sgs + sizeof(ts2sgs)/sizeof(ts2sgs[0]); s++) {
31633           if (s->sgspd == sgspd) return(s->tspd);
31634     }
31635     return B9600;
31636   }


31639   #if ENABLE_BINCOMPAT
31640   /*==========================================================================*
31641    *                              do_ioctl_compat                             *
31642    *==========================================================================*/
31643   PRIVATE void do_ioctl_compat(tp, m_ptr)
31644   tty_t *tp;
31645   message *m_ptr;
31646   {
31647   /* Handle the old sgtty ioctl's that packed the sgtty or tchars struct into
31648    * the Minix message.  Efficient then, troublesome now.
31649    */
31650     int minor, proc, func, result, r;
31651     long flags, erki, spek;
31652     u8_t erase, kill, intr, quit, xon, xoff, brk, eof, ispeed, ospeed;
31653     struct sgttyb sg;
31654     struct tchars tc;
31655     message reply_mess;
31656
31657     minor = m_ptr->TTY_LINE;
31658     proc = m_ptr->PROC_NR;
31659     func = m_ptr->REQUEST;
```

```
31660       spek = m_ptr->m2_l1;
31661       flags = m_ptr->m2_l2;
31662
31663       switch(func)
31664       {
31665         case (('t'<<8) | 8):        /* TIOCGETP */
31666             r = compat_getp(tp, &sg);
31667             erase = sg.sg_erase;
31668             kill = sg.sg_kill;
31669             ispeed = sg.sg_ispeed;
31670             ospeed = sg.sg_ospeed;
31671             flags = sg.sg_flags;
31672             erki = ((long)ospeed<<24) | ((long)ispeed<<16) | ((long)erase<<8) |kill;
31673             break;
31674         case (('t'<<8) | 18):       /* TIOCGETC */
31675             r = compat_getc(tp, &tc);
31676             intr = tc.t_intrc;
31677             quit = tc.t_quitc;
31678             xon = tc.t_startc;
31679             xoff = tc.t_stopc;
31680             brk = tc.t_brkc;
31681             eof = tc.t_eofc;
31682             erki = ((long)intr<<24) | ((long)quit<<16) | ((long)xon<<8) | xoff;
31683             flags = (eof << 8) | brk;
31684             break;
31685         case (('t'<<8) | 17):       /* TIOCSETC */
31686             tc.t_stopc = (spek >> 0) & 0xFF;
31687             tc.t_startc = (spek >> 8) & 0xFF;
31688             tc.t_quitc = (spek >> 16) & 0xFF;
31689             tc.t_intrc = (spek >> 24) & 0xFF;
31690             tc.t_brkc = (flags >> 0) & 0xFF;
31691             tc.t_eofc = (flags >> 8) & 0xFF;
31692             r = compat_setc(tp, &tc);
31693             break;
31694         case (('t'<<8) | 9):        /* TIOCSETP */
31695             sg.sg_erase = (spek >> 8) & 0xFF;
31696             sg.sg_kill = (spek >> 0) & 0xFF;
31697             sg.sg_ispeed = (spek >> 16) & 0xFF;
31698             sg.sg_ospeed = (spek >> 24) & 0xFF;
31699             sg.sg_flags = flags;
31700             r = compat_setp(tp, &sg);
31701             break;
31702         default:
31703             r = ENOTTY;
31704       }
31705       reply_mess.m_type = TASK_REPLY;
31706       reply_mess.REP_PROC_NR = m_ptr->PROC_NR;
31707       reply_mess.REP_STATUS = r;
31708       reply_mess.m2_l1 = erki;
31709       reply_mess.m2_l2 = flags;
31710       send(m_ptr->m_source, &reply_mess);
31711     }
31712     #endif /* ENABLE_BINCOMPAT */
31713     #endif /* ENABLE_SRCCOMPAT || ENABLE_BINCOMPAT */
31714
31715
31716
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/wdeth.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

31800    /*
31801    wdeth.c
31802
31803    Created:        March 14, 1994 by Philip Homburg
31804    */
31805
31806    #include "kernel.h"
31807    #include <net/gen/ether.h>
31808    #include <net/gen/eth_io.h>
31809    #include "assert.h"
31810    INIT_ASSERT
31811    #include "dp8390.h"
31812    #include "wdeth.h"
31813
31814    #if ENABLE_NETWORKING
31815
31816    #if !__minix_vmd
31817    #define debug           0
31818    #endif
31819
31820    #define WET_ETHERNET    0x01            /* Ethernet transceiver */
31821    #define WET_STARLAN     0x02            /* Starlan transceiver */
31822    #define WET_INTERF_CHIP 0x04            /* has a WD83C583 interface chip */
31823    #define WET_BRD_16BIT   0x08            /* 16 bit board */
31824    #define WET_SLT_16BIT   0x10            /* 16 bit slot */
31825    #define WET_790         0x20            /* '790 chip */
31826
31827    static int we_int_table[8]= { 9, 3, 5, 7, 10, 11, 15, 4 };
31828    static int we_790int_table[8]= { 0, 9, 3, 5, 7, 10, 11, 15 };
31829
31830    _PROTOTYPE( static void we_init, (dpeth_t *dep)                         );
31831    _PROTOTYPE( static void we_stop, (dpeth_t *dep)                         );
31832    _PROTOTYPE( static int we_aliasing, (dpeth_t *dep)                      );
31833    _PROTOTYPE( static int we_interface_chip, (dpeth_t *dep)                );
31834    _PROTOTYPE( static int we_16bitboard, (dpeth_t *dep)                    );
31835    _PROTOTYPE( static int we_16bitslot, (dpeth_t *dep)                     );
31836    _PROTOTYPE( static int we_ultra, (dpeth_t *dep)                         );
31837
31838    /*===========================================================================*
31839     *                              wdeth_probe                                  *
31840     *===========================================================================*/
31841    int wdeth_probe(dep)
31842    dpeth_t *dep;
31843    {
31844            int sum;
31845
31846            if (dep->de_linmem == 0)
31847                    return 0;       /* No shared memory, so no WD board */
31848
31849            sum = inb_we(dep, EPL_EA0) + inb_we(dep, EPL_EA1) +
31850                    inb_we(dep, EPL_EA2) + inb_we(dep, EPL_EA3) +
31851                    inb_we(dep, EPL_EA4) + inb_we(dep, EPL_EA5) +
31852                    inb_we(dep, EPL_TLB) + inb_we(dep, EPL_CHKSUM);
31853            if ((sum & 0xFF) != 0xFF)
31854                    return 0;       /* No ethernet board at this address */
```

```
31855
31856           dep->de_initf= we_init;
31857           dep->de_stopf= we_stop;
31858           dep->de_prog_IO= 0;
31859           return 1;
31860   }


31863   /*============================================================================*
31864    *                              we_init                                       *
31865    *============================================================================*/
31866   static void we_init(dep)
31867   dpeth_t *dep;
31868   {
31869           int i, int_indx, int_nr;
31870           int tlb, rambit, revision;
31871           int icr, irr, hwr, b, gcr;
31872           int we_type;
31873           int sendq_nr;
31874
31875           assert(dep->de_mode == DEM_ENABLED);
31876           assert(!(dep->de_flags & DEF_ENABLED));
31877
31878           dep->de_address.ea_addr[0] = inb_we(dep, EPL_EA0);
31879           dep->de_address.ea_addr[1] = inb_we(dep, EPL_EA1);
31880           dep->de_address.ea_addr[2] = inb_we(dep, EPL_EA2);
31881           dep->de_address.ea_addr[3] = inb_we(dep, EPL_EA3);
31882           dep->de_address.ea_addr[4] = inb_we(dep, EPL_EA4);
31883           dep->de_address.ea_addr[5] = inb_we(dep, EPL_EA5);
31884
31885           dep->de_dp8390_port= dep->de_base_port + EPL_DP8390;
31886
31887           dep->de_16bit= 0;
31888
31889           we_type= 0;
31890           we_type |= WET_ETHERNET;        /* assume ethernet */
31891           if (we_ultra(dep))
31892                   we_type |= WET_790;
31893           if (!we_aliasing(dep))
31894           {
31895                   if (we_interface_chip(dep))
31896                           we_type |= WET_INTERF_CHIP;
31897                   if (we_16bitboard(dep))
31898                   {
31899                           we_type |= WET_BRD_16BIT;
31900                           if (we_16bitslot(dep))
31901                                   we_type |= WET_SLT_16BIT;
31902                   }
31903           }
31904           if (we_type & WET_SLT_16BIT)
31905                   dep->de_16bit= 1;
31906
31907           /* look at the on board ram size. */
31908           tlb= inb_we(dep, EPL_TLB);
31909           revision= tlb & E_TLB_REV;
31910           rambit= tlb & E_TLB_RAM;
31911
31912           if (revision < 2)
31913           {
31914                   dep->de_ramsize= 0x2000;                        /* 8K */
```

```
31915                   if (we_type & WET_BRD_16BIT)
31916                           dep->de_ramsize= 0x4000;                /* 16K */
31917                   else if ((we_type & WET_INTERF_CHIP) &&
31918                           inb_we(dep, EPL_ICR) & E_ICR_MEMBIT)
31919                   {
31920                           dep->de_ramsize= 0x8000;                /* 32K */
31921                   }
31922           }
31923           else
31924           {
31925                   if (we_type & WET_BRD_16BIT)
31926                   {
31927                           /* 32K or 16K */
31928                           dep->de_ramsize= rambit ? 0x8000 : 0x4000;
31929                   }
31930                   else
31931                   {
31932                           /* 32K or 8K */
31933                           dep->de_ramsize= rambit ? 0x8000 : 0x2000;
31934                   }
31935           }
31936
31937           if (we_type & WET_790)
31938           {
31939                   outb_we(dep, EPL_MSR, E_MSR_RESET);
31940                   if ((we_type & (WET_BRD_16BIT|WET_SLT_16BIT)) ==
31941                           (WET_BRD_16BIT|WET_SLT_16BIT))
31942                   {
31943                           outb_we(dep, EPL_LAAR, E_LAAR_LAN16E | E_LAAR_MEM16E);
31944                   }
31945           }
31946           else if (we_type & WET_BRD_16BIT)
31947           {
31948                   if (we_type & WET_SLT_16BIT)
31949                   {
31950                           outb_we(dep, EPL_LAAR, E_LAAR_A19 | E_LAAR_SOFTINT |
31951                                   E_LAAR_LAN16E | E_LAAR_MEM16E);
31952                   }
31953                   else
31954                   {
31955                           outb_we(dep, EPL_LAAR, E_LAAR_A19 | E_LAAR_SOFTINT |
31956                                   E_LAAR_LAN16E);
31957                   }
31958           }
31959
31960           if (we_type & WET_790)
31961           {
31962                   outb_we(dep, EPL_MSR, E_MSR_MENABLE);
31963                   hwr= inb_we(dep, EPL_790_HWR);
31964                   outb_we(dep, EPL_790_HWR, hwr | E_790_HWR_SWH);
31965                   b= inb_we(dep, EPL_790_B);
31966                   outb_we(dep, EPL_790_B, ((dep->de_linmem >> 13) & 0x0f) |
31967                           ((dep->de_linmem >> 11) & 0x40) | (b & 0xb0));
31968                   outb_we(dep, EPL_790_HWR, hwr & ~E_790_HWR_SWH);
31969           }
31970           else
31971           {
31972                   outb_we(dep, EPL_MSR, E_MSR_RESET);
31973                   outb_we(dep, EPL_MSR, E_MSR_MENABLE |
31974                           ((dep->de_linmem >> 13) & E_MSR_MEMADDR));
```

```
31975           }
31976
31977           if ((we_type & WET_INTERF_CHIP) && !(we_type & WET_790))
31978           {
31979                   icr= inb_we(dep, EPL_ICR);
31980                   irr= inb_we(dep, EPL_IRR);
31981                   int_indx= (icr & E_ICR_IR2) |
31982                           ((irr & (E_IRR_IR0|E_IRR_IR1)) >> 5);
31983                   int_nr= we_int_table[int_indx];
31984   #if DEBUG
31985    { printf("wdeth.c: encoded irq= %d\n", int_nr); }
31986   #endif
31987                   if (dep->de_irq & DEI_DEFAULT) dep->de_irq= int_nr;
31988
31989                   outb_we(dep, EPL_IRR, irr | E_IRR_IEN);
31990           }
31991           if (we_type & WET_790)
31992           {
31993                   hwr= inb_we(dep, EPL_790_HWR);
31994                   outb_we(dep, EPL_790_HWR, hwr | E_790_HWR_SWH);
31995
31996                   gcr= inb_we(dep, EPL_790_GCR);
31997
31998                   outb_we(dep, EPL_790_HWR, hwr & ~E_790_HWR_SWH);
31999
32000                   int_indx= ((gcr & E_790_GCR_IR2) >> 4) |
32001                           ((gcr & (E_790_GCR_IR1|E_790_GCR_IR0)) >> 2);
32002                   int_nr= we_790int_table[int_indx];
32003   #if DEBUG
32004    { printf("wdeth.c: encoded irq= %d\n", int_nr); }
32005   #endif
32006                   if (dep->de_irq & DEI_DEFAULT) dep->de_irq= int_nr;
32007
32008                   icr= inb_we(dep, EPL_790_ICR);
32009                   outb_we(dep, EPL_790_ICR, icr | E_790_ICR_EIL);
32010           }
32011
32012           /* Strip the "default flag." */
32013           dep->de_irq &= ~DEI_DEFAULT;
32014
32015           if (!debug)
32016           {
32017                   printf("wdeth: WD80%d3 at %X:%d:%lX\n",
32018                           we_type & WET_BRD_16BIT ? 1 : 0,
32019                           dep->de_base_port, dep->de_irq, dep->de_linmem);
32020           }
32021           else
32022           {
32023                   printf("wdeth: Western Digital %s%s card %s%s",
32024                           we_type & WET_BRD_16BIT ? "16-bit " : "",
32025                           we_type & WET_ETHERNET ? "Ethernet" :
32026                           we_type & WET_STARLAN ? "Starlan" : "Network",
32027                           we_type & WET_INTERF_CHIP ? "with an interface chip " : "",
32028                           we_type & WET_SLT_16BIT ? "in a 16-bit slot " : "");
32029                   printf(
32030           "at I/O address 0x%X, memory address 0x%lX, memory size 0x%X, irq %d\n",
32031                           dep->de_base_port, dep->de_linmem, dep->de_ramsize,
32032                           dep->de_irq);
32033           }
32034
```

```
32035           dep->de_offset_page= 0;         /* Shared memory starts at 0 */
32036
32037           /* Allocate one send buffer (1.5KB) per 8KB of on board memory. */
32038           sendq_nr= dep->de_ramsize / 0x2000;
32039           if (sendq_nr < 1)
32040                   sendq_nr= 1;
32041           else if (sendq_nr > SENDQ_NR)
32042                   sendq_nr= SENDQ_NR;
32043           dep->de_sendq_nr= sendq_nr;
32044           for (i= 0; i<sendq_nr; i++)
32045                   dep->de_sendq[i].sq_sendpage= i*SENDQ_PAGES;
32046
32047           dep->de_startpage= i*SENDQ_PAGES;
32048           dep->de_stoppage= dep->de_ramsize / DP_PAGESIZE;
32049   }


32052   /*============================================================================*
32053    *                               we_stop                                      *
32054    *============================================================================*/
32055   static void we_stop(dep)
32056   dpeth_t *dep;
32057   {
32058           if (dep->de_16bit)
32059                   outb_we(dep, EPL_LAAR, E_LAAR_A19 | E_LAAR_LAN16E);
32060           outb_we(dep, EPL_MSR, E_MSR_RESET);
32061           outb_we(dep, EPL_MSR, 0);
32062   }


32065   /*============================================================================*
32066    *                               we_aliasing                                  *
32067    *============================================================================*/
32068   static int we_aliasing(dep)
32069   dpeth_t *dep;
32070   {
32071   /* Determine whether wd8003 hardware performs register aliasing. This implies
32072    * an old WD8003E board. */
32073
32074           if (inb_we(dep, EPL_REG1) != inb_we(dep, EPL_EA1))
32075                   return 0;
32076           if (inb_we(dep, EPL_REG2) != inb_we(dep, EPL_EA2))
32077                   return 0;
32078           if (inb_we(dep, EPL_REG3) != inb_we(dep, EPL_EA3))
32079                   return 0;
32080           if (inb_we(dep, EPL_REG4) != inb_we(dep, EPL_EA4))
32081                   return 0;
32082           if (inb_we(dep, EPL_REG7) != inb_we(dep, EPL_CHKSUM))
32083                   return 0;
32084           return 1;
32085   }


32088   /*============================================================================*
32089    *                               we_interface_chip                            *
32090    *============================================================================*/
32091   static int we_interface_chip(dep)
32092   dpeth_t *dep;
32093   {
32094   /* Determine if the board has an interface chip. */
```

```
32095
32096           outb_we(dep, EPL_GP2, 0x35);
32097           if (inb_we(dep, EPL_GP2) != 0x35)
32098                   return 0;
32099           outb_we(dep, EPL_GP2, 0x3A);
32100           if (inb_we(dep, EPL_GP2) != 0x3A)
32101                   return 0;
32102           return 1;
32103   }


32106   /*============================================================================*
32107    *                               we_16bitboard                                *
32108    *============================================================================*/
32109   static int we_16bitboard(dep)
32110   dpeth_t *dep;
32111   {
32112   /* Determine whether the board is capable of doing 16 bit memory moves.
32113    * If the 16 bit enable bit is unchangable by software we'll assume an
32114    * 8 bit board.
32115    */
32116           int icr;
32117           u8_t tlb;
32118
32119           icr= inb_we(dep, EPL_ICR);
32120
32121           outb_we(dep, EPL_ICR, icr ^ E_ICR_16BIT);
32122           if (inb_we(dep, EPL_ICR) == icr)
32123           {
32124                   tlb= inb_we(dep, EPL_TLB);
32125   #if DEBUG
32126                   printf("wdeth: tlb= 0x%x\n", tlb);
32127   #endif
32128                   return tlb == E_TLB_EB || tlb == E_TLB_E ||
32129                           tlb == E_TLB_SMCE || tlb == E_TLB_SMC8216C;
32130           }
32131           outb_we(dep, EPL_ICR, icr);
32132           return 1;
32133   }


32136   /*============================================================================*
32137    *                               we_16bitslot                                 *
32138    *============================================================================*/
32139   static int we_16bitslot(dep)
32140   dpeth_t *dep;
32141   {
32142   /* Determine if the 16 bit board in plugged into a 16 bit slot.  */
32143           return !!(inb_we(dep, EPL_ICR) & E_ICR_16BIT);
32144   }


32147   /*============================================================================*
32148    *                               we_ultra                                     *
32149    *============================================================================*/
32150   static int we_ultra(dep)
32151   dpeth_t *dep;
32152   {
32153   /* Determine if we has an '790 chip.  */
32154           u8_t tlb;
```

```
32155
32156           tlb= inb_we(dep, EPL_TLB);
32157           return tlb == E_TLB_SMC8216C;
32158   }

32160   #endif /* ENABLE_NETWORKING */
32161
32162   /*
32163    * $PchHeader: /mount/hd2/minix/sys/kernel/ibm/RCS/wdeth.c,v 1.4 1995/01/12 22:03:57 philip Exp $
32164    */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                       src/kernel/wini.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

32200   /*      wini.c - choose a winchester driver.            Author: Kees J. Bot
32201    *                                                              28 May 1994
32202    * Several different winchester drivers may be compiled
32203    * into the kernel, but only one may run.  That one is chosen here using
32204    * the boot variable 'hd'.
32205    */
32206
32207   #include "kernel.h"
32208   #include "driver.h"
32209
32210   #if ENABLE_WINI
32211
32212   /* Map driver name to task function. */
32213   struct hdmap {
32214     char          *name;
32215     task_t        *task;
32216   } hdmap[] = {
32217
32218   #if ENABLE_AT_WINI
32219     { "at",       at_winchester_task      },
32220   #endif
32221
32222   #if ENABLE_BIOS_WINI
32223     { "bios",     bios_winchester_task    },
32224   #endif
32225
32226   #if ENABLE_ESDI_WINI
32227     { "esdi",     esdi_winchester_task    },
32228   #endif
32229
32230   #if ENABLE_XT_WINI
32231     { "xt",       xt_winchester_task      },
32232   #endif
32233
32234   };
32235
32236
32237   /*==========================================================================*
32238    *                              winchester_task                             *
32239    *==========================================================================*/
```

```
32240    PUBLIC void winchester_task()
32241    {
32242      /* Call the default or selected winchester task. */
32243      char *hd;
32244      struct hdmap *map;
32245
32246      hd = k_getenv("hd");
32247
32248      for (map = hdmap; map < hdmap + sizeof(hdmap)/sizeof(hdmap[0]); map++) {
32249            if (hd == NULL || strcmp(hd, map->name) == 0) {
32250                    /* Run the selected winchester task. */
32251                    (*map->task)();
32252            }
32253      }
32254      panic("no hd driver", NO_NUM);
32255    }
32256    #endif /* ENABLE_WINI */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/xt_wini.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
32300    /* This file contains the device dependent part of a driver for the WD
32301     * winchester controller from Western Digital (WX-2 and related controllers).
32302     * It was written by Adri Koppes.
32303     *
32304     * The file contains one entry point:
32305     *
32306     *       xt_winchester_task:    main entry when system is brought up
32307     *
32308     *
32309     * Changes:
32310     *      10 Apr 1987 by Gary Oliver: use with the Western Digital WX-2.
32311     *                ? by Harry McGavran: robust operation on turbo clones.
32312     *                ? by Mike Mitchell: WX-2 auto configure operation.
32313     *       2 May 1992 by Kees J. Bot: device dependent/independent split.
32314     */
32315
32316    #include "kernel.h"
32317    #include "driver.h"
32318    #include "drvlib.h"
32319
32320    #if ENABLE_XT_WINI
32321
32322    /* If the DMA buffer is large enough then use it always. */
32323    #define USE_BUF         (DMA_BUF_SIZE > BLOCK_SIZE)
32324
32325    /* I/O Ports used by winchester disk task. */
32326    #define WIN_DATA       0x320    /* winchester disk controller data register */
32327    #define WIN_STATUS     0x321    /* winchester disk controller status register */
32328    #define WST_REQ        0x001    /* Request bit */
32329    #define WST_INPUT      0x002    /* Set if controller is writing to cpu */
32330    #define WST_BUS        0x004    /* Command/status bit */
32331    #define WST_BUSY       0x008    /* Busy */
32332    #define WST_DRQ        0x010    /* DMA request */
32333    #define WST_IRQ        0x020    /* Interrupt request */
32334    #define WIN_SELECT     0x322    /* winchester disk controller select port */
```

```
32335    #define WIN_DMA        0x323    /* winchester disk controller dma register */
32336    #define DMA_ADDR       0x006    /* port for low 16 bits of DMA address */
32337    #define DMA_TOP        0x082    /* port for top 4 bits of 20-bit DMA addr */
32338    #define DMA_COUNT      0x007    /* port for DMA count (count =  bytes - 1) */
32339    #define DMA_FLIPFLOP   0x00C    /* DMA byte pointer flop-flop */
32340    #define DMA_MODE       0x00B    /* DMA mode port */
32341    #define DMA_INIT       0x00A    /* DMA init port */
32342
32343    /* Winchester disk controller command bytes. */
32344    #define WIN_RECALIBRATE 0x01    /* command for the drive to recalibrate */
32345    #define WIN_SENSE       0x03    /* command for the controller to get its status */
32346    #define WIN_READ        0x08    /* command for the drive to read */
32347    #define WIN_WRITE       0x0a    /* command for the drive to write */
32348    #define WIN_SPECIFY     0x0C    /* command for the controller to accept params  */
32349    #define WIN_ECC_READ    0x0D    /* command for the controller to read ecc length */
32350
32351    #define DMA_INT            3    /* Command with dma and interrupt */
32352    #define INT                2    /* Command with interrupt, no dma */
32353    #define NO_DMA_INT         0    /* Command without dma and interrupt */
32354
32355    /* DMA channel commands. */
32356    #define DMA_READ        0x47    /* DMA read opcode */
32357    #define DMA_WRITE       0x4B    /* DMA write opcode */
32358
32359    /* Parameters for the disk drive. */
32360    #ifndef NR_SECTORS
32361    /* For RLL drives NR_SECTORS has to be defined in the makefile or in config.h.
32362     * There is some hope of getting it from the parameter table for these drives,
32363     * and then this driver should use wn_sectors like at_wini.c.
32364     * Unfortunately it is not standard in XT parameter tables.
32365     */
32366    #define NR_SECTORS        17    /* number of sectors per track */
32367    #endif
32368
32369    /* Error codes */
32370    #define ERR              (-1)   /* general error */
32371    #define ERR_BAD_SECTOR   (-2)   /* block marked bad detected */
32372
32373    /* Miscellaneous. */
32374    #define MAX_DRIVES         2    /* this driver support two drives (hd0 - hd9) */
32375    #define MAX_ERRORS         4    /* how often to try rd/wt before quitting */
32376    #define MAX_RESULTS        4    /* max number of bytes controller returns */
32377    #define NR_DEVICES      (MAX_DRIVES * DEV_PER_DRIVE)
32378    #define SUB_PER_DRIVE   (NR_PARTITIONS * NR_PARTITIONS)
32379    #define NR_SUBDEVS      (MAX_DRIVES * SUB_PER_DRIVE)
32380    #define MAX_WIN_RETRY  32000    /* max # times to try to output to WIN */
32381    #if AUTO_BIOS
32382    #define AUTO_PARAM     0x1AD    /* drive parameter table starts here in sect 0  */
32383    #define AUTO_ENABLE     0x10    /* auto bios enabled bit from status reg */
32384    /* some start up parameters in order to extract the drive parameter table */
32385    /* from the winchester. these should not need changed. */
32386    #define AUTO_CYLS        306    /* default number of cylinders */
32387    #define AUTO_HEADS         4    /* default number of heads */
32388    #define AUTO_RWC         307    /* default reduced write cylinder */
32389    #define AUTO_WPC         307    /* default write precomp cylinder */
32390    #define AUTO_ECC          11    /* default ecc burst */
32391    #define AUTO_CTRL          5    /* default winchester stepping speed byte */
32392    #endif
32393
32394    /* Variables. */
```

```
32395   PRIVATE struct wini {           /* main drive struct, one entry per drive */
32396     unsigned wn_cylinders;        /* number of cylinders */
32397     unsigned wn_heads;            /* number of heads */
32398     unsigned wn_reduced_wr;       /* first cylinder with reduced write current */
32399     unsigned wn_precomp;          /* first cylinder with write precompensation */
32400     unsigned wn_max_ecc;          /* maximum ECC burst length */
32401     unsigned wn_ctlbyte;          /* control byte for COMMANDS (10-Apr-87 GO) */
32402     unsigned wn_open_ct;          /* in-use count */
32403     struct device wn_part[DEV_PER_DRIVE];    /* primary partitions: hd[0-4] */
32404     struct device wn_subpart[SUB_PER_DRIVE]; /* subpartitions: hd[1-4][a-d] */
32405   } wini[MAX_DRIVES], *w_wn;
32406
32407   PRIVATE struct trans {
32408     struct iorequest_s *tr_iop;   /* belongs to this I/O request */
32409     unsigned long tr_block;       /* first sector to transfer */
32410     unsigned tr_count;            /* byte count */
32411     phys_bytes tr_phys;           /* user physical address */
32412     phys_bytes tr_dma;            /* DMA physical address */
32413   } wtrans[NR_IOREQS];
32414
32415   PRIVATE int w_need_reset = FALSE;       /* set when controller must be reset */
32416   PRIVATE int nr_drives;                  /* Number of drives */
32417   PRIVATE int w_switches;                 /* Drive type switches */
32418   PRIVATE struct trans *w_tp;             /* to add transfer requests */
32419   PRIVATE unsigned w_count;               /* number of bytes to transfer */
32420   PRIVATE unsigned long w_nextblock;      /* next block on disk to transfer */
32421   PRIVATE int w_opcode;                   /* DEV_READ or DEV_WRITE */
32422   PRIVATE int w_drive;                    /* selected drive */
32423   PRIVATE struct device *w_dv;            /* device's base and size */
32424   PRIVATE char w_results[MAX_RESULTS];/* the controller can give lots of output */
32425
32426
32427   FORWARD _PROTOTYPE( struct device *w_prepare, (int device) );
32428   FORWARD _PROTOTYPE( char *w_name, (void) );
32429   FORWARD _PROTOTYPE( int w_schedule, (int proc_nr, struct iorequest_s *iop) );
32430   FORWARD _PROTOTYPE( int w_finish, (void) );
32431   FORWARD _PROTOTYPE( void w_dma_setup, (struct trans *tp, unsigned count) );
32432   FORWARD _PROTOTYPE( int w_transfer, (struct trans *tp, unsigned count) );
32433   FORWARD _PROTOTYPE( int win_results, (void) );
32434   FORWARD _PROTOTYPE( void win_out, (int val) );
32435   FORWARD _PROTOTYPE( int w_reset, (void) );
32436   FORWARD _PROTOTYPE( int w_handler, (int irq) );
32437   FORWARD _PROTOTYPE( int win_specify, (int drive) );
32438   FORWARD _PROTOTYPE( int check_init, (void) );
32439   FORWARD _PROTOTYPE( int read_ecc, (void) );
32440   FORWARD _PROTOTYPE( int hd_wait, (int bits) );
32441   FORWARD _PROTOTYPE( int com_out, (int mode, u8_t *command) );
32442   FORWARD _PROTOTYPE( void init_params, (void) );
32443   FORWARD _PROTOTYPE( int w_do_open, (struct driver *dp, message *m_ptr) );
32444   FORWARD _PROTOTYPE( int w_do_close, (struct driver *dp, message *m_ptr) );
32445   FORWARD _PROTOTYPE( void w_init, (void) );
32446   FORWARD _PROTOTYPE( void copy_param, (char *src, struct wini *dest) );
32447   FORWARD _PROTOTYPE( void w_geometry, (struct partition *entry));
32448
32449
32450   /* Entry points to this driver. */
32451   PRIVATE struct driver w_dtab = {
32452     w_name,       /* current device's name */
32453     w_do_open,    /* open or mount request, initialize device */
32454     w_do_close,   /* release device */
```

```
32455     do_diocntl,   /* get or set a partition's geometry */
32456     w_prepare,    /* prepare for I/O on a given minor device */
32457     w_schedule,   /* precompute cylinder, head, sector, etc. */
32458     w_finish,     /* do the I/O */
32459     nop_cleanup,  /* no cleanup needed */
32460     w_geometry    /* tell the geometry of the disk */
32461   };
32462
32463
32464   /*============================================================================*
32465    *                               xt_winchester_task                           *
32466    *============================================================================*/
32467   PUBLIC void xt_winchester_task()
32468   {
32469     init_params();
32470
32471     put_irq_handler(XT_WINI_IRQ, w_handler);
32472     enable_irq(XT_WINI_IRQ);      /* ready for winchester interrupts */
32473
32474     driver_task(&w_dtab);
32475   }


32478   /*============================================================================*
32479    *                               w_prepare                                    *
32480    *============================================================================*/
32481   PRIVATE struct device *w_prepare(device)
32482   int device;
32483   {
32484   /* Prepare for I/O on a device. */
32485
32486     /* Nothing to transfer as yet. */
32487     w_count = 0;
32488
32489     if (device < NR_DEVICES) {                    /* hd0, hd1, ... */
32490           w_drive = device / DEV_PER_DRIVE;       /* save drive number */
32491           w_wn = &wini[w_drive];
32492           w_dv = &w_wn->wn_part[device % DEV_PER_DRIVE];
32493     } else
32494     if ((unsigned) (device -= MINOR_hd1a) < NR_SUBDEVS) { /* hd1a, hd1b, ... */
32495           w_drive = device / SUB_PER_DRIVE;
32496           w_wn = &wini[w_drive];
32497           w_dv = &w_wn->wn_subpart[device % SUB_PER_DRIVE];
32498     } else {
32499           return(NIL_DEV);
32500     }
32501     return(w_drive < nr_drives ? w_dv : NIL_DEV);
32502   }


32505   /*============================================================================*
32506    *                               w_name                                       *
32507    *============================================================================*/
32508   PRIVATE char *w_name()
32509   {
32510   /* Return a name for the current device. */
32511     static char name[] = "xt-hd5";
32512
32513     name[5] = '0' + w_drive * DEV_PER_DRIVE;
32514     return name;
```

```
32515   }


32518   /*===========================================================================*
32519    *                              w_schedule                                   *
32520    *===========================================================================*/
32521   PRIVATE int w_schedule(proc_nr, iop)
32522   int proc_nr;                    /* process doing the request */
32523   struct iorequest_s *iop;        /* pointer to read or write request */
32524   {
32525   /* Gather I/O requests on consecutive blocks so they may be read/written
32526    * in one command if using a buffer.  Check and gather all the requests
32527    * and try to finish them as fast as possible if unbuffered.
32528    */
32529     int r, opcode;
32530     unsigned long pos;
32531     unsigned nbytes, count, dma_count;
32532     unsigned long block;
32533     phys_bytes user_phys, dma_phys;
32534
32535     /* This many bytes to read/write */
32536     nbytes = iop->io_nbytes;
32537     if ((nbytes & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);
32538
32539     /* From/to this position on the device */
32540     pos = iop->io_position;
32541     if ((pos & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);
32542
32543     /* To/from this user address */
32544     user_phys = numap(proc_nr, (vir_bytes) iop->io_buf, nbytes);
32545     if (user_phys == 0) return(iop->io_nbytes = EINVAL);
32546
32547     /* Read or write? */
32548     opcode = iop->io_request & ~OPTIONAL_IO;
32549
32550     /* Which block on disk and how close to EOF? */
32551     if (pos >= w_dv->dv_size) return(OK);         /* At EOF */
32552     if (pos + nbytes > w_dv->dv_size) nbytes = w_dv->dv_size - pos;
32553     block = (w_dv->dv_base + pos) >> SECTOR_SHIFT;
32554
32555     if (USE_BUF && w_count > 0 && block != w_nextblock) {
32556           /* This new request can't be chained to the job being built */
32557           if ((r = w_finish()) != OK) return(r);
32558     }
32559
32560     /* The next consecutive block */
32561     if (USE_BUF) w_nextblock = block + (nbytes >> SECTOR_SHIFT);
32562
32563     /* While there are "unscheduled" bytes in the request: */
32564     do {
32565           count = nbytes;
32566
32567           if (USE_BUF) {
32568                   if (w_count == DMA_BUF_SIZE) {
32569                           /* Can't transfer more than the buffer allows. */
32570                           if ((r = w_finish()) != OK) return(r);
32571                   }
32572
32573                   if (w_count + count > DMA_BUF_SIZE)
32574                           count = DMA_BUF_SIZE - w_count;
```

```
32575           } else {
32576                   if (w_tp == wtrans + NR_IOREQS) {
32577                           /* All transfer slots in use. */
32578                           if ((r = w_finish()) != OK) return(r);
32579                   }
32580           }
32581
32582           if (w_count == 0) {
32583                   /* The first request in a row, initialize. */
32584                   w_opcode = opcode;
32585                   w_tp = wtrans;
32586           }
32587
32588           if (USE_BUF) {
32589                   dma_phys = tmp_phys + w_count;
32590           } else {
32591                   /* Memory chunk to DMA. */
32592                   dma_phys = user_phys;
32593                   dma_count = dma_bytes_left(dma_phys);
32594
32595                   if (dma_count < count) {
32596                           /* Nearing a 64K boundary. */
32597                           if (dma_count >= SECTOR_SIZE) {
32598                                   /* Can read a few sectors before hitting the
32599                                    * boundary.
32600                                    */
32601                                   count = dma_count & ~SECTOR_MASK;
32602                           } else {
32603                                   /* Must use the special buffer for this. */
32604                                   count = SECTOR_SIZE;
32605                                   dma_phys = tmp_phys;
32606                           }
32607                   }
32608           }
32609
32610           /* Store I/O parameters */
32611           w_tp->tr_iop = iop;
32612           w_tp->tr_block = block;
32613           w_tp->tr_count = count;
32614           w_tp->tr_phys = user_phys;
32615           w_tp->tr_dma = dma_phys;
32616
32617           /* Update counters */
32618           w_tp++;
32619           w_count += count;
32620           block += count >> SECTOR_SHIFT;
32621           user_phys += count;
32622           nbytes -= count;
32623     } while (nbytes > 0);
32624
32625     return(OK);
32626   }


32629   /*============================================================================*
32630    *                              w_finish                                      *
32631    *============================================================================*/
32632   PRIVATE int w_finish()
32633   {
32634   /* Carry out the I/O requests gathered in wtrans[]. */
```

```
32635
32636     struct trans *tp = wtrans, *tp2;
32637     unsigned count;
32638     int r, errors = 0, many = USE_BUF;
32639
32640     if (w_count == 0) return(OK); /* Spurious finish. */
32641
32642     do {
32643           if (w_opcode == DEV_WRITE) {
32644                   tp2 = tp;
32645                   count = 0;
32646                   do {
32647                           if (USE_BUF || tp2->tr_dma == tmp_phys) {
32648                                   phys_copy(tp2->tr_phys, tp2->tr_dma,
32649                                                   (phys_bytes) tp2->tr_count);
32650                           }
32651                           count += tp2->tr_count;
32652                           tp2++;
32653                   } while (many && count < w_count);
32654           } else {
32655                   count = many ? w_count : tp->tr_count;
32656           }
32657
32658           /* First check to see if a reset is needed. */
32659           if (w_need_reset) w_reset();
32660
32661           /* Now set up the DMA chip. */
32662           w_dma_setup(tp, count);
32663
32664           /* Perform the transfer. */
32665           r = w_transfer(tp, count);
32666
32667           if (r != OK) {
32668                   /* An error occurred, try again block by block unless */
32669                   if (r == ERR_BAD_SECTOR || ++errors == MAX_ERRORS)
32670                           return(tp->tr_iop->io_nbytes = EIO);
32671
32672                   /* Reset if halfway, but bail out if optional I/O. */
32673                   if (errors == MAX_ERRORS / 2) {
32674                           w_need_reset = TRUE;
32675                           if (tp->tr_iop->io_request & OPTIONAL_IO)
32676                                   return(tp->tr_iop->io_nbytes = EIO);
32677                   }
32678                   many = 0;
32679                   continue;
32680           }
32681           errors = 0;
32682
32683           w_count -= count;
32684
32685           do {
32686                   if (w_opcode == DEV_READ) {
32687                           if (USE_BUF || tp->tr_dma == tmp_phys) {
32688                                   phys_copy(tp->tr_dma, tp->tr_phys,
32689                                                   (phys_bytes) tp->tr_count);
32690                           }
32691                   }
32692                   tp->tr_iop->io_nbytes -= tp->tr_count;
32693                   count -= tp->tr_count;
32694                   tp++;
```

```
32695           } while (count > 0);
32696     } while (w_count > 0);
32697
32698     return(OK);
32699   }


32702   /*==========================================================================*
32703    *                              w_dma_setup                                 *
32704    *==========================================================================*/
32705   PRIVATE void w_dma_setup(tp, count)
32706   struct trans *tp;               /* pointer to the transfer struct */
32707   unsigned count;                 /* bytes to transfer */
32708   {
32709   /* The IBM PC can perform DMA operations by using the DMA chip.  To use it,
32710    * the DMA (Direct Memory Access) chip is loaded with the 20-bit memory address
32711    * to by read from or written to, the byte count minus 1, and a read or write
32712    * opcode.  This routine sets up the DMA chip.  Note that the chip is not
32713    * capable of doing a DMA across a 64K boundary (e.g., you can't read a
32714    * 512-byte block starting at physical address 65520).
32715    */
32716
32717     /* Set up the DMA registers. */
32718     out_byte(DMA_FLIPFLOP, 0);            /* write anything to reset it */
32719     out_byte(DMA_MODE, w_opcode == DEV_WRITE ? DMA_WRITE : DMA_READ);
32720     out_byte(DMA_ADDR, (int) tp->tr_dma >>  0);
32721     out_byte(DMA_ADDR, (int) tp->tr_dma >>  8);
32722     out_byte(DMA_TOP, (int) (tp->tr_dma >> 16));
32723     out_byte(DMA_COUNT, (count - 1) >> 0);
32724     out_byte(DMA_COUNT, (count - 1) >> 8);
32725   }


32728   /*==========================================================================*
32729    *                              w_transfer                                  *
32730    *==========================================================================*/
32731   PRIVATE int w_transfer(tp, count)
32732   struct trans *tp;               /* pointer to the transfer struct */
32733   unsigned count;                 /* transferring count bytes */
32734   {
32735   /* Read or write count bytes starting with tp->tr_block. */
32736
32737     unsigned cylinder, sector, head, secspcyl = w_wn->wn_heads * NR_SECTORS;
32738     u8_t command[6];
32739     message mess;
32740
32741     cylinder = tp->tr_block / secspcyl;
32742     head = (tp->tr_block % secspcyl) / NR_SECTORS;
32743     sector = tp->tr_block % NR_SECTORS;
32744
32745     /* The command is issued by outputting 6 bytes to the controller chip. */
32746     command[0] = w_opcode == DEV_WRITE ? WIN_WRITE : WIN_READ;
32747     command[1] = head | (w_drive << 5);
32748     command[2] = ((cylinder & 0x0300) >> 2) | sector;
32749     command[3] = cylinder & 0xFF;
32750     command[4] = count >> SECTOR_SHIFT;
32751     command[5] = w_wn->wn_ctlbyte;
32752
32753     if (com_out(DMA_INT, command) != OK)
32754           return(ERR);
```

```
32755
32756     out_byte(DMA_INIT, 3);        /* initialize DMA */
32757
32758     /* Block, waiting for disk interrupt. */
32759     receive(HARDWARE, &mess);
32760
32761     /* Get controller status and check for errors. */
32762     if (win_results() == OK)
32763           return(OK);
32764     if ((w_results[0] & 63) == 24)
32765           read_ecc();
32766     else
32767           w_need_reset = TRUE;
32768     return(ERR);
32769   }


32772   /*============================================================================*
32773    *                              win_results                                   *
32774    *============================================================================*/
32775   PRIVATE int win_results()
32776   {
32777   /* Extract results from the controller after an operation. */
32778
32779     int i, status;
32780     u8_t command[6];
32781
32782     status = in_byte(WIN_DATA);
32783     out_byte(WIN_DMA, 0);
32784     if (!(status & 2))            /* Test "error" bit */
32785           return(OK);
32786     command[0] = WIN_SENSE;
32787     command[1] = w_drive << 5;
32788     if (com_out(NO_DMA_INT, command) != OK)
32789           return(ERR);
32790
32791     /* Loop, extracting bytes from WIN */
32792     for (i = 0; i < MAX_RESULTS; i++) {
32793           if (hd_wait(WST_REQ) != OK)
32794                   return(ERR);
32795           status = in_byte(WIN_DATA);
32796           w_results[i] = status & BYTE;
32797     }
32798     if (hd_wait(WST_REQ) != OK)   /* Missing from                 */
32799            return (ERR);          /* Original.  11-Apr-87 G.O.    */
32800
32801     status = in_byte(WIN_DATA);    /* Read "error" flag */
32802
32803     if (((status & 2) != 0) || (w_results[0] & 0x3F)) {
32804           return(ERR);
32805     } else
32806           return(OK);
32807   }


32810   /*============================================================================*
32811    *                              win_out                                       *
32812    *============================================================================*/
32813   PRIVATE void win_out(val)
32814   int val;               /* write this byte to winchester disk controller */
```

```
32815   {
32816   /* Output a byte to the controller.  This is not entirely trivial, since you
32817    * can only write to it when it is listening, and it decides when to listen.
32818    * If the controller refuses to listen, the WIN chip is given a hard reset.
32819    */
32820     int r;
32821
32822     if (w_need_reset) return;     /* if controller is not listening, return */
32823
32824     do {
32825           r = in_byte(WIN_STATUS);
32826     } while((r & (WST_REQ | WST_BUSY)) == WST_BUSY);
32827
32828     out_byte(WIN_DATA, val);
32829   }


32832   /*===========================================================================*
32833    *                               w_reset                                     *
32834    *===========================================================================*/
32835   PRIVATE int w_reset()
32836   {
32837   /* Issue a reset to the controller.  This is done after any catastrophe,
32838    * like the controller refusing to respond.
32839    */
32840
32841     int r, i, drive;
32842     u8_t command[6];
32843     message mess;
32844
32845     /* Strobe reset bit low. */
32846     out_byte(WIN_STATUS, 0);
32847
32848     milli_delay(5);       /* Wait for a while */
32849
32850     out_byte(WIN_SELECT, 0);      /* Issue select pulse */
32851     for (i = 0; i < MAX_WIN_RETRY; i++) {
32852           r = in_byte(WIN_STATUS);
32853           if (r & (WST_DRQ | WST_IRQ))
32854                   return(ERR);
32855
32856           if ((r & (WST_BUSY | WST_BUS | WST_REQ)) ==
32857                   (WST_BUSY | WST_BUS | WST_REQ))
32858                   break;
32859     }
32860
32861     if (i == MAX_WIN_RETRY) {
32862           printf("%s: reset failed, status = %x\n", w_name(), r);
32863           return(ERR);
32864     }
32865
32866     /* Reset succeeded.  Tell WIN drive parameters. */
32867     w_need_reset = FALSE;
32868
32869     for (drive = 0; drive < nr_drives; drive++) {
32870           if (win_specify(drive) != OK)
32871                   return (ERR);
32872
32873           command[0] = WIN_RECALIBRATE;
32874           command[1] = drive << 5;
```

```
32875           command[5] = wini[drive].wn_ctlbyte;
32876
32877           if (com_out(INT, command) != OK)
32878                   return(ERR);
32879
32880           receive(HARDWARE, &mess);
32881
32882           if (win_results() != OK) {
32883                   /* No actual drive present? */
32884                   nr_drives = drive;
32885           }
32886     }
32887     return(nr_drives > 0 ? OK : ERR);
32888   }


32891   /*============================================================================*
32892    *                              w_handler                                     *
32893    *============================================================================*/
32894   PRIVATE int w_handler(irq)
32895   int irq;
32896   {
32897   /* Disk interrupt, send message to winchester task and reenable interrupts. */
32898
32899     int r, i;
32900
32901     out_byte(DMA_INIT, 0x07);     /* Disable int from DMA */
32902
32903     for (i = 0; i < MAX_WIN_RETRY; ++i) {
32904           r = in_byte(WIN_STATUS);
32905           if (r & WST_IRQ)
32906                   break;          /* Exit if end of int */
32907     }
32908
32909     interrupt(WINCHESTER);
32910     return 1;
32911   }


32914   /*============================================================================*
32915    *                              win_specify                                   *
32916    *============================================================================*/
32917   PRIVATE int win_specify(drive)
32918   int drive;
32919   {
32920     struct wini *wn = &wini[drive];
32921     u8_t command[6];
32922
32923     command[0] = WIN_SPECIFY;             /* Specify some parameters */
32924     command[1] = drive << 5;              /* Drive number */
32925
32926     if (com_out(NO_DMA_INT, command) != OK)       /* Output command block */
32927           return(ERR);
32928
32929           /* No. of cylinders (high byte) */
32930     win_out(wn->wn_cylinders >> 8);
32931
32932           /* No. of cylinders (low byte) */
32933     win_out(wn->wn_cylinders);
32934
```

```
32935           /* No. of heads */
32936     win_out(wn->wn_heads);
32937
32938           /* Start reduced write (high byte) */
32939     win_out(wn->wn_reduced_wr >> 8);
32940
32941           /* Start reduced write (low byte) */
32942     win_out(wn->wn_reduced_wr);
32943
32944           /* Start write precompensation (high byte) */
32945     win_out(wn->wn_precomp >> 8);
32946
32947           /* Start write precompensation (low byte) */
32948     win_out(wn->wn_precomp);
32949
32950           /* Ecc burst length */
32951     win_out(wn->wn_max_ecc);
32952
32953     if (check_init() != OK) {  /* See if controller accepted parameters */
32954           w_need_reset = TRUE;
32955           return(ERR);
32956     } else
32957           return(OK);
32958   }


32961   /*============================================================================*
32962    *                              check_init                                    *
32963    *============================================================================*/
32964   PRIVATE int check_init()
32965   {
32966   /* Routine to check if controller accepted the parameters */
32967     int r, s;
32968
32969     if (hd_wait(WST_REQ | WST_INPUT) == OK) {
32970           r = in_byte(WIN_DATA);
32971
32972           do {
32973                   s = in_byte(WIN_STATUS);
32974           } while(s & WST_BUSY);          /* Loop while still busy */
32975
32976           if (r & 2)              /* Test error bit */
32977                   return(ERR);
32978           else
32979                   return(OK);
32980     } else
32981           return (ERR);   /* Missing from original: 11-Apr-87 G.O. */
32982   }


32985   /*============================================================================*
32986    *                              read_ecc                                      *
32987    *============================================================================*/
32988   PRIVATE int read_ecc()
32989   {
32990   /* Read the ecc burst-length and let the controller correct the data */
32991
32992     int r;
32993     u8_t command[6];
32994
```

```
32995       command[0] = WIN_ECC_READ;
32996       if (com_out(NO_DMA_INT, command) == OK && hd_wait(WST_REQ) == OK) {
32997             r = in_byte(WIN_DATA);
32998             if (hd_wait(WST_REQ) == OK) {
32999                     r = in_byte(WIN_DATA);
33000                     if (r & 1)
33001                             w_need_reset = TRUE;
33002             }
33003       }
33004       return(ERR);
33005     }


33008     /*===========================================================================*
33009      *                              hd_wait                                      *
33010      *===========================================================================*/
33011     PRIVATE int hd_wait(bits)
33012     int bits;
33013     {
33014     /* Wait until the controller is ready to receive a command or send status */
33015
33016       int r, i = 0;
33017
33018       do {
33019             r = in_byte(WIN_STATUS) & bits;
33020       } while ((i++ < MAX_WIN_RETRY) && r != bits);         /* Wait for ALL bits */
33021
33022       if (i >= MAX_WIN_RETRY) {
33023             w_need_reset = TRUE;
33024             return(ERR);
33025       } else
33026             return(OK);
33027     }


33030     /*===========================================================================*
33031      *                              com_out                                      *
33032      *===========================================================================*/
33033     PRIVATE int com_out(mode, commandp)
33034     int mode;
33035     u8_t *commandp;
33036     {
33037     /* Output the command block to the winchester controller and return status */
33038
33039       int i, r;
33040
33041       out_byte(WIN_DMA, mode);
33042       out_byte(WIN_SELECT, mode);
33043       for (i = 0; i < MAX_WIN_RETRY; i++) {
33044             r = in_byte(WIN_STATUS);
33045             if (r & WST_BUSY)
33046                     break;
33047       }
33048
33049       if (i == MAX_WIN_RETRY) {
33050             w_need_reset = TRUE;
33051             return(ERR);
33052       }
33053
33054
```

```
33055       for (i = 0; i < 6; i++) {
33056             if (hd_wait(WST_REQ) != OK)
33057                   break;          /* No data request pending */
33058
33059             r = in_byte(WIN_STATUS);
33060
33061             if ((r & (WST_BUSY | WST_BUS | WST_INPUT)) !=
33062                     (WST_BUSY | WST_BUS))
33063                   break;
33064
33065             out_byte(WIN_DATA, commandp[i]);
33066       }
33067
33068       if (i != 6)
33069             return(ERR);
33070       else
33071             return(OK);
33072     }


33075     /*==========================================================================*
33076      *                              init_params                                 *
33077      *==========================================================================*/
33078     PRIVATE void init_params()
33079     {
33080     /* This routine is called at startup to initialize the number of drives and
33081      * the controller.
33082      */
33083       u16_t parv[2];
33084       unsigned int drive;
33085       int dtype;
33086       phys_bytes address, buf_phys;
33087       char buf[16];
33088
33089       /* Get the number of drives from the bios */
33090       buf_phys = vir2phys(buf);
33091       phys_copy(0x475L, buf_phys, 1L);
33092       nr_drives = buf[0] & 0xFF;
33093       if (nr_drives > MAX_DRIVES) nr_drives = MAX_DRIVES;
33094
33095       /* Read the switches from the controller */
33096       w_switches = in_byte(WIN_SELECT);
33097
33098     #if AUTO_BIOS
33099       /* If no auto configuration or not enabled then go to the ROM. */
33100       if (!(w_switches & AUTO_ENABLE)) {
33101     #endif
33102             for (drive = 0; drive < nr_drives; drive++) {
33103                   /* Calculate the drive type */
33104                   dtype = (w_switches >> (2 * drive)) & 03;
33105
33106                   /* Copy the BIOS parameter vector */
33107                   phys_copy(WINI_0_PARM_VEC * 4L, vir2phys(parv), 4L);
33108
33109                   /* Calculate the parameters' address and copy them to buf */
33110                   address = hclick_to_physb(parv[1]) + parv[0] + 16 * dtype;
33111                   phys_copy(address, buf_phys, 16L);
33112
33113                   /* Copy the parameters to the structure of the drive. */
33114                   copy_param(buf, &wini[drive]);
```

```
33115           }
33116   #if AUTO_BIOS
33117     }
33118   #endif
33119   }


33122   /*==========================================================================*
33123    *                              w_do_open                                   *
33124    *==========================================================================*/
33125   PRIVATE int w_do_open(dp, m_ptr)
33126   struct driver *dp;
33127   message *m_ptr;
33128   {
33129   /* Device open: Initialize the controller and read the partition table. */
33130
33131     static int init_done = FALSE;
33132
33133     if (!init_done) { w_init(); init_done = TRUE; }
33134
33135     if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
33136
33137     if (w_wn->wn_open_ct++ == 0) {
33138           /* Partition the disk. */
33139           partition(&w_dtab, w_drive * DEV_PER_DRIVE, P_PRIMARY);
33140     }
33141     return(OK);
33142   }


33145   /*==========================================================================*
33146    *                              w_do_close                                  *
33147    *==========================================================================*/
33148   PRIVATE int w_do_close(dp, m_ptr)
33149   struct driver *dp;
33150   message *m_ptr;
33151   {
33152   /* Device close: Release a device. */
33153
33154     if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
33155     w_wn->wn_open_ct--;
33156     return(OK);
33157   }


33160   /*==========================================================================*
33161    *                              w_init                                      *
33162    *==========================================================================*/
33163   PRIVATE void w_init()
33164   {
33165     /* Initialize the controller. */
33166
33167     int drive;
33168     struct wini *wn;
33169   #if AUTO_BIOS
33170     message mess;
33171
33172     for (drive = 0; drive < nr_drives; drive++) {
33173           /* Get the drive parameters from sector zero of the drive if the
33174            * autoconfig mode of the controller has been selected.
```

```
33175             */
33176            if (w_switches & AUTO_ENABLE) {
33177                    /* Set up some phony parameters so that we can read the
33178                     * first sector from the winchester.  All drives will have
33179                     * one cylinder and one head but set up initially to the
33180                     * mini scribe drives from IBM.
33181                     */
33182                    wn = &wini[drive];
33183                    wn->wn_cylinders = AUTO_CYLS;
33184                    wn->wn_heads = AUTO_HEADS;
33185                    wn->wn_reduced_wr = AUTO_RWC;
33186                    wn->wn_precomp = AUTO_WPC;
33187                    wn->wn_max_ecc = AUTO_ECC;
33188                    wn->wn_ctlbyte = AUTO_CTRL;
33189                    wn->wn_part[0].dv_size = SECTOR_SIZE;
33190            }
33191      }
33192
33193      /* Initialize controller to read parameters from the drives. */
33194      if (nr_drives > 0 && w_reset() != OK) nr_drives = 0;
33195
33196      for (drive = 0; drive < nr_drives; drive++) {
33197            if (w_switches & AUTO_ENABLE) {
33198                    /* read the first sector from the drive */
33199                    mess.DEVICE = drive * DEV_PER_DRIVE;
33200                    mess.POSITION = 0L;
33201                    mess.COUNT = SECTOR_SIZE;
33202                    mess.ADDRESS = (char *) tmp_buf;
33203                    mess.PROC_NR = WINCHESTER;
33204                    mess.m_type = DEV_READ;
33205                    if (do_rdwt(&w_dtab, &mess) != SECTOR_SIZE) {
33206                            printf("%s: can't read parameters\n", w_name());
33207                            nr_drives = drive;
33208                            break;
33209                    }
33210
33211                    /* save the parameter tables for later use */
33212                    copy_param(&tmp_buf[AUTO_PARAM], &wini[drive]);
33213            }
33214      }
33215    #endif
33216
33217      if (nr_drives > 0 && w_reset() != OK) nr_drives = 0;
33218
33219      /* Set the size of each disk. */
33220      for (drive = 0; drive < nr_drives; drive++) {
33221            (void) w_prepare(drive * DEV_PER_DRIVE);
33222            wn = w_wn;
33223            wn->wn_part[0].dv_size = ((unsigned long) wn->wn_cylinders *
33224                                      wn->wn_heads * NR_SECTORS) << SECTOR_SHIFT;
33225            printf("%s: %d cylinders, %d heads, %d sectors per track\n",
33226                    w_name(), wn->wn_cylinders, wn->wn_heads, NR_SECTORS);
33227      }
33228    }


33231    /*============================================================================*
33232     *                              copy_param                                    *
33233     *============================================================================*/
33234    PRIVATE void copy_param(src, dest)
```

```
33235    char *src;
33236    struct wini *dest;
33237    {
33238    /* This routine copies the parameters from src to dest.  */
33239
33240      dest->wn_cylinders = bp_cylinders(src);
33241      dest->wn_heads = bp_heads(src);
33242      dest->wn_reduced_wr = bp_reduced_wr(src);
33243      dest->wn_precomp = bp_precomp(src);
33244      dest->wn_max_ecc = bp_max_ecc(src);
33245      dest->wn_ctlbyte = bp_ctlbyte(src);
33246    }


33249    /*============================================================================*
33250     *                              w_geometry                                    *
33251     *============================================================================*/
33252    PRIVATE void w_geometry(entry)
33253    struct partition *entry;
33254    {
33255      entry->cylinders = w_wn->wn_cylinders;
33256      entry->heads = w_wn->wn_heads;
33257      entry->sectors = NR_SECTORS;
33258    }
33259    #endif /* ENABLE_XT_WINI */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/klib.s
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

33300    #
33301    ! Chooses between the 8086 and 386 versions of the low level kernel code.
33302
33303    #include <minix/config.h>
33304    #if _WORD_SIZE == 2
33305    #include "klib88.s"
33306    #else
33307    #include "klib386.s"
33308    #endif


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/klib386.s
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

33400    #
33401    ! sections
33402
33403    .sect .text; .sect .rom; .sect .data; .sect .bss
33404
33405    #include <minix/config.h>
33406    #include <minix/const.h>
33407    #include "const.h"
33408    #include "sconst.h"
33409    #include "protect.h"
```

```
33410
33411   ! This file contains a number of assembly code utility routines needed by the
33412   ! kernel.  They are:
33413
33414   .define _monitor        ! exit Minix and return to the monitor
33415   .define _check_mem      ! check a block of memory, return the valid size
33416   .define _cp_mess        ! copies messages from source to destination
33417   .define _exit           ! dummy for library routines
33418   .define __exit          ! dummy for library routines
33419   .define ___exit         ! dummy for library routines
33420   .define ___main         ! dummy for GCC
33421   .define _in_byte        ! read a byte from a port and return it
33422   .define _in_word        ! read a word from a port and return it
33423   .define _out_byte       ! write a byte to a port
33424   .define _out_word       ! write a word to a port
33425   .define _port_read      ! transfer data from (disk controller) port to memory
33426   .define _port_read_byte ! likewise byte by byte
33427   .define _port_write     ! transfer data from memory to (disk controller) port
33428   .define _port_write_byte ! likewise byte by byte
33429   .define _lock           ! disable interrupts
33430   .define _unlock         ! enable interrupts
33431   .define _enable_irq     ! enable an irq at the 8259 controller
33432   .define _disable_irq    ! disable an irq
33433   .define _phys_copy      ! copy data from anywhere to anywhere in memory
33434   .define _mem_rdw        ! copy one word from [segment:offset]
33435   .define _reset          ! reset the system
33436   .define _mem_vid_copy   ! copy data to video ram
33437   .define _vid_vid_copy   ! move data in video ram
33438   .define _level0         ! call a function at level 0
33439
33440   ! The routines only guarantee to preserve the registers the C compiler
33441   ! expects to be preserved (ebx, esi, edi, ebp, esp, segment registers, and
33442   ! direction bit in the flags).
33443
33444   ! imported variables
33445
33446   .sect .bss
33447   .extern _mon_return, _mon_sp
33448   .extern _irq_use
33449   .extern _blank_color
33450   .extern _ext_memsize
33451   .extern _gdt
33452   .extern _low_memsize
33453   .extern _sizes
33454   .extern _vid_seg
33455   .extern _vid_size
33456   .extern _vid_mask
33457   .extern _level0_func
33458
33459   .sect .text
33460   !*===========================================================================*
33461   !*                                monitor                                    *
33462   !*===========================================================================*
33463   ! PUBLIC void monitor();
33464   ! Return to the monitor.
33465
33466   _monitor:
33467           mov     eax, (_reboot_code)     ! address of new parameters
33468           mov     esp, (_mon_sp)          ! restore monitor stack pointer
33469       o16 mov     dx, SS_SELECTOR         ! monitor data segment
```

```
33470           mov     ds, dx
33471           mov     es, dx
33472           mov     fs, dx
33473           mov     gs, dx
33474           mov     ss, dx
33475           pop     edi
33476           pop     esi
33477           pop     ebp
33478       o16 retf                        ! return to the monitor
33479
33480
33481   #if ENABLE_BIOS_WINI
33482   !*===========================================================================*
33483   !*                              bios13                                       *
33484   !*===========================================================================*
33485   ! PUBLIC void bios13();
33486   .define _bios13
33487   _bios13:
33488           cmpb    (_mon_return), 0        ! is the monitor there?
33489           jnz     0f
33490           movb    (_Ax+1), 0x01           ! "invalid command"
33491           ret
33492   0:      push    ebp                     ! save C registers
33493           push    esi
33494           push    edi
33495           push    ebx
33496           pushf                           ! save flags
33497           cli                             ! no interruptions
33498
33499           inb     INT2_CTLMASK
33500           movb    ah, al
33501           inb     INT_CTLMASK
33502           push    eax                     ! save interrupt masks
33503           mov     eax, (_irq_use)         ! map of in-use IRQs
33504           and     eax, ~[1<<CLOCK_IRQ]    ! there is a special clock handler
33505           outb    INT_CTLMASK             ! enable all unused IRQs and vv.
33506           movb    al, ah
33507           outb    INT2_CTLMASK
33508
33509           mov     eax, cr0
33510           push    eax                     ! save machine status word
33511
33512           mov     eax, SS_SELECTOR        ! monitor data segment
33513           mov     ss, ax
33514           xchg    esp, (_mon_sp)          ! switch stacks
33515       o16 push    (_Es)                   ! parameters used in bios 13 call
33516       o16 push    (_Dx)
33517       o16 push    (_Cx)
33518       o16 push    (_Bx)
33519       o16 push    (_Ax)
33520           mov     ds, ax                  ! remaining data selectors
33521           mov     es, ax
33522           mov     fs, ax
33523           mov     gs, ax
33524           push    cs
33525           push    return                  ! kernel return address and selector
33526       o16 jmpf    16+2*4+5*2+2*4(esp)     ! make the call
33527   return:
33528       o16 pop     (_Ax)
33529       o16 pop     (_Bx)
```

```
33530         o16 pop     (_Cx)
33531         o16 pop     (_Dx)
33532         o16 pop     (_Es)
33533             lgdt    (_gdt+GDT_SELECTOR)     ! reload global descriptor table
33534             jmpf    CS_SELECTOR:csinit      ! restore everything
33535     csinit: mov     eax, DS_SELECTOR
33536             mov     ds, ax
33537             mov     es, ax
33538             mov     fs, ax
33539             mov     gs, ax
33540             mov     ss, ax
33541             xchg    esp, (_mon_sp)          ! unswitch stacks
33542             lidt    (_gdt+IDT_SELECTOR)     ! reload interrupt descriptor table
33543             andb    (_gdt+TSS_SELECTOR+DESC_ACCESS), ~0x02  ! clear TSS busy bit
33544             mov     ax, TSS_SELECTOR
33545             ltr     ax                      ! set TSS register
33546
33547             pop     eax
33548             mov     cr0, eax                ! restore machine status word
33549
33550             pop     eax
33551             outb    INT_CTLMASK             ! restore interrupt masks
33552             movb    al, ah
33553             outb    INT2_CTLMASK
33554
33555             popf                            ! restore flags
33556             pop     ebx                     ! restore C registers
33557             pop     edi
33558             pop     esi
33559             pop     ebp
33560             ret
33561
33562     .sect .bss
33563     .define _Ax, _Bx, _Cx, _Dx, _Es         ! 8086 register variables
33564     .comm   _Ax, 4
33565     .comm   _Bx, 4
33566     .comm   _Cx, 4
33567     .comm   _Dx, 4
33568     .comm   _Es, 4
33569     .sect .text
33570     #endif /* ENABLE_BIOS_WINI */
33571
33572
33573     !*===========================================================================*
33574     !*                                check_mem                                  *
33575     !*===========================================================================*
33576     ! PUBLIC phys_bytes check_mem(phys_bytes base, phys_bytes size);
33577     ! Check a block of memory, return the amount valid.
33578     ! Only every 16th byte is checked.
33579     ! An initial size of 0 means everything.
33580     ! This really should do some alias checks.
33581
33582     CM_DENSITY      =       16
33583     CM_LOG_DENSITY  =       4
33584     TEST1PATTERN    =       0x55            ! memory test pattern 1
33585     TEST2PATTERN    =       0xAA            ! memory test pattern 2
33586
33587     CHKM_ARGS       =       4 + 4 + 4       ! 4 + 4
33588     !                       ds ebx eip      base size
33589
```

```
33590   _check_mem:
33591           push    ebx
33592           push    ds
33593       o16 mov     ax, FLAT_DS_SELECTOR
33594           mov     ds, ax
33595           mov     eax, CHKM_ARGS(esp)
33596           mov     ebx, eax
33597           mov     ecx, CHKM_ARGS+4(esp)
33598           shr     ecx, CM_LOG_DENSITY
33599   cm_loop:
33600           movb    dl, TEST1PATTERN
33601           xchgb   dl, (eax)               ! write test pattern, remember original
33602           xchgb   dl, (eax)               ! restore original, read test pattern
33603           cmpb    dl, TEST1PATTERN        ! must agree if good real memory
33604           jnz     cm_exit                 ! if different, memory is unusable
33605           movb    dl, TEST2PATTERN
33606           xchgb   dl, (eax)
33607           xchgb   dl, (eax)
33608           add     eax, CM_DENSITY
33609           cmpb    dl, TEST2PATTERN
33610           loopz   cm_loop
33611   cm_exit:
33612           sub     eax, ebx
33613           pop     ds
33614           pop     ebx
33615           ret
33616
33617
33618   !*===========================================================================*
33619   !*                              cp_mess                                      *
33620   !*===========================================================================*
33621   ! PUBLIC void cp_mess(int src, phys_clicks src_clicks, vir_bytes src_offset,
33622   !                     phys_clicks dst_clicks, vir_bytes dst_offset);
33623   ! This routine makes a fast copy of a message from anywhere in the address
33624   ! space to anywhere else.  It also copies the source address provided as a
33625   ! parameter to the call into the first word of the destination message.
33626   !
33627   ! Note that the message size, "Msize" is in DWORDS (not bytes) and must be set
33628   ! correctly.  Changing the definition of message in the type file and not
33629   ! changing it here will lead to total disaster.
33630
33631   CM_ARGS =       4 + 4 + 4 + 4 + 4       ! 4 + 4 + 4 + 4 + 4
33632   !               es  ds edi esi eip      proc scl sof dcl dof
33633
33634           .align  16
33635   _cp_mess:
33636           cld
33637           push    esi
33638           push    edi
33639           push    ds
33640           push    es
33641
33642           mov     eax, FLAT_DS_SELECTOR
33643           mov     ds, ax
33644           mov     es, ax
33645
33646           mov     esi, CM_ARGS+4(esp)             ! src clicks
33647           shl     esi, CLICK_SHIFT
33648           add     esi, CM_ARGS+4+4(esp)           ! src offset
33649           mov     edi, CM_ARGS+4+4+4(esp)         ! dst clicks
```

```
33650           shl     edi, CLICK_SHIFT
33651           add     edi, CM_ARGS+4+4+4+4(esp)       ! dst offset
33652
33653           mov     eax, CM_ARGS(esp)       ! process number of sender
33654           stos                            ! copy number of sender to dest message
33655           add     esi, 4                  ! do not copy first word
33656           mov     ecx, Msize - 1          ! remember, first word does not count
33657           rep
33658           movs                            ! copy the message
33659
33660           pop     es
33661           pop     ds
33662           pop     edi
33663           pop     esi
33664           ret                             ! that is all folks!
33665
33666
33667   !*===========================================================================*
33668   !*                              exit                                         *
33669   !*===========================================================================*
33670   ! PUBLIC void exit();
33671   ! Some library routines use exit, so provide a dummy version.
33672   ! Actual calls to exit cannot occur in the kernel.
33673   ! GNU CC likes to call ___main from main() for nonobvious reasons.
33674
33675   _exit:
33676   __exit:
33677   ___exit:
33678           sti
33679           jmp     ___exit
33680
33681   ___main:
33682           ret
33683
33684
33685   !*===========================================================================*
33686   !*                              in_byte                                      *
33687   !*===========================================================================*
33688   ! PUBLIC unsigned in_byte(port_t port);
33689   ! Read an (unsigned) byte from the i/o port  port  and return it.
33690
33691           .align  16
33692   _in_byte:
33693           mov     edx, 4(esp)             ! port
33694           sub     eax, eax
33695           inb     dx                      ! read 1 byte
33696           ret
33697
33698
33699   !*===========================================================================*
33700   !*                              in_word                                      *
33701   !*===========================================================================*
33702   ! PUBLIC unsigned in_word(port_t port);
33703   ! Read an (unsigned) word from the i/o port  port  and return it.
33704
33705           .align  16
33706   _in_word:
33707           mov     edx, 4(esp)             ! port
33708           sub     eax, eax
33709        o16 in     dx                      ! read 1 word
```

```
33710           ret
33711
33712
33713   !*===========================================================================*
33714   !*                              out_byte                                     *
33715   !*===========================================================================*
33716   ! PUBLIC void out_byte(port_t port, u8_t value);
33717   ! Write  value  (cast to a byte)  to the I/O port  port.
33718
33719           .align  16
33720   _out_byte:
33721           mov     edx, 4(esp)             ! port
33722           movb    al, 4+4(esp)            ! value
33723           outb    dx                      ! output 1 byte
33724           ret
33725
33726
33727   !*===========================================================================*
33728   !*                              out_word                                     *
33729   !*===========================================================================*
33730   ! PUBLIC void out_word(Port_t port, U16_t value);
33731   ! Write  value  (cast to a word)  to the I/O port  port.
33732
33733           .align  16
33734   _out_word:
33735           mov     edx, 4(esp)             ! port
33736           mov     eax, 4+4(esp)           ! value
33737       o16 out     dx                      ! output 1 word
33738           ret
33739
33740
33741   !*===========================================================================*
33742   !*                              port_read                                    *
33743   !*===========================================================================*
33744   ! PUBLIC void port_read(port_t port, phys_bytes destination, unsigned bytcount);
33745   ! Transfer data from (hard disk controller) port to memory.
33746
33747   PR_ARGS =       4 + 4 + 4               ! 4 + 4 + 4
33748   !               es edi eip              port dst len
33749
33750           .align  16
33751   _port_read:
33752           cld
33753           push    edi
33754           push    es
33755           mov     ecx, FLAT_DS_SELECTOR
33756           mov     es, cx
33757           mov     edx, PR_ARGS(esp)       ! port to read from
33758           mov     edi, PR_ARGS+4(esp)     ! destination addr
33759           mov     ecx, PR_ARGS+4+4(esp)   ! byte count
33760           shr     ecx, 1                  ! word count
33761           rep                             ! (hardware cannot handle dwords)
33762       o16 ins                             ! read everything
33763           pop     es
33764           pop     edi
33765           ret
33766
33767
33768   !*===========================================================================*
33769   !*                              port_read_byte                               *
```

```
33770   !*===========================================================================*
33771   ! PUBLIC void port_read_byte(port_t port, phys_bytes destination,
33772   !                                               unsigned bytcount);
33773   ! Transfer data from port to memory.
33774
33775   PR_ARGS_B =     4 + 4 + 4               ! 4 + 4 + 4
33776   !               es edi eip              port dst len
33777
33778   _port_read_byte:
33779           cld
33780           push    edi
33781           push    es
33782           mov     ecx, FLAT_DS_SELECTOR
33783           mov     es, cx
33784           mov     edx, PR_ARGS_B(esp)
33785           mov     edi, PR_ARGS_B+4(esp)
33786           mov     ecx, PR_ARGS_B+4+4(esp)
33787           rep
33788           insb
33789           pop     es
33790           pop     edi
33791           ret
33792
33793
33794   !*===========================================================================*
33795   !*                              port_write                                   *
33796   !*===========================================================================*
33797   ! PUBLIC void port_write(port_t port, phys_bytes source, unsigned bytcount);
33798   ! Transfer data from memory to (hard disk controller) port.
33799
33800   PW_ARGS =       4 + 4 + 4               ! 4 + 4 + 4
33801   !               es edi eip              port src len
33802
33803           .align  16
33804   _port_write:
33805           cld
33806           push    esi
33807           push    ds
33808           mov     ecx, FLAT_DS_SELECTOR
33809           mov     ds, cx
33810           mov     edx, PW_ARGS(esp)       ! port to write to
33811           mov     esi, PW_ARGS+4(esp)     ! source addr
33812           mov     ecx, PW_ARGS+4+4(esp)   ! byte count
33813           shr     ecx, 1                  ! word count
33814           rep                             ! (hardware cannot handle dwords)
33815       o16 outs                            ! write everything
33816           pop     ds
33817           pop     esi
33818           ret
33819
33820
33821   !*===========================================================================*
33822   !*                              port_write_byte                              *
33823   !*===========================================================================*
33824   ! PUBLIC void port_write_byte(port_t port, phys_bytes source,
33825   !                                               unsigned bytcount);
33826   ! Transfer data from memory to port.
33827
33828   PW_ARGS_B =     4 + 4 + 4               ! 4 + 4 + 4
33829   !               es edi eip              port src len
```

```
33830
33831   _port_write_byte:
33832           cld
33833           push    esi
33834           push    ds
33835           mov     ecx, FLAT_DS_SELECTOR
33836           mov     ds, cx
33837           mov     edx, PW_ARGS_B(esp)
33838           mov     esi, PW_ARGS_B+4(esp)
33839           mov     ecx, PW_ARGS_B+4+4(esp)
33840           rep
33841           outsb
33842           pop     ds
33843           pop     esi
33844           ret
33845
33846
33847   !*===========================================================================*
33848   !*                              lock                                         *
33849   !*===========================================================================*
33850   ! PUBLIC void lock();
33851   ! Disable CPU interrupts.
33852
33853           .align  16
33854   _lock:
33855           cli                             ! disable interrupts
33856           ret
33857
33858
33859   !*===========================================================================*
33860   !*                              unlock                                       *
33861   !*===========================================================================*
33862   ! PUBLIC void unlock();
33863   ! Enable CPU interrupts.
33864
33865           .align  16
33866   _unlock:
33867           sti
33868           ret
33869
33870
33871   !*===========================================================================*
33872   !*                              enable_irq                                   *
33873   !*===========================================================================*/
33874   ! PUBLIC void enable_irq(unsigned irq)
33875   ! Enable an interrupt request line by clearing an 8259 bit.
33876   ! Equivalent code for irq < 8:
33877   !       out_byte(INT_CTLMASK, in_byte(INT_CTLMASK) & ~(1 << irq));
33878
33879           .align  16
33880   _enable_irq:
33881           mov     ecx, 4(esp)             ! irq
33882           pushf
33883           cli
33884           movb    ah, ~1
33885           rolb    ah, cl                  ! ah = ~(1 << (irq % 8))
33886           cmpb    cl, 8
33887           jae     enable_8                ! enable irq >= 8 at the slave 8259
33888   enable_0:
33889           inb     INT_CTLMASK
```

```
33890           andb    al, ah
33891           outb    INT_CTLMASK             ! clear bit at master 8259
33892           popf
33893           ret
33894           .align  4
33895   enable_8:
33896           inb     INT2_CTLMASK
33897           andb    al, ah
33898           outb    INT2_CTLMASK            ! clear bit at slave 8259
33899           popf
33900           ret
33901
33902
33903   !*===========================================================================*
33904   !*                              disable_irq                                  *
33905   !*===========================================================================*/
33906   ! PUBLIC int disable_irq(unsigned irq)
33907   ! Disable an interrupt request line by setting an 8259 bit.
33908   ! Equivalent code for irq < 8:
33909   !       out_byte(INT_CTLMASK, in_byte(INT_CTLMASK) | (1 << irq));
33910   ! Returns true iff the interrupt was not already disabled.
33911
33912           .align  16
33913   _disable_irq:
33914           mov     ecx, 4(esp)             ! irq
33915           pushf
33916           cli
33917           movb    ah, 1
33918           rolb    ah, cl                  ! ah = (1 << (irq % 8))
33919           cmpb    cl, 8
33920           jae     disable_8               ! disable irq >= 8 at the slave 8259
33921   disable_0:
33922           inb     INT_CTLMASK
33923           testb   al, ah
33924           jnz     dis_already             ! already disabled?
33925           orb     al, ah
33926           outb    INT_CTLMASK             ! set bit at master 8259
33927           popf
33928           mov     eax, 1                  ! disabled by this function
33929           ret
33930   disable_8:
33931           inb     INT2_CTLMASK
33932           testb   al, ah
33933           jnz     dis_already             ! already disabled?
33934           orb     al, ah
33935           outb    INT2_CTLMASK            ! set bit at slave 8259
33936           popf
33937           mov     eax, 1                  ! disabled by this function
33938           ret
33939   dis_already:
33940           popf
33941           xor     eax, eax                ! already disabled
33942           ret
33943
33944
33945   !*===========================================================================*
33946   !*                              phys_copy                                    *
33947   !*===========================================================================*
33948   ! PUBLIC void phys_copy(phys_bytes source, phys_bytes destination,
33949   !                       phys_bytes bytecount);
```

```
33950   ! Copy a block of physical memory.
33951
33952   PC_ARGS =       4 + 4 + 4 + 4   ! 4 + 4 + 4
33953   !               es edi esi eip   src dst len
33954
33955           .align  16
33956   _phys_copy:
33957           cld
33958           push    esi
33959           push    edi
33960           push    es
33961
33962           mov     eax, FLAT_DS_SELECTOR
33963           mov     es, ax
33964
33965           mov     esi, PC_ARGS(esp)
33966           mov     edi, PC_ARGS+4(esp)
33967           mov     eax, PC_ARGS+4+4(esp)
33968
33969           cmp     eax, 10                 ! avoid align overhead for small counts
33970           jb      pc_small
33971           mov     ecx, esi                ! align source, hope target is too
33972           neg     ecx
33973           and     ecx, 3                  ! count for alignment
33974           sub     eax, ecx
33975           rep
33976      eseg movsb
33977           mov     ecx, eax
33978           shr     ecx, 2                  ! count of dwords
33979           rep
33980      eseg movs
33981           and     eax, 3
33982   pc_small:
33983           xchg    ecx, eax                ! remainder
33984           rep
33985      eseg movsb
33986
33987           pop     es
33988           pop     edi
33989           pop     esi
33990           ret
33991
33992
33993   !*===========================================================================*
33994   !*                                mem_rdw                                    *
33995   !*===========================================================================*
33996   ! PUBLIC u16_t mem_rdw(U16_t segment, u16_t *offset);
33997   ! Load and return word at far pointer segment:offset.
33998
33999           .align  16
34000   _mem_rdw:
34001           mov     cx, ds
34002           mov     ds, 4(esp)              ! segment
34003           mov     eax, 4+4(esp)           ! offset
34004           movzx   eax, (eax)              ! word to return
34005           mov     ds, cx
34006           ret
34007
34008
34009   !*===========================================================================*
```

```
34010   !*                              reset                                  *
34011   !*=====================================================================*
34012   ! PUBLIC void reset();
34013   ! Reset the system by loading IDT with offset 0 and interrupting.
34014
34015   _reset:
34016           lidt    (idt_zero)
34017           int     3               ! anything goes, the 386 will not like it
34018   .sect .data
34019   idt_zero:       .data4  0, 0
34020   .sect .text
34021
34022
34023   !*=====================================================================*
34024   !*                              mem_vid_copy                           *
34025   !*=====================================================================*
34026   ! PUBLIC void mem_vid_copy(u16 *src, unsigned dst, unsigned count);
34027   !
34028   ! Copy count characters from kernel memory to video memory.  Src, dst and
34029   ! count are character (word) based video offsets and counts.  If src is null
34030   ! then screen memory is blanked by filling it with blank_color.
34031
34032   MVC_ARGS        =       4 + 4 + 4 + 4   ! 4 + 4 + 4
34033   !                       es edi esi eip   src dst ct
34034
34035   _mem_vid_copy:
34036           push    esi
34037           push    edi
34038           push    es
34039           mov     esi, MVC_ARGS(esp)      ! source
34040           mov     edi, MVC_ARGS+4(esp)    ! destination
34041           mov     edx, MVC_ARGS+4+4(esp)  ! count
34042           mov     es, (_vid_seg)          ! destination is video segment
34043           cld                             ! make sure direction is up
34044   mvc_loop:
34045           and     edi, (_vid_mask)        ! wrap address
34046           mov     ecx, edx                ! one chunk to copy
34047           mov     eax, (_vid_size)
34048           sub     eax, edi
34049           cmp     ecx, eax
34050           jbe     0f
34051           mov     ecx, eax                ! ecx = min(ecx, vid_size - edi)
34052   0:      sub     edx, ecx                ! count -= ecx
34053           shl     edi, 1                  ! byte address
34054           test    esi, esi                ! source == 0 means blank the screen
34055           jz      mvc_blank
34056   mvc_copy:
34057           rep                             ! copy words to video memory
34058       o16 movs
34059           jmp     mvc_test
34060   mvc_blank:
34061           mov     eax, (_blank_color)     ! ax = blanking character
34062           rep
34063       o16 stos                            ! copy blanks to video memory
34064           !jmp    mvc_test
34065   mvc_test:
34066           shr     edi, 1                  ! word addresses
34067           test    edx, edx
34068           jnz     mvc_loop
34069   mvc_done:
```

```
34070           pop     es
34071           pop     edi
34072           pop     esi
34073           ret
34074
34075
34076   !*===========================================================================*
34077   !*                              vid_vid_copy                                 *
34078   !*===========================================================================*
34079   ! PUBLIC void vid_vid_copy(unsigned src, unsigned dst, unsigned count);
34080   !
34081   ! Copy count characters from video memory to video memory.  Handle overlap.
34082   ! Used for scrolling, line or character insertion and deletion.  Src, dst
34083   ! and count are character (word) based video offsets and counts.
34084
34085   VVC_ARGS        =       4 + 4 + 4 + 4   ! 4 + 4 + 4
34086   !                       es edi esi eip   src dst ct
34087
34088   _vid_vid_copy:
34089           push    esi
34090           push    edi
34091           push    es
34092           mov     esi, VVC_ARGS(esp)      ! source
34093           mov     edi, VVC_ARGS+4(esp)    ! destination
34094           mov     edx, VVC_ARGS+4+4(esp)  ! count
34095           mov     es, (_vid_seg)          ! use video segment
34096           cmp     esi, edi                ! copy up or down?
34097           jb      vvc_down
34098   vvc_up:
34099           cld                             ! direction is up
34100   vvc_uploop:
34101           and     esi, (_vid_mask)        ! wrap addresses
34102           and     edi, (_vid_mask)
34103           mov     ecx, edx                ! one chunk to copy
34104           mov     eax, (_vid_size)
34105           sub     eax, esi
34106           cmp     ecx, eax
34107           jbe     0f
34108           mov     ecx, eax                ! ecx = min(ecx, vid_size - esi)
34109   0:      mov     eax, (_vid_size)
34110           sub     eax, edi
34111           cmp     ecx, eax
34112           jbe     0f
34113           mov     ecx, eax                ! ecx = min(ecx, vid_size - edi)
34114   0:      sub     edx, ecx                ! count -= ecx
34115           shl     esi, 1
34116           shl     edi, 1                  ! byte addresses
34117           rep
34118   eseg o16 movs                           ! copy video words
34119           shr     esi, 1
34120           shr     edi, 1                  ! word addresses
34121           test    edx, edx
34122           jnz     vvc_uploop              ! again?
34123           jmp     vvc_done
34124   vvc_down:
34125           std                             ! direction is down
34126           lea     esi, -1(esi)(edx*1)     ! start copying at the top
34127           lea     edi, -1(edi)(edx*1)
34128   vvc_downloop:
34129           and     esi, (_vid_mask)        ! wrap addresses
```

```
34130           and     edi, (_vid_mask)
34131           mov     ecx, edx                ! one chunk to copy
34132           lea     eax, 1(esi)
34133           cmp     ecx, eax
34134           jbe     0f
34135           mov     ecx, eax                ! ecx = min(ecx, esi + 1)
34136   0:      lea     eax, 1(edi)
34137           cmp     ecx, eax
34138           jbe     0f
34139           mov     ecx, eax                ! ecx = min(ecx, edi + 1)
34140   0:      sub     edx, ecx                ! count -= ecx
34141           shl     esi, 1
34142           shl     edi, 1                  ! byte addresses
34143           rep
34144   eseg o16 movs                           ! copy video words
34145           shr     esi, 1
34146           shr     edi, 1                  ! word addresses
34147           test    edx, edx
34148           jnz     vvc_downloop            ! again?
34149           cld                             ! C compiler expect up
34150           !jmp    vvc_done
34151   vvc_done:
34152           pop     es
34153           pop     edi
34154           pop     esi
34155           ret
34156
34157
34158   !*===========================================================================*
34159   !*                              level0                                       *
34160   !*===========================================================================*
34161   ! PUBLIC void level0(void (*func)(void))
34162   ! Call a function at permission level 0.  This allows kernel tasks to do
34163   ! things that are only possible at the most privileged CPU level.
34164   !
34165   _level0:
34166           mov     eax, 4(esp)
34167           mov     (_level0_func), eax
34168           int     LEVEL0_VECTOR
34169           ret
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/klib88.s
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

34200   #
34201   #include <minix/config.h>
34202   #include <minix/const.h>
34203   #include "const.h"
34204   #include "sconst.h"
34205   #include "protect.h"
34206
34207   ! This file contains a number of assembly code utility routines needed by the
34208   ! kernel.  They are:
34209
```

```
34210   .define _monitor        ! exit Minix and return to the monitor
34211   .define real2prot       ! switch from real to protected mode
34212   .define prot2real       ! switch from protected to real mode
34213   .define _check_mem      ! check a block of memory, return the valid size
34214   .define _cp_mess        ! copies messages from source to destination
34215   .define _exit           ! dummy for library routines
34216   .define __exit          ! dummy for library routines
34217   .define ___exit         ! dummy for library routines
34218   .define .fat, .trp      ! dummies for library routines
34219   .define _in_byte        ! read a byte from a port and return it
34220   .define _in_word        ! read a word from a port and return it
34221   .define _out_byte       ! write a byte to a port
34222   .define _out_word       ! write a word to a port
34223   .define _port_read      ! transfer data from (disk controller) port to memory
34224   .define _port_read_byte ! likewise byte by byte
34225   .define _port_write     ! transfer data from memory to (disk controller) port
34226   .define _port_write_byte ! likewise byte by byte
34227   .define _lock           ! disable interrupts
34228   .define _unlock         ! enable interrupts
34229   .define _enable_irq     ! enable an irq at the 8259 controller
34230   .define _disable_irq    ! disable an irq
34231   .define _phys_copy      ! copy data from anywhere to anywhere in memory
34232   .define _mem_rdw        ! copy one word from [segment:offset]
34233   .define _reset          ! reset the system
34234   .define _mem_vid_copy   ! copy data to video ram
34235   .define _vid_vid_copy   ! move data in video ram
34236   .define _level0         ! call a function at level 0
34237   .define klib_init_prot  ! initialize klib functions for protected mode
34238
34239   ! The routines only guarantee to preserve the registers the C compiler
34240   ! expects to be preserved (si, di, bp, sp, segment registers, and direction
34241   ! bit in the flags), though some of the older ones preserve bx, cx and dx.
34242
34243   #define DS_286_OFFSET   DS_286_INDEX*DESC_SIZE
34244   #define ES_286_OFFSET   ES_286_INDEX*DESC_SIZE
34245   #       define EM_XFER_FUNC     0x87
34246   #define JMP_OPCODE      0xE9    /* opcode used for patching */
34247   #define OFF_MASK        0x000F  /* offset mask for phys_b -> hclick:offset */
34248   #define HCHIGH_MASK     0x0F    /* h/w click mask for low byte of hi word */
34249   #define HCLOW_MASK      0xF0    /* h/w click mask for low byte of low word */
34250
34251   ! Imported functions
34252
34253   .extern p_restart
34254   .extern p_save
34255   .extern _restart
34256   .extern save
34257
34258   ! Exported variables
34259
34260   .extern kernel_cs
34261
34262   ! Imported variables
34263
34264   .extern kernel_ds
34265   .extern _irq_use
34266   .extern _blank_color
34267   .extern _gdt
34268   .extern _protected_mode
34269   .extern _vid_seg
```

```
34270   .extern _vid_size
34271   .extern _vid_mask
34272   .extern _level0_func
34273
34274           .text
34275   !*===========================================================================*
34276   !*                              monitor                                      *
34277   !*===========================================================================*
34278   ! PUBLIC void monitor();
34279   ! Return to the monitor.
34280
34281   _monitor:
34282           call    prot2real               ! switch to real mode
34283           mov     ax, _reboot_code+0      ! address of new parameters
34284           mov     dx, _reboot_code+2
34285           mov     sp, _mon_sp             ! restore monitor stack pointer
34286           mov     bx, _mon_ss             ! monitor data segment
34287           mov     ds, bx
34288           mov     es, bx
34289           mov     ss, bx
34290           pop     di
34291           pop     si
34292           pop     bp
34293           retf                            ! return to the monitor
34294
34295
34296   #if ENABLE_BIOS_WINI
34297   !*===========================================================================*
34298   !*                              bios13                                       *
34299   !*===========================================================================*
34300   ! PUBLIC void bios13();
34301   .define _bios13
34302   _bios13:                        ! make a BIOS 0x13 call for disk I/O
34303           push    si
34304           push    di              ! save C variable registers
34305           pushf                   ! save flags
34306
34307           call    int13           ! make the actual call
34308
34309           popf                    ! restore flags
34310           pop     di              ! restore C registers
34311           pop     si
34312           ret
34313
34314   ! Make a BIOS 0x13 call from protected mode
34315   p_bios13:
34316           push    bp
34317           push    si
34318           push    di                      ! save C variable registers
34319           pushf                           ! save flags
34320           cli                             ! no interruptions
34321           inb     INT2_CTLMASK
34322           movb    ah, al
34323           inb     INT_CTLMASK
34324           push    ax                      ! save interrupt masks
34325           mov     ax, _irq_use            ! map of in-use IRQs
34326           and     ax, #~[1<<CLOCK_IRQ]    ! there is a special clock handler
34327           outb    INT_CTLMASK             ! enable all unused IRQs and vv.
34328           movb    al, ah
34329           outb    INT2_CTLMASK
```

```
34330
34331           smsw    ax
34332           push    ax                      ! save machine status word
34333           call    prot2real               ! switch to real mode
34334
34335           call    int13                   ! make the actual call
34336
34337           call    real2prot               ! back to protected mode
34338           pop     ax
34339           lmsw    ax                      ! restore msw
34340
34341           pop     ax                      ! restore interrupt masks
34342           outb    INT_CTLMASK
34343           movb    al, ah
34344           outb    INT2_CTLMASK
34345           popf                            ! restore flags
34346           pop     di
34347           pop     si
34348           pop     bp                      ! restore C variable registers
34349           ret
34350
34351   int13:
34352           mov     ax, _Ax         ! load parameters
34353           mov     bx, _Bx
34354           mov     cx, _Cx
34355           mov     dx, _Dx
34356           mov     es, _Es
34357           sti                     ! enable interrupts
34358           int     0x13            ! make the BIOS call
34359           cli                     ! disable interrupts
34360           mov     _Ax, ax         ! save results
34361           mov     _Bx, bx
34362           mov     _Cx, cx
34363           mov     _Dx, dx
34364           mov     _Es, es
34365           mov     ax, ds
34366           mov     es, ax          ! restore es
34367           ret
34368
34369   .bss
34370   .define _Ax, _Bx, _Cx, _Dx, _Es         ! 8086 register variables
34371   .comm   _Ax, 2
34372   .comm   _Bx, 2
34373   .comm   _Cx, 2
34374   .comm   _Dx, 2
34375   .comm   _Es, 2
34376   .text
34377   #endif /* ENABLE_BIOS_WINI */
34378
34379
34380   !*===========================================================================*
34381   !*                              real2prot                                    *
34382   !*===========================================================================*
34383   ! Switch from real to protected mode.
34384   real2prot:
34385           lgdt    _gdt+GDT_SELECTOR       ! set global descriptor table
34386           smsw    ax
34387           mov     msw, ax                 ! save real mode msw
34388           orb     al, #0x01               ! set PE (protection enable) bit
34389           lmsw    ax                      ! set msw, enabling protected mode
```

```
34390
34391           jmpf    csinit, CS_SELECTOR     ! set code segment selector
34392   csinit:
34393           mov     ax, #DS_SELECTOR        ! set data selectors
34394           mov     ds, ax
34395           mov     es, ax
34396           mov     ss, ax
34397           lidt    _gdt+IDT_SELECTOR       ! set interrupt vectors
34398           andb    _gdt+TSS_SELECTOR+DESC_ACCESS, #~0x02  ! clear TSS busy bit
34399           mov     ax, #TSS_SELECTOR
34400           ltr     ax                      ! set TSS register
34401
34402           movb    ah, #0xDF
34403           jmp     gate_A20                ! enable the A20 address line
34404
34405
34406   !*===========================================================================*
34407   !*                              prot2real                                    *
34408   !*===========================================================================*
34409   ! Switch from protected to real mode.
34410   prot2real:
34411           mov     save_sp, sp             ! save stack pointer
34412           cmp     _processor, #386        ! is this a 386?
34413           jae     p2r386
34414   p2r286:
34415           mov     _gdt+ES_286_OFFSET+DESC_BASE, #0x0400
34416           movb    _gdt+ES_286_OFFSET+DESC_BASE_MIDDLE, #0x00
34417           mov     ax, #ES_286_SELECTOR
34418           mov     es, ax                  ! BIOS data segment
34419     eseg  mov     0x0067, #real           ! set return from shutdown address
34420     cseg  mov     ax, kernel_cs
34421     eseg  mov     0x0069, ax
34422           movb    al, #0x8F
34423           outb    0x70                    ! select CMOS byte 0x0F (disable NMI)
34424           jmp     .+2
34425           movb    al, #0x0A
34426           outb    0x71                    ! set shutdown code to 0x0A "jump far"
34427           jmp     p_reset                 ! cause a processor shutdown
34428   p2r386:
34429           lidt    idt_vectors             ! real mode interrupt vectors
34430           push    _gdt+CS_SELECTOR+0
34431           push    _gdt+DS_SELECTOR+0      ! save CS and DS limits
34432           mov     _gdt+CS_SELECTOR+0, #0xFFFF
34433           mov     _gdt+DS_SELECTOR+0, #0xFFFF ! set 64k limits
34434           jmpf    cs64k, CS_SELECTOR      ! reload selectors
34435   cs64k:  mov     ax, #DS_SELECTOR
34436           mov     ds, ax
34437           mov     es, ax
34438           mov     ss, ax
34439           pop     _gdt+DS_SELECTOR+0
34440           pop     _gdt+CS_SELECTOR+0      ! restore CS and DS limits
34441           .data1  0x0F,0x20,0xC0          ! mov   eax, cr0
34442           mov     ax, msw                 ! restore real mode (16 bits) msw
34443           .data1  0x0F,0x22,0xC0          ! mov   cr0, eax
34444           .data1  0xEA                    ! jmpf real, "kernel_cs"
34445           .data2  real
34446   kernel_cs:
34447           .data2  0
34448   real:
34449     cseg  mov     ax, kernel_ds           ! reload data segment registers
```

```
34450           mov     ds, ax
34451           mov     es, ax
34452           mov     ss, ax
34453           mov     sp, save_sp             ! restore stack
34454
34455           movb    ah, #0xDD
34456           !jmp    gate_A20                ! disable the A20 address line
34457
34458   ! Enable (ah = 0xDF) or disable (ah = 0xDD) the A20 address line.
34459   gate_A20:
34460           call    kb_wait
34461           movb    al, #0xD1       ! Tell keyboard that a command is coming
34462           outb    0x64
34463           call    kb_wait
34464           movb    al, ah          ! Enable or disable code
34465           outb    0x60
34466           call    kb_wait
34467           mov     ax, #25         ! 25 microsec delay for slow keyboard chip
34468   0:      out     0xED            ! Write to an unused port (lus)
34469           dec     ax
34470           jne     0b
34471           ret
34472   kb_wait:
34473           inb     0x64
34474           testb   al, #0x02       ! Keyboard input buffer full?
34475           jnz     kb_wait         ! If so, wait
34476           ret
34477
34478
34479   !*===========================================================================*
34480   !*                              check_mem                                    *
34481   !*===========================================================================*
34482   ! PUBLIC phys_bytes check_mem(phys_bytes base, phys_bytes size);
34483   ! Check a block of memory, return the amount valid.
34484   ! Only every 16th byte is checked.
34485   ! This only works in protected mode.
34486   ! An initial size of 0 means everything.
34487   ! This really should do some alias checks.
34488
34489   PCM_DENSITY     =       256     ! resolution of check
34490                                   ! the shift logic depends on this being 256
34491   TEST1PATTERN    =       0x55    ! memory test pattern 1
34492   TEST2PATTERN    =       0xAA    ! memory test pattern 2
34493
34494   _check_mem:
34495           pop     bx
34496           pop     _gdt+DS_286_OFFSET+DESC_BASE
34497           pop     ax              ! pop base into base of source descriptor
34498           movb    _gdt+DS_286_OFFSET+DESC_BASE_MIDDLE,al
34499           pop     cx              ! byte count in dx:cx
34500           pop     dx
34501           sub     sp,#4+4
34502           push    bx
34503           push    ds
34504
34505           sub     ax,ax           ! prepare for early exit
34506           test    dx,#0xFF00
34507           jnz     cm_lexit        ! cannot handle bases above 16M
34508           movb    cl,ch           ! divide size by 256 and discard high byte
34509           movb    ch,dl
```

```
34510           push    cx              ! save divided size
34511           sub     bx,bx           ! test bytes at bases of segments
34512   cm_loop:
34513           mov     ax,#DS_286_SELECTOR
34514           mov     ds,ax
34515           movb    dl,#TEST1PATTERN
34516           xchgb   dl,(bx)         ! write test pattern, remember original value
34517           xchgb   dl,(bx)         ! restore original value, read test pattern
34518           cmpb    dl,#TEST1PATTERN        ! must agree if good real memory
34519           jnz     cm_exit         ! if different, memory is unusable
34520           movb    dl,#TEST2PATTERN
34521           xchgb   dl,(bx)
34522           xchgb   dl,(bx)
34523           cmpb    dl,#TEST2PATTERN
34524           jnz     cm_exit
34525                                   ! next segment, test for wraparound at 16M
34526                                   ! assuming es == old ds
34527     eseg  add     _gdt+DS_286_OFFSET+DESC_BASE,#PCM_DENSITY
34528     eseg  adcb    _gdt+DS_286_OFFSET+DESC_BASE_MIDDLE,#0
34529           loopnz  cm_loop
34530
34531   cm_exit:
34532           pop     ax
34533           sub     ax,cx           ! verified size in multiples of PCM_DENSITY
34534   cm_1exit:
34535           movb    dl,ah           ! convert to phys_bytes in dx:ax
34536           subb    dh,dh
34537           movb    ah,al
34538           movb    al,dh
34539           pop     ds
34540           ret
34541
34542
34543   !*===========================================================================*
34544   !*                              cp_mess                                      *
34545   !*===========================================================================*
34546   ! PUBLIC void cp_mess(int src, phys_clicks src_clicks, vir_bytes src_offset,
34547   !                     phys_clicks dst_clicks, vir_bytes dst_offset);
34548   ! This routine makes a fast copy of a message from anywhere in the address
34549   ! space to anywhere else.  It also copies the source address provided as a
34550   ! parameter to the call into the first word of the destination message.
34551   !
34552   ! Note that the message size, "Msize" is in WORDS (not bytes) and must be set
34553   ! correctly.  Changing the definition of message in the type file and not
34554   ! changing it here will lead to total disaster.
34555
34556   _cp_mess:
34557           cld
34558           push es                 ! save es
34559           push ds                 ! save ds
34560           mov bx,sp               ! index off bx because machine cannot use sp
34561           push si                 ! save si
34562           push di                 ! save di
34563
34564           mov     ax,12(bx)       ! destination click
34565   #if HCLICK_SHIFT > CLICK_SHIFT
34566   #error /* Small click sizes are not supported (right shift will lose bits). */
34567   #endif
34568   #if HCLICK_SHIFT < CLICK_SHIFT
34569           movb    cl,#CLICK_SHIFT-HCLICK_SHIFT
```

```
34570           shl     ax,cl           ! destination segment
34571   #endif
34572           mov     es,ax
34573           mov     di,14(bx)       ! offset of destination message
34574
34575   ! Be careful not to destroy ds before we are finished with the bx pointer.
34576   ! We are using bx and not the more natural bp to save pushing bp.
34577
34578           mov     ax,6(bx)        ! process number of sender
34579           mov     si,10(bx)       ! offset of source message
34580           mov     bx,8(bx)        ! source click (finished with bx as a pointer)
34581   #if HCLICK_SHIFT < CLICK_SHIFT
34582           shl     bx,cl           ! source segment
34583   #endif
34584           mov     ds,bx
34585
34586           stos                    ! copy process number of sender to dest message
34587           add si,*2               ! do not copy first word
34588           mov cx,*Msize-1         ! remember, first word does not count
34589           rep                     ! iterate cx times to copy 11 words
34590           movs                    ! copy the message
34591           pop di                  ! restore di
34592           pop si                  ! restore si
34593           pop ds                  ! restore ds
34594           pop es                  ! restore es
34595           ret                     ! that is all folks!
34596
34597
34598   !*===========================================================================*
34599   !*                              exit                                         *
34600   !*===========================================================================*
34601   ! PUBLIC void exit();
34602   ! Some library routines use exit, so provide a dummy version.
34603   ! Actual calls to exit cannot occur in the kernel.
34604   ! Same for .fat & .trp.
34605
34606   _exit:
34607   __exit:
34608   ___exit:
34609   .fat:
34610   .trp:
34611           sti
34612           jmp __exit
34613
34614
34615   !*===========================================================================*
34616   !*                              in_byte                                      *
34617   !*===========================================================================*
34618   ! PUBLIC unsigned in_byte(port_t port);
34619   ! Read an (unsigned) byte from the i/o port  port  and return it.
34620
34621   _in_byte:
34622           pop     bx
34623           pop     dx              ! port
34624           dec     sp
34625           dec     sp
34626           inb                     ! input 1 byte
34627           subb    ah,ah           ! unsign extend
34628           jmp     (bx)
34629
```

```
34630
34631   !*===========================================================================*
34632   !*                              in_word                                      *
34633   !*===========================================================================*
34634   ! PUBLIC unsigned short in_word(port_t port);
34635   ! Read an (unsigned) word from the i/o port and return it.
34636
34637   _in_word:
34638           pop     bx
34639           pop     dx              ! port
34640           dec     sp
34641           dec     sp              ! added to new klib.s 3/21/91 d.e.c.
34642           inw                     ! input 1 word no sign extend needed
34643           jmp     (bx)
34644
34645
34646   !*===========================================================================*
34647   !*                              out_byte                                     *
34648   !*===========================================================================*
34649   ! PUBLIC void out_byte(port_t port, int value);
34650   ! Write  value  (cast to a byte)  to the I/O port  port.
34651
34652   _out_byte:
34653           pop     bx
34654           pop     dx              ! port
34655           pop     ax              ! value
34656           sub     sp,#2+2
34657           outb                    ! output 1 byte
34658           jmp     (bx)
34659
34660
34661   !*===========================================================================*
34662   !*                              out_word                                     *
34663   !*===========================================================================*
34664   ! PUBLIC void out_word(port_t port, int value);
34665   ! Write  value  (cast to a word)  to the I/O port  port.
34666
34667   _out_word:
34668           pop     bx
34669           pop     dx              ! port
34670           pop     ax              ! value
34671           sub     sp,#2+2
34672           outw                    ! output 1 word
34673           jmp     (bx)
34674
34675
34676   !*===========================================================================*
34677   !*                              port_read                                    *
34678   !*===========================================================================*
34679   ! PUBLIC void port_read(port_t port, phys_bytes destination,unsigned bytcount);
34680   ! Transfer data from (hard disk controller) port to memory.
34681
34682   _port_read:
34683           push    bp
34684           mov     bp,sp
34685           push    di
34686           push    es
34687
34688           call    portio_setup
34689           shr     cx,#1           ! count in words
```

```
34690           mov     di,bx           ! di = destination offset
34691           mov     es,ax           ! es = destination segment
34692           rep
34693           ins
34694
34695           pop     es
34696           pop     di
34697           pop     bp
34698           ret
34699
34700   portio_setup:
34701           mov     ax,4+2(bp)      ! source/destination address in dx:ax
34702           mov     dx,4+2+2(bp)
34703           mov     bx,ax
34704           and     bx,#OFF_MASK    ! bx = offset = address % 16
34705           andb    dl,#HCHIGH_MASK ! ax = segment = address / 16 % 0x10000
34706           andb    al,#HCLOW_MASK
34707           orb     al,dl
34708           movb    cl,#HCLICK_SHIFT
34709           ror     ax,cl
34710           mov     cx,4+2+4(bp)    ! count in bytes
34711           mov     dx,4(bp)        ! port to read from
34712           cld                     ! direction is UP
34713           ret
34714
34715
34716   !*===========================================================================*
34717   !*                              port_read_byte                               *
34718   !*===========================================================================*
34719   ! PUBLIC void port_read_byte(port_t port, phys_bytes destination,
34720   !                                                       unsigned bytcount);
34721   ! Transfer data port to memory.
34722
34723   _port_read_byte:
34724           push    bp
34725           mov     bp,sp
34726           push    di
34727           push    es
34728
34729           call    portio_setup
34730           mov     di,bx           ! di = destination offset
34731           mov     es,ax           ! es = destination segment
34732           rep
34733           insb
34734
34735           pop     es
34736           pop     di
34737           pop     bp
34738           ret
34739
34740
34741   !*===========================================================================*
34742   !*                              port_write                                   *
34743   !*===========================================================================*
34744   ! PUBLIC void port_write(port_t port, phys_bytes source, unsigned bytcount);
34745   ! Transfer data from memory to (hard disk controller) port.
34746
34747   _port_write:
34748           push    bp
34749           mov     bp,sp
```

```
34750           push    si
34751           push    ds
34752
34753           call    portio_setup
34754           shr     cx,#1           ! count in words
34755           mov     si,bx           ! si = source offset
34756           mov     ds,ax           ! ds = source segment
34757           rep
34758           outs
34759
34760           pop     ds
34761           pop     si
34762           pop     bp
34763           ret
34764
34765
34766   !*===========================================================================*
34767   !*                              port_write_byte                              *
34768   !*===========================================================================*
34769   ! PUBLIC void port_write_byte(port_t port, phys_bytes source,
34770   !                                                     unsigned bytcount);
34771   ! Transfer data from memory to port.
34772
34773   _port_write_byte:
34774           push    bp
34775           mov     bp,sp
34776           push    si
34777           push    ds
34778
34779           call    portio_setup
34780           mov     si,bx           ! si = source offset
34781           mov     ds,ax           ! ds = source segment
34782           rep
34783           outsb
34784
34785           pop     ds
34786           pop     si
34787           pop     bp
34788           ret
34789
34790
34791   !*===========================================================================*
34792   !*                              lock                                         *
34793   !*===========================================================================*
34794   ! PUBLIC void lock();
34795   ! Disable CPU interrupts.
34796
34797   _lock:
34798           cli                             ! disable interrupts
34799           ret
34800
34801
34802   !*===========================================================================*
34803   !*                              unlock                                       *
34804   !*===========================================================================*
34805   ! PUBLIC void unlock();
34806   ! Enable CPU interrupts.
34807
34808   _unlock:
34809           sti
```

```
34810           ret
34811
34812
34813   !*===========================================================================*
34814   !*                              enable_irq                                   *
34815   !*===========================================================================*/
34816   ! PUBLIC void enable_irq(unsigned irq)
34817   ! Enable an interrupt request line by clearing an 8259 bit.
34818   ! Equivalent code for irq < 8:
34819   !       out_byte(INT_CTLMASK, in_byte(INT_CTLMASK) & ~(1 << irq));
34820
34821   _enable_irq:
34822           mov     bx, sp
34823           mov     cx, 2(bx)               ! irq
34824           pushf
34825           cli
34826           movb    ah, #~1
34827           rolb    ah, cl                  ! ah = ~(1 << (irq % 8))
34828           cmpb    cl, #8
34829           jae     enable_8                ! enable irq >= 8 at the slave 8259
34830   enable_0:
34831           inb     INT_CTLMASK
34832           andb    al, ah
34833           outb    INT_CTLMASK             ! clear bit at master 8259
34834           popf
34835           ret
34836   enable_8:
34837           inb     INT2_CTLMASK
34838           andb    al, ah
34839           outb    INT2_CTLMASK            ! clear bit at slave 8259
34840           popf
34841           ret
34842
34843
34844   !*===========================================================================*
34845   !*                              disable_irq                                  *
34846   !*===========================================================================*/
34847   ! PUBLIC int disable_irq(unsigned irq)
34848   ! Disable an interrupt request line by setting an 8259 bit.
34849   ! Equivalent code for irq < 8:
34850   !       out_byte(INT_CTLMASK, in_byte(INT_CTLMASK) | (1 << irq));
34851   ! Returns true iff the interrupt was not already disabled.
34852
34853   _disable_irq:
34854           mov     bx, sp
34855           mov     cx, 2(bx)               ! irq
34856           pushf
34857           cli
34858           movb    ah, #1
34859           rolb    ah, cl                  ! ah = (1 << (irq % 8))
34860           cmpb    cl, #8
34861           jae     disable_8               ! disable irq >= 8 at the slave 8259
34862   disable_0:
34863           inb     INT_CTLMASK
34864           testb   al, ah
34865           jnz     dis_already             ! already disabled?
34866           orb     al, ah
34867           outb    INT_CTLMASK             ! set bit at master 8259
34868           popf
34869           mov     ax, #1                  ! disabled by this function
```

```
34870           ret
34871   disable_8:
34872           inb     INT2_CTLMASK
34873           testb   al, ah
34874           jnz     dis_already             ! already disabled?
34875           orb     al, ah
34876           outb    INT2_CTLMASK            ! set bit at slave 8259
34877           popf
34878           mov     ax, #1                  ! disabled by this function
34879           ret
34880   dis_already:
34881           popf
34882           xor     ax, ax                  ! already disabled
34883           ret
34884
34885
34886   !*===========================================================================*
34887   !*                              phys_copy                                    *
34888   !*===========================================================================*
34889   ! PUBLIC void phys_copy(phys_bytes source, phys_bytes destination,
34890   !                       phys_bytes bytecount);
34891   ! Copy a block of physical memory.
34892
34893   SRCLO   =       4
34894   SRCHI   =       6
34895   DESTLO  =       8
34896   DESTHI  =       10
34897   COUNTLO =       12
34898   COUNTHI =       14
34899
34900   _phys_copy:
34901           push    bp              ! save only registers required by C
34902           mov     bp,sp           ! set bp to point to source arg less 4
34903
34904           push    si              ! save si
34905           push    di              ! save di
34906           push    ds              ! save ds
34907           push    es              ! save es
34908
34909           mov     ax,SRCLO(bp)    ! dx:ax = source address (dx is NOT segment)
34910           mov     dx,SRCHI(bp)
34911           mov     si,ax           ! si = source offset = address % 16
34912           and     si,#OFF_MASK
34913           andb    dl,#HCHIGH_MASK ! ds = source segment = address / 16 % 0x10000
34914           andb    al,#HCLOW_MASK
34915           orb     al,dl           ! now bottom 4 bits of dx are in ax
34916           movb    cl,#HCLICK_SHIFT ! rotate them to the top 4
34917           ror     ax,cl
34918           mov     ds,ax
34919
34920           mov     ax,DESTLO(bp)   ! dx:ax = destination addr (dx is NOT segment)
34921           mov     dx,DESTHI(bp)
34922           mov     di,ax           ! di = dest offset = address % 16
34923           and     di,#OFF_MASK
34924           andb    dl,#HCHIGH_MASK ! es = dest segment = address / 16 % 0x10000
34925           andb    al,#HCLOW_MASK
34926           orb     al,dl
34927           ror     ax,cl
34928           mov     es,ax
34929
```

```
34930           mov     ax,COUNTLO(bp)  ! dx:ax = remaining count
34931           mov     dx,COUNTHI(bp)
34932
34933   ! copy upwards (cannot handle overlapped copy)
34934
34935   pc_loop:
34936           mov     cx,ax           ! provisional count for this iteration
34937           test    ax,ax           ! if count >= 0x8000, only do 0x8000 per iter
34938           js      pc_bigcount     ! low byte already >= 0x8000
34939           test    dx,dx
34940           jz      pc_upcount      ! less than 0x8000
34941   pc_bigcount:
34942           mov     cx,#0x8000      ! use maximum count per iteration
34943   pc_upcount:
34944           sub     ax,cx           ! update count
34945           sbb     dx,#0           ! cannot underflow, so carry clear now for rcr
34946           rcr     cx,#1           ! count in words, carry remembers if byte
34947           jnb     pc_even         ! no odd byte
34948           movb                    ! copy odd byte
34949   pc_even:
34950           rep                     ! copy 1 word at a time
34951           movs                    ! word copy
34952
34953           mov     cx,ax           ! test if remaining count is 0
34954           or      cx,dx
34955           jnz     pc_more         ! more to do
34956
34957           pop     es              ! restore es
34958           pop     ds              ! restore ds
34959           pop     di              ! restore di
34960           pop     si              ! restore si
34961           pop     bp              ! restore bp
34962           ret                     ! return to caller
34963
34964   pc_more:
34965           sub     si,#0x8000      ! adjust pointers so the offset does not
34966           mov     cx,ds           ! overflow in the next 0x8000 bytes
34967           add     cx,#0x800       ! pointers end up same physical location
34968           mov     ds,cx           ! the current offsets are known >= 0x8000
34969           sub     di,#0x8000      ! since we just copied that many
34970           mov     cx,es
34971           add     cx,#0x800
34972           mov     es,cx
34973           jmp     pc_loop         ! start next iteration
34974
34975
34976   !*===========================================================================*
34977   !*                              mem_rdw                                      *
34978   !*===========================================================================*
34979   ! PUBLIC u16_t mem_rdw(u16_t segment, u16_t *offset);
34980   ! Load and return the word at the far pointer  segment:offset.
34981
34982   _mem_rdw:
34983           mov     cx,ds           ! save ds
34984           pop     dx              ! return adr
34985           pop     ds              ! segment
34986           pop     bx              ! offset
34987           sub     sp,#2+2         ! adjust for parameters popped
34988           mov     ax,(bx)         ! load the word to return
34989           mov     ds,cx           ! restore ds
```

```
34990           jmp     (dx)            ! return
34991
34992
34993   !*===========================================================================*
34994   !*                              reset                                        *
34995   !*===========================================================================*
34996   ! PUBLIC void reset();
34997   ! Reset the system.
34998   ! In real mode we simply jump to the reset address.
34999
35000   _reset:
35001           jmpf    0,0xFFFF
35002
35003
35004   !*===========================================================================*
35005   !*                              mem_vid_copy                                 *
35006   !*===========================================================================*
35007   ! PUBLIC void mem_vid_copy(u16 *src, unsigned dst, unsigned count);
35008   !
35009   ! Copy count characters from kernel memory to video memory.  Src, dst and
35010   ! count are character (word) based video offsets and counts.  If src is null
35011   ! then screen memory is blanked by filling it with blank_color.
35012
35013   MVC_ARGS        =       2 + 2 + 2 + 2   ! 2 + 2 + 2
35014   !                       es  di  si  ip   src dst ct
35015
35016   _mem_vid_copy:
35017           push    si
35018           push    di
35019           push    es
35020           mov     bx, sp
35021           mov     si, MVC_ARGS(bx)        ! source
35022           mov     di, MVC_ARGS+2(bx)      ! destination
35023           mov     dx, MVC_ARGS+2+2(bx)    ! count
35024           mov     es, _vid_seg            ! destination is video segment
35025           cld                             ! make sure direction is up
35026   mvc_loop:
35027           and     di, _vid_mask           ! wrap address
35028           mov     cx, dx                  ! one chunk to copy
35029           mov     ax, _vid_size
35030           sub     ax, di
35031           cmp     cx, ax
35032           jbe     0f
35033           mov     cx, ax                  ! cx = min(cx, vid_size - di)
35034   0:      sub     dx, cx                  ! count -= cx
35035           shl     di, #1                  ! byte address
35036           test    si, si                  ! source == 0 means blank the screen
35037           jz      mvc_blank
35038   mvc_copy:
35039           rep                             ! copy words to video memory
35040           movs
35041           jmp     mvc_test
35042   mvc_blank:
35043           mov     ax, _blank_color        ! ax = blanking character
35044           rep
35045           stos                            ! copy blanks to video memory
35046           !jmp    mvc_test
35047   mvc_test:
35048           shr     di, #1                  ! word addresses
35049           test    dx, dx
```

```
35050           jnz     mvc_loop
35051   mvc_done:
35052           pop     es
35053           pop     di
35054           pop     si
35055           ret
35056
35057
35058   !*===========================================================================*
35059   !*                              vid_vid_copy                                 *
35060   !*===========================================================================*
35061   ! PUBLIC void vid_vid_copy(unsigned src, unsigned dst, unsigned count);
35062   !
35063   ! Copy count characters from video memory to video memory.  Handle overlap.
35064   ! Used for scrolling, line or character insertion and deletion.  Src, dst
35065   ! and count are character (word) based video offsets and counts.
35066
35067   VVC_ARGS        =       2 + 2 + 2 + 2   ! 2 + 2 + 2
35068   !                       es  di  si  ip   src dst ct
35069
35070   _vid_vid_copy:
35071           push    si
35072           push    di
35073           push    es
35074           mov     bx, sp
35075           mov     si, VVC_ARGS(bx)        ! source
35076           mov     di, VVC_ARGS+2(bx)      ! destination
35077           mov     dx, VVC_ARGS+2+2(bx)    ! count
35078           mov     es, _vid_seg            ! use video segment
35079           cmp     si, di                  ! copy up or down?
35080           jb      vvc_down
35081   vvc_up:
35082           cld                             ! direction is up
35083   vvc_uploop:
35084           and     si, _vid_mask           ! wrap addresses
35085           and     di, _vid_mask
35086           mov     cx, dx                  ! one chunk to copy
35087           mov     ax, _vid_size
35088           sub     ax, si
35089           cmp     cx, ax
35090           jbe     0f
35091           mov     cx, ax                  ! cx = min(cx, vid_size - si)
35092   0:      mov     ax, _vid_size
35093           sub     ax, di
35094           cmp     cx, ax
35095           jbe     0f
35096           mov     cx, ax                  ! cx = min(cx, vid_size - di)
35097   0:      sub     dx, cx                  ! count -= cx
35098           shl     si, #1
35099           shl     di, #1                  ! byte addresses
35100           rep
35101      eseg movs                            ! copy video words
35102           shr     si, #1
35103           shr     di, #1                  ! word addresses
35104           test    dx, dx
35105           jnz     vvc_uploop              ! again?
35106           jmp     vvc_done
35107   vvc_down:
35108           std                             ! direction is down
35109           add     si, dx                  ! start copying at the top
```

```
35110           dec     si
35111           add     di, dx
35112           dec     di
35113   vvc_downloop:
35114           and     si, _vid_mask           ! wrap addresses
35115           and     di, _vid_mask
35116           mov     cx, dx                  ! one chunk to copy
35117           lea     ax, 1(si)
35118           cmp     cx, ax
35119           jbe     0f
35120           mov     cx, ax                  ! cx = min(cx, si + 1)
35121   0:      lea     ax, 1(di)
35122           cmp     cx, ax
35123           jbe     0f
35124           mov     cx, ax                  ! cx = min(cx, di + 1)
35125   0:      sub     dx, cx                  ! count -= cx
35126           shl     si, #1
35127           shl     di, #1                  ! byte addresses
35128           rep
35129      eseg movs                            ! copy video words
35130           shr     si, #1
35131           shr     di, #1                  ! word addresses
35132           test    dx, dx
35133           jnz     vvc_downloop            ! again?
35134           cld                             ! C compiler expect up
35135           !jmp    vvc_done
35136   vvc_done:
35137           pop     es
35138           pop     di
35139           pop     si
35140           ret
35141
35142
35143   !*===========================================================================*
35144   !*                              level0                                       *
35145   !*===========================================================================*
35146   ! PUBLIC void level0(void (*func)(void))
35147   ! Not very interesting in real mode, see p_level0.
35148   !
35149   _level0:
35150           mov     bx, sp
35151           jmp     @2(bx)
35152
35153
35154   !*===========================================================================*
35155   !*                              klib_init_prot                               *
35156   !*===========================================================================*
35157   ! PUBLIC void klib_init_prot();
35158   ! Initialize klib for protected mode by patching some real mode functions
35159   ! at their starts to jump to their protected mode equivalents, according to
35160   ! the patch table.  Saves a lot of tests on the "protected_mode" variable.
35161   ! Note that this function must be run in real mode, for it writes the code
35162   ! segment.  (One otherwise has to set up a descriptor, etc, etc.)
35163
35164   klib_init_prot:
35165           mov     si,#patch_table
35166   kip_next:
35167           lods                    ! original function
35168           mov     bx,ax
35169     cseg  movb    (bx),#JMP_OPCODE ! overwrite start of function by a long jump
```

```
35170           lods                    ! new function - target of jump
35171           sub     ax,bx           ! relative jump
35172           sub     ax,#3           ! adjust by length of jump instruction
35173     cseg  mov     1(bx),ax        ! set address
35174           cmp     si,#end_patch_table ! end of table?
35175           jb      kip_next
35176   kip_done:
35177           ret
35178
35179
35180   !*===========================================================================*
35181   !*                      variants for protected mode                          *
35182   !*===========================================================================*
35183   ! Some routines are different in protected mode.
35184   ! The only essential difference is the handling of segment registers.
35185   ! One complication is that the method of building segment descriptors is not
35186   ! reentrant, so the protected mode versions must not be called by interrupt
35187   ! handlers.
35188
35189
35190   !*===========================================================================*
35191   !*                              p_cp_mess                                    *
35192   !*===========================================================================*
35193   ! The real mode version attempts to be efficient by passing raw segments but
35194   ! that just gets in the way here.
35195
35196   p_cp_mess:
35197           cld
35198           pop     dx
35199           pop     bx              ! proc
35200           pop     cx              ! source clicks
35201           pop     ax              ! source offset
35202   #if CLICK_SHIFT != HCLICK_SHIFT + 4
35203   #error /* the only click size supported is 256, to avoid slow shifts here */
35204   #endif
35205           addb    ah,cl           ! calculate source offset
35206           adcb    ch,#0           ! and put in base of source descriptor
35207           mov     _gdt+DS_286_OFFSET+DESC_BASE,ax
35208           movb    _gdt+DS_286_OFFSET+DESC_BASE_MIDDLE,ch
35209           pop     cx              ! destination clicks
35210           pop     ax              ! destination offset
35211           addb    ah,cl           ! calculate destination offset
35212           adcb    ch,#0           ! and put in base of destination descriptor
35213           mov     _gdt+ES_286_OFFSET+DESC_BASE,ax
35214           movb    _gdt+ES_286_OFFSET+DESC_BASE_MIDDLE,ch
35215           sub     sp,#2+2+2+2+2
35216
35217           push    ds
35218           push    es
35219           mov     ax,#DS_286_SELECTOR
35220           mov     ds,ax
35221           mov     ax,#ES_286_SELECTOR
35222           mov     es,ax
35223
35224     eseg  mov     0,bx            ! proc no. of sender from arg, not msg
35225           mov     ax,si
35226           mov     bx,di
35227           mov     si,#2           ! src offset is now 2 relative to start of seg
35228           mov     di,si           ! and destination offset
35229           mov     cx,#Msize-1     ! word count
```

```
35230           rep
35231           movs
35232           mov     di,bx
35233           mov     si,ax
35234           pop     es
35235           pop     ds
35236           jmp     (dx)
35237
35238
35239   !*===========================================================================*
35240   !*                              p_portio_setup                               *
35241   !*===========================================================================*
35242   ! The port_read, port_write, etc. functions need a setup routine that uses
35243   ! a segment descriptor.
35244   p_portio_setup:
35245           mov     ax,4+2(bp)      ! source/destination address in dx:ax
35246           mov     dx,4+2+2(bp)
35247           mov     _gdt+DS_286_OFFSET+DESC_BASE,ax
35248           movb    _gdt+DS_286_OFFSET+DESC_BASE_MIDDLE,dl
35249           xor     bx,bx           ! bx = 0 = start of segment
35250           mov     ax,#DS_286_SELECTOR ! ax = segment selector
35251           mov     cx,4+2+4(bp)    ! count in bytes
35252           mov     dx,4(bp)        ! port to read from
35253           cld                     ! direction is UP
35254           ret
35255
35256
35257   !*===========================================================================*
35258   !*                              p_phys_copy                                  *
35259   !*===========================================================================*
35260   p_phys_copy:
35261           cld
35262           pop     dx
35263           pop     _gdt+DS_286_OFFSET+DESC_BASE
35264           pop     ax              ! pop source into base of source descriptor
35265           movb    _gdt+DS_286_OFFSET+DESC_BASE_MIDDLE,al
35266           pop     _gdt+ES_286_OFFSET+DESC_BASE
35267           pop     ax              ! pop destination into base of dst descriptor
35268           movb    _gdt+ES_286_OFFSET+DESC_BASE_MIDDLE,al
35269           pop     cx              ! byte count in bx:cx
35270           pop     bx
35271           sub     sp,#4+4+4
35272
35273           push    di
35274           push    si
35275           push    es
35276           push    ds
35277           sub     si,si           ! src offset is now 0 relative to start of seg
35278           mov     di,si           ! and destination offset
35279           jmp     ppc_next
35280
35281   ! It is too much trouble to align the segment bases, so word alignment is hard.
35282   ! Avoiding the book-keeping for alignment may be good anyway.
35283
35284   ppc_large:
35285           push    cx
35286           mov     cx,#0x8000      ! copy a large chunk of this many words
35287           rep
35288           movs
35289           pop     cx
```

```
35290           dec     bx
35291           pop     ds              ! update the descriptors
35292           incb    _gdt+DS_286_OFFSET+DESC_BASE_MIDDLE
35293           incb    _gdt+ES_286_OFFSET+DESC_BASE_MIDDLE
35294           push    ds
35295   ppc_next:
35296           mov     ax,#DS_286_SELECTOR     ! (re)load the selectors
35297           mov     ds,ax
35298           mov     ax,#ES_286_SELECTOR
35299           mov     es,ax
35300           test    bx,bx
35301           jnz     ppc_large
35302
35303           shr     cx,#1           ! word count
35304           rep
35305           movs                    ! move any leftover words
35306           rcl     cx,#1           ! restore old bit 0
35307           rep
35308           movb                    ! move any leftover byte
35309           pop     ds
35310           pop     es
35311           pop     si
35312           pop     di
35313           jmp     (dx)
35314
35315   !*===========================================================================*
35316   !*                              p_reset                                      *
35317   !*===========================================================================*
35318   ! Reset the system by loading IDT with offset 0 and interrupting.
35319
35320   p_reset:
35321           lidt    idt_zero
35322           int     3               ! anything goes, the 286 will not like it
35323
35324
35325   !*===========================================================================*
35326   !*                              p_level0                                     *
35327   !*===========================================================================*
35328   ! PUBLIC void level0(void (*func)(void))
35329   ! Call a function at permission level 0.  This allows kernel tasks to do
35330   ! things that are only possible at the most privileged CPU level.
35331   !
35332   p_level0:
35333           mov     bx, sp
35334           mov     ax, 2(bx)
35335           mov     _level0_func, ax
35336           int     LEVEL0_VECTOR
35337           ret
35338
35339
35340   !*===========================================================================*
35341   !*                              data                                         *
35342   !*===========================================================================*
35343           .data
35344   patch_table:                    ! pairs (old function, new function)
35345   #if ENABLE_BIOS_WINI
35346           .data2  _bios13, p_bios13
35347   #endif
35348           .data2  _cp_mess, p_cp_mess
35349           .data2  _phys_copy, p_phys_copy
```

```
35350             .data2  portio_setup, p_portio_setup
35351             .data2  _reset, p_reset
35352             .data2  _level0, p_level0
35353             .data2  _restart, p_restart     ! in mpx file
35354             .data2  save, p_save    ! in mpx file
35355     end_patch_table:                ! end of table
35356
35357     idt_vectors:                    ! limit and base of real mode interrupt vectors
35358             .data2  0x3FF
35359     idt_zero:                       ! zero limit IDT to cause a processor shutdown
35360             .data2  0, 0, 0
35361
35362             .bss
35363     save_sp:                        ! place to put sp when switching to real mode
35364             .space  2
35365     msw:                            ! saved real mode machine status word
35366             .space  2
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/mpx.s
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

35400     #
35401     ! Chooses between the 8086 and 386 versions of the Minix startup code.
35402
35403     #include <minix/config.h>
35404     #if _WORD_SIZE == 2
35405     #include "mpx88.s"
35406     #else
35407     #include "mpx386.s"
35408     #endif
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/mpx386.s
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

35500     #
35501     ! This file contains the assembler startup code for Minix and the 32-bit
35502     ! interrupt handlers.  It cooperates with start.c to set up a good
35503     ! environment for main().
35504
35505     ! This file is part of the lowest layer of the MINIX kernel.  The other part
35506     ! is "proc.c".  The lowest layer does process switching and message handling.
35507
35508     ! Every transition to the kernel goes through this file.  Transitions are
35509     ! caused by sending/receiving messages and by most interrupts.  (RS232
35510     ! interrupts may be handled in the file "rs2.s" and then they rarely enter
35511     ! the kernel.)
35512
35513     ! Transitions to the kernel may be nested.  The initial entry may be with a
35514     ! system call, exception or hardware interrupt; reentries may only be made
35515     ! by hardware interrupts.  The count of reentries is kept in "k_reenter".
35516     ! It is important for deciding whether to switch to the kernel stack and
35517     ! for protecting the message passing code in "proc.c".
35518
35519     ! For the message passing trap, most of the machine state is saved in the
```

```
35520   ! proc table.  (Some of the registers need not be saved.)  Then the stack is
35521   ! switched to "k_stack", and interrupts are reenabled.  Finally, the system
35522   ! call handler (in C) is called.  When it returns, interrupts are disabled
35523   ! again and the code falls into the restart routine, to finish off held-up
35524   ! interrupts and run the process or task whose pointer is in "proc_ptr".
35525
35526   ! Hardware interrupt handlers do the same, except  (1) The entire state must
35527   ! be saved.  (2) There are too many handlers to do this inline, so the save
35528   ! routine is called.  A few cycles are saved by pushing the address of the
35529   ! appropiate restart routine for a return later.  (3) A stack switch is
35530   ! avoided when the stack is already switched.  (4) The (master) 8259 interrupt
35531   ! controller is reenabled centrally in save().  (5) Each interrupt handler
35532   ! masks its interrupt line using the 8259 before enabling (other unmasked)
35533   ! interrupts, and unmasks it after servicing the interrupt.  This limits the
35534   ! nest level to the number of lines and protects the handler from itself.
35535
35536   ! For communication with the boot monitor at startup time some constant
35537   ! data are compiled into the beginning of the text segment. This facilitates
35538   ! reading the data at the start of the boot process, since only the first
35539   ! sector of the file needs to be read.
35540
35541   ! Some data storage is also allocated at the end of this file. This data
35542   ! will be at the start of the data segment of the kernel and will be read
35543   ! and modified by the boot monitor before the kernel starts.
35544
35545   ! sections
35546
35547   .sect .text
35548   begtext:
35549   .sect .rom
35550   begrom:
35551   .sect .data
35552   begdata:
35553   .sect .bss
35554   begbss:
35555
35556   #include <minix/config.h>
35557   #include <minix/const.h>
35558   #include <minix/com.h>
35559   #include "const.h"
35560   #include "protect.h"
35561   #include "sconst.h"
35562
35563   /* Selected 386 tss offsets. */
35564   #define TSS3_S_SP0      4
35565
35566   ! Exported functions
35567   ! Note: in assembly language the .define statement applied to a function name
35568   ! is loosely equivalent to a prototype in C code -- it makes it possible to
35569   ! link to an entity declared in the assembly code but does not create
35570   ! the entity.
35571
35572   .define _idle_task
35573   .define _restart
35574   .define save
35575
35576   .define _divide_error
35577   .define _single_step_exception
35578   .define _nmi
35579   .define _breakpoint_exception
```

```
35580    .define _overflow
35581    .define _bounds_check
35582    .define _inval_opcode
35583    .define _copr_not_available
35584    .define _double_fault
35585    .define _copr_seg_overrun
35586    .define _inval_tss
35587    .define _segment_not_present
35588    .define _stack_exception
35589    .define _general_protection
35590    .define _page_fault
35591    .define _copr_error
35592
35593    .define _hwint00        ! handlers for hardware interrupts
35594    .define _hwint01
35595    .define _hwint02
35596    .define _hwint03
35597    .define _hwint04
35598    .define _hwint05
35599    .define _hwint06
35600    .define _hwint07
35601    .define _hwint08
35602    .define _hwint09
35603    .define _hwint10
35604    .define _hwint11
35605    .define _hwint12
35606    .define _hwint13
35607    .define _hwint14
35608    .define _hwint15
35609
35610    .define _s_call
35611    .define _p_s_call
35612    .define _level0_call
35613
35614    ! Imported functions.
35615
35616    .extern _cstart
35617    .extern _main
35618    .extern _exception
35619    .extern _interrupt
35620    .extern _sys_call
35621    .extern _unhold
35622
35623    ! Exported variables.
35624    ! Note: when used with a variable the .define does not reserve storage,
35625    ! it makes the name externally visible so it may be linked to.
35626
35627    .define begbss
35628    .define begdata
35629    .define _sizes
35630
35631    ! Imported variables.
35632
35633    .extern _gdt
35634    .extern _code_base
35635    .extern _data_base
35636    .extern _held_head
35637    .extern _k_reenter
35638    .extern _pc_at
35639    .extern _proc_ptr
```

```
35640   .extern _ps_mca
35641   .extern _tss
35642   .extern _level0_func
35643   .extern _mon_sp
35644   .extern _mon_return
35645   .extern _reboot_code
35646
35647   .sect .text
35648   !*===========================================================================*
35649   !*                                MINIX                                      *
35650   !*===========================================================================*
35651   MINIX:                          ! this is the entry point for the MINIX kernel
35652           jmp     over_flags      ! skip over the next few bytes
35653           .data2  CLICK_SHIFT     ! for the monitor: memory granularity
35654   flags:
35655           .data2  0x002D          ! boot monitor flags:
35656                                   !       call in 386 mode, make stack,
35657                                   !       load high, will return
35658           nop                     ! extra byte to sync up disassembler
35659   over_flags:
35660
35661   ! Set up a C stack frame on the monitor stack.  (The monitor sets cs and ds
35662   ! right.  The ss descriptor still references the monitor data segment.)
35663           movzx   esp, sp         ! monitor stack is a 16 bit stack
35664           push    ebp
35665           mov     ebp, esp
35666           push    esi
35667           push    edi
35668           cmp     4(ebp), 0       ! nonzero if return possible
35669           jz      noret
35670           inc     (_mon_return)
35671   noret:  mov     (_mon_sp), esp  ! save stack pointer for later return
35672
35673   ! Copy the monitor global descriptor table to the address space of kernel and
35674   ! switch over to it.  Prot_init() can then update it with immediate effect.
35675
35676           sgdt    (_gdt+GDT_SELECTOR)             ! get the monitor gdtr
35677           mov     esi, (_gdt+GDT_SELECTOR+2)      ! absolute address of GDT
35678           mov     ebx, _gdt                       ! address of kernel GDT
35679           mov     ecx, 8*8                        ! copying eight descriptors
35680   copygdt:
35681    eseg   movb    al, (esi)
35682           movb    (ebx), al
35683           inc     esi
35684           inc     ebx
35685           loop    copygdt
35686           mov     eax, (_gdt+DS_SELECTOR+2)       ! base of kernel data
35687           and     eax, 0x00FFFFFF                 ! only 24 bits
35688           add     eax, _gdt                       ! eax = vir2phys(gdt)
35689           mov     (_gdt+GDT_SELECTOR+2), eax      ! set base of GDT
35690           lgdt    (_gdt+GDT_SELECTOR)             ! switch over to kernel GDT
35691
35692   ! Locate boot parameters, set up kernel segment registers and stack.
35693           mov     ebx, 8(ebp)     ! boot parameters offset
35694           mov     edx, 12(ebp)    ! boot parameters length
35695           mov     ax, ds          ! kernel data
35696           mov     es, ax
35697           mov     fs, ax
35698           mov     gs, ax
35699           mov     ss, ax
```

```
35700           mov     esp, k_stktop   ! set sp to point to the top of kernel stack
35701
35702   ! Call C startup code to set up a proper environment to run main().
35703           push    edx
35704           push    ebx
35705           push    SS_SELECTOR
35706           push    MON_CS_SELECTOR
35707           push    DS_SELECTOR
35708           push    CS_SELECTOR
35709           call    _cstart         ! cstart(cs, ds, mcs, mds, parmoff, parmlen)
35710           add     esp, 6*4
35711
35712   ! Reload gdtr, idtr and the segment registers to global descriptor table set
35713   ! up by prot_init().
35714
35715           lgdt    (_gdt+GDT_SELECTOR)
35716           lidt    (_gdt+IDT_SELECTOR)
35717
35718           jmpf    CS_SELECTOR:csinit
35719   csinit:
35720       o16 mov     ax, DS_SELECTOR
35721           mov     ds, ax
35722           mov     es, ax
35723           mov     fs, ax
35724           mov     gs, ax
35725           mov     ss, ax
35726       o16 mov     ax, TSS_SELECTOR        ! no other TSS is used
35727           ltr     ax
35728           push    0                       ! set flags to known good state
35729           popf                            ! esp, clear nested task and int enable
35730
35731           jmp     _main                   ! main()
35732
35733
35734   !*===========================================================================*
35735   !*                              interrupt handlers                           *
35736   !*              interrupt handlers for 386 32-bit protected mode             *
35737   !*===========================================================================*
35738
35739   !*===========================================================================*
35740   !*                              hwint00 - 07                                 *
35741   !*===========================================================================*
35742   ! Note this is a macro, it looks like a subroutine.
35743   #define hwint_master(irq)       \
35744           call    save                    /* save interrupted process state */;\
35745           inb     INT_CTLMASK                                                 ;\
35746           orb     al, [1<<irq]                                                ;\
35747           outb    INT_CTLMASK             /* disable the irq                */;\
35748           movb    al, ENABLE                                                  ;\
35749           outb    INT_CTL                 /* reenable master 8259           */;\
35750           sti                             /* enable interrupts              */;\
35751           push    irq                     /* irq                            */;\
35752           call    (_irq_table + 4*irq)    /* eax = (*irq_table[irq])(irq)   */;\
35753           pop     ecx                                                         ;\
35754           cli                             /* disable interrupts             */;\
35755           test    eax, eax                /* need to reenable irq?          */;\
35756           jz      0f                                                          ;\
35757           inb     INT_CTLMASK                                                 ;\
35758           andb    al, ~[1<<irq]                                               ;\
35759           outb    INT_CTLMASK             /* enable the irq                 */;\
```

```
35760   0:      ret                             /* restart (another) process      */
35761
35762   ! Each of these entry points is an expansion of the hwint_master macro
35763           .align  16
35764   _hwint00:               ! Interrupt routine for irq 0 (the clock).
35765           hwint_master(0)
35766
35767           .align  16
35768   _hwint01:               ! Interrupt routine for irq 1 (keyboard)
35769           hwint_master(1)
35770
35771           .align  16
35772   _hwint02:               ! Interrupt routine for irq 2 (cascade!)
35773           hwint_master(2)
35774
35775           .align  16
35776   _hwint03:               ! Interrupt routine for irq 3 (second serial)
35777           hwint_master(3)
35778
35779           .align  16
35780   _hwint04:               ! Interrupt routine for irq 4 (first serial)
35781           hwint_master(4)
35782
35783           .align  16
35784   _hwint05:               ! Interrupt routine for irq 5 (XT winchester)
35785           hwint_master(5)
35786
35787           .align  16
35788   _hwint06:               ! Interrupt routine for irq 6 (floppy)
35789           hwint_master(6)
35790
35791           .align  16
35792   _hwint07:               ! Interrupt routine for irq 7 (printer)
35793           hwint_master(7)
35794
35795   !*===========================================================================*
35796   !*                              hwint08 - 15                                 *
35797   !*===========================================================================*
35798   ! Note this is a macro, it looks like a subroutine.
35799   #define hwint_slave(irq)        \
35800           call    save                    /* save interrupted process state */;\
35801           inb     INT2_CTLMASK                                                 ;\
35802           orb     al, [1<<[irq-8]]                                             ;\
35803           outb    INT2_CTLMASK            /* disable the irq                */;\
35804           movb    al, ENABLE                                                   ;\
35805           outb    INT_CTL                 /* reenable master 8259           */;\
35806           jmp     .+2                     /* delay                          */;\
35807           outb    INT2_CTL                /* reenable slave 8259            */;\
35808           sti                             /* enable interrupts              */;\
35809           push    irq                     /* irq                            */;\
35810           call    (_irq_table + 4*irq)    /* eax = (*irq_table[irq])(irq)   */;\
35811           pop     ecx                                                          ;\
35812           cli                             /* disable interrupts             */;\
35813           test    eax, eax                /* need to reenable irq?          */;\
35814           jz      0f                                                           ;\
35815           inb     INT2_CTLMASK                                                 ;\
35816           andb    al, ~[1<<[irq-8]]                                            ;\
35817           outb    INT2_CTLMASK            /* enable the irq                 */;\
35818   0:      ret                             /* restart (another) process      */
35819
```

```
35820   ! Each of these entry points is an expansion of the hwint_slave macro
35821           .align  16
35822   _hwint08:               ! Interrupt routine for irq 8 (realtime clock)
35823           hwint_slave(8)
35824
35825           .align  16
35826   _hwint09:               ! Interrupt routine for irq 9 (irq 2 redirected)
35827           hwint_slave(9)
35828
35829           .align  16
35830   _hwint10:               ! Interrupt routine for irq 10
35831           hwint_slave(10)
35832
35833           .align  16
35834   _hwint11:               ! Interrupt routine for irq 11
35835           hwint_slave(11)
35836
35837           .align  16
35838   _hwint12:               ! Interrupt routine for irq 12
35839           hwint_slave(12)
35840
35841           .align  16
35842   _hwint13:               ! Interrupt routine for irq 13 (FPU exception)
35843           hwint_slave(13)
35844
35845           .align  16
35846   _hwint14:               ! Interrupt routine for irq 14 (AT winchester)
35847           hwint_slave(14)
35848
35849           .align  16
35850   _hwint15:               ! Interrupt routine for irq 15
35851           hwint_slave(15)
35852
35853   !*===========================================================================*
35854   !*                               save                                        *
35855   !*===========================================================================*
35856   ! Save for protected mode.
35857   ! This is much simpler than for 8086 mode, because the stack already points
35858   ! into the process table, or has already been switched to the kernel stack.
35859
35860           .align  16
35861   save:
35862           cld                     ! set direction flag to a known value
35863           pushad                  ! save "general" registers
35864       o16 push    ds              ! save ds
35865       o16 push    es              ! save es
35866       o16 push    fs              ! save fs
35867       o16 push    gs              ! save gs
35868           mov     dx, ss          ! ss is kernel data segment
35869           mov     ds, dx          ! load rest of kernel segments
35870           mov     es, dx          ! kernel does not use fs, gs
35871           mov     eax, esp        ! prepare to return
35872           incb    (_k_reenter)    ! from -1 if not reentering
35873           jnz     set_restart1    ! stack is already kernel stack
35874           mov     esp, k_stktop
35875           push    _restart        ! build return address for int handler
35876           xor     ebp, ebp        ! for stacktrace
35877           jmp     RETADR-P_STACKBASE(eax)
35878
35879           .align  4
```

```
35880   set_restart1:
35881           push    restart1
35882           jmp     RETADR-P_STACKBASE(eax)
35883
35884   !*===========================================================================*
35885   !*                              _s_call                                      *
35886   !*===========================================================================*
35887           .align  16
35888   _s_call:
35889   _p_s_call:
35890           cld                     ! set direction flag to a known value
35891           sub     esp, 6*4        ! skip RETADR, eax, ecx, edx, ebx, est
35892           push    ebp             ! stack already points into proc table
35893           push    esi
35894           push    edi
35895       o16 push    ds
35896       o16 push    es
35897       o16 push    fs
35898       o16 push    gs
35899           mov     dx, ss
35900           mov     ds, dx
35901           mov     es, dx
35902           incb    (_k_reenter)
35903           mov     esi, esp        ! assumes P_STACKBASE == 0
35904           mov     esp, k_stktop
35905           xor     ebp, ebp        ! for stacktrace
35906                                   ! end of inline save
35907           sti                     ! allow SWITCHER to be interrupted
35908                                   ! now set up parameters for sys_call()
35909           push    ebx             ! pointer to user message
35910           push    eax             ! src/dest
35911           push    ecx             ! SEND/RECEIVE/BOTH
35912           call    _sys_call       ! sys_call(function, src_dest, m_ptr)
35913                                   ! caller is now explicitly in proc_ptr
35914           mov     AXREG(esi), eax ! sys_call MUST PRESERVE si
35915           cli                     ! disable interrupts
35916
35917   ! Fall into code to restart proc/task running.
35918
35919   !*===========================================================================*
35920   !*                              restart                                      *
35921   !*===========================================================================*
35922   _restart:
35923
35924   ! Flush any held-up interrupts.
35925   ! This reenables interrupts, so the current interrupt handler may reenter.
35926   ! This does not matter, because the current handler is about to exit and no
35927   ! other handlers can reenter since flushing is only done when k_reenter == 0.
35928
35929           cmp     (_held_head), 0 ! do fast test to usually avoid function call
35930           jz      over_call_unhold
35931           call    _unhold         ! this is rare so overhead acceptable
35932   over_call_unhold:
35933           mov     esp, (_proc_ptr)        ! will assume P_STACKBASE == 0
35934           lldt    P_LDT_SEL(esp)          ! enable segment descriptors for task
35935           lea     eax, P_STACKTOP(esp)    ! arrange for next interrupt
35936           mov     (_tss+TSS3_S_SP0), eax  ! to save state in process table
35937   restart1:
35938           decb    (_k_reenter)
35939       o16 pop     gs
```

```
35940         o16 pop     fs
35941         o16 pop     es
35942         o16 pop     ds
35943             popad
35944             add     esp, 4          ! skip return adr
35945             iretd                   ! continue process
35946
35947     !*===========================================================================*
35948     !*                              exception handlers                           *
35949     !*===========================================================================*
35950     _divide_error:
35951             push    DIVIDE_VECTOR
35952             jmp     exception
35953
35954     _single_step_exception:
35955             push    DEBUG_VECTOR
35956             jmp     exception
35957
35958     _nmi:
35959             push    NMI_VECTOR
35960             jmp     exception
35961
35962     _breakpoint_exception:
35963             push    BREAKPOINT_VECTOR
35964             jmp     exception
35965
35966     _overflow:
35967             push    OVERFLOW_VECTOR
35968             jmp     exception
35969
35970     _bounds_check:
35971             push    BOUNDS_VECTOR
35972             jmp     exception
35973
35974     _inval_opcode:
35975             push    INVAL_OP_VECTOR
35976             jmp     exception
35977
35978     _copr_not_available:
35979             push    COPROC_NOT_VECTOR
35980             jmp     exception
35981
35982     _double_fault:
35983             push    DOUBLE_FAULT_VECTOR
35984             jmp     errexception
35985
35986     _copr_seg_overrun:
35987             push    COPROC_SEG_VECTOR
35988             jmp     exception
35989
35990     _inval_tss:
35991             push    INVAL_TSS_VECTOR
35992             jmp     errexception
35993
35994     _segment_not_present:
35995             push    SEG_NOT_VECTOR
35996             jmp     errexception
35997
35998     _stack_exception:
35999             push    STACK_FAULT_VECTOR
```

```
36000           jmp     errexception
36001
36002   _general_protection:
36003           push    PROTECTION_VECTOR
36004           jmp     errexception
36005
36006   _page_fault:
36007           push    PAGE_FAULT_VECTOR
36008           jmp     errexception
36009
36010   _copr_error:
36011           push    COPROC_ERR_VECTOR
36012           jmp     exception
36013
36014   !*===========================================================================*
36015   !*                               exception                                   *
36016   !*===========================================================================*
36017   ! This is called for all exceptions which do not push an error code.
36018
36019           .align  16
36020   exception:
36021    sseg   mov     (trap_errno), 0         ! clear trap_errno
36022    sseg   pop     (ex_number)
36023           jmp     exception1
36024
36025   !*===========================================================================*
36026   !*                               errexception                                *
36027   !*===========================================================================*
36028   ! This is called for all exceptions which push an error code.
36029
36030           .align  16
36031   errexception:
36032    sseg   pop     (ex_number)
36033    sseg   pop     (trap_errno)
36034   exception1:                             ! Common for all exceptions.
36035           push    eax                     ! eax is scratch register
36036           mov     eax, 0+4(esp)           ! old eip
36037    sseg   mov     (old_eip), eax
36038           movzx   eax, 4+4(esp)           ! old cs
36039    sseg   mov     (old_cs), eax
36040           mov     eax, 8+4(esp)           ! old eflags
36041    sseg   mov     (old_eflags), eax
36042           pop     eax
36043           call    save
36044           push    (old_eflags)
36045           push    (old_cs)
36046           push    (old_eip)
36047           push    (trap_errno)
36048           push    (ex_number)
36049           call    _exception              ! (ex_number, trap_errno, old_eip,
36050                                           !       old_cs, old_eflags)
36051           add     esp, 5*4
36052           cli
36053           ret
36054
36055   !*===========================================================================*
36056   !*                               level0_call                                 *
36057   !*===========================================================================*
36058   _level0_call:
36059           call    save
```

```
36060           jmp     (_level0_func)
36061
36062   !*===========================================================================*
36063   !*                              idle_task                                    *
36064   !*===========================================================================*
36065   _idle_task:                     ! executed when there is no work
36066           jmp     _idle_task      ! a "hlt" before this fails in protected mode
36067
36068   !*===========================================================================*
36069   !*                              data                                         *
36070   !*===========================================================================*
36071   ! These declarations assure that storage will be allocated at the very
36072   ! beginning of the kernel data section, so the boot monitor can be easily
36073   ! told how to patch these locations. Note that the magic number is put
36074   ! here by the compiler, but will be read by, and then overwritten by,
36075   ! the boot monitor. When the kernel starts the sizes array will be
36076   ! found here, as if it had been initialized by the compiler.
36077
36078   .sect .rom      ! Before the string table please
36079   _sizes:                         ! sizes of kernel, mm, fs filled in by boot
36080           .data2  0x526F          ! this must be the first data entry (magic #)
36081           .space  16*2*2-2        ! monitor uses previous word and this space
36082                                   ! extra space allows for additional servers
36083   .sect .bss
36084   k_stack:
36085           .space  K_STACK_BYTES   ! kernel stack
36086   k_stktop:                       ! top of kernel stack
36087           .comm   ex_number, 4
36088           .comm   trap_errno, 4
36089           .comm   old_eip, 4
36090           .comm   old_cs, 4
36091           .comm   old_eflags, 4
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/mpx88.s
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
36100   #
36101   ! This file contains the assembler startup code for Minix and the 16-bit
36102   ! interrupt handlers.  It cooperates with cstart.c to set up a good
36103   ! environment for main().
36104
36105   ! This file is part of the lowest layer of the MINIX kernel.  The other part
36106   ! is "proc.c".  The lowest layer does process switching and message handling.
36107
36108   ! Every transition to the kernel goes through this file.  Transitions are
36109   ! caused by sending/receiving messages and by most interrupts.  (RS232
36110   ! interrupts may be handled in the file "rs2.s" and then they rarely enter
36111   ! the kernel.)
36112
36113   ! Transitions to the kernel may be nested.  The initial entry may be with a
36114   ! system call, exception or hardware interrupt; reentries may only be made
36115   ! by hardware interrupts.  The count of reentries is kept in "k_reenter".
36116   ! It is important for deciding whether to switch to the kernel stack and
36117   ! for protecting the message passing code in "proc.c".
36118
36119   ! For the message passing trap, most of the machine state is saved in the
```

```
36120   ! proc table.  (Some of the registers need not be saved.)  Then the stack is
36121   ! switched to "k_stack", and interrupts are reenabled.  Finally, the system
36122   ! call handler (in C) is called.  When it returns, interrupts are disabled
36123   ! again and the code falls into the restart routine, to finish off held-up
36124   ! interrupts and run the process or task whose pointer is in "proc_ptr".
36125
36126   ! Hardware interrupt handlers do the same, except  (1) The entire state must
36127   ! be saved.  (2) There are too many handlers to do this inline, so the save
36128   ! routine is called.  A few cycles are saved by pushing the address of the
36129   ! appropiate restart routine for a return later.  (3) A stack switch is
36130   ! avoided when the stack is already switched.  (4) The (master) 8259 interrupt
36131   ! controller is reenabled centrally in save().  (5) Each interrupt handler
36132   ! masks its interrupt line using the 8259 before enabling (other unmasked)
36133   ! interrupts, and unmasks it after servicing the interrupt.  This limits the
36134   ! nest level to the number of lines and protects the handler from itself.
36135
36136   ! For communication with the boot monitor at startup time some constant
36137   ! data are compiled into the beginning of the text segment. This facilitates
36138   ! reading the data at the start of the boot process, since only the first
36139   ! sector of the file needs to be read.
36140
36141   ! Some data storage is also allocated at the end of this file. This data
36142   ! will be at the start of the data segment of the kernel and will be read
36143   ! and modified by the boot monitor before the kernel starts.
36144
36145   #include <minix/config.h>
36146   #include <minix/const.h>
36147   #include <minix/com.h>
36148   #include "const.h"
36149   #include "sconst.h"
36150   #include "protect.h"
36151
36152   ! The external entry points into this file are:
36153   ! Note: in assembly language the .define statement applied to a function name
36154   ! is loosely equivalent to a prototype in C code -- it makes it possible to
36155   ! link to an entity declared in the assembly code but does not create
36156   ! the entity.
36157
36158   .define _idle_task      ! executed when there is no work
36159   .define _int00          ! handlers for traps and exceptions
36160   .define _int01
36161   .define _int02
36162   .define _int03
36163   .define _int04
36164   .define _int05
36165   .define _int06
36166   .define _int07
36167   .define _hwint00        ! handlers for hardware interrupts
36168   .define _hwint01
36169   .define _hwint02
36170   .define _hwint03
36171   .define _hwint04
36172   .define _hwint05
36173   .define _hwint06
36174   .define _hwint07
36175   .define _hwint08
36176   .define _hwint09
36177   .define _hwint10
36178   .define _hwint11
36179   .define _hwint12
```

```
36180   .define _hwint13
36181   .define _hwint14
36182   .define _hwint15
36183   .define _restart        ! start running a task or process
36184   .define save            ! save the machine state in the proc table
36185   .define _s_call         ! process or task wants to send or receive a message
36186
36187   ! Imported functions.
36188
36189   .extern _cstart
36190   .extern _main
36191   .extern _exception
36192   .extern _interrupt
36193   .extern _sys_call
36194   .extern _unhold
36195   .extern klib_init_prot
36196   .extern real2prot
36197
36198   ! Exported variables.
36199   ! Note: when used with a variable the .define does not reserve storage,
36200   ! it makes the name externally visible so it may be linked to.
36201
36202   .define kernel_ds
36203   .define begbss
36204   .define begdata
36205   .define _sizes
36206
36207   ! Imported variables.
36208
36209   .extern kernel_cs
36210   .extern _gdt
36211   .extern _code_base
36212   .extern _data_base
36213   .extern _held_head
36214   .extern _k_reenter
36215   .extern _pc_at
36216   .extern _proc_ptr
36217   .extern _protected_mode
36218   .extern _ps_mca
36219   .extern _irq_table
36220
36221           .text
36222   !*===========================================================================*
36223   !*                              MINIX                                        *
36224   !*===========================================================================*
36225   MINIX:                          ! this is the entry point for the MINIX kernel
36226           jmp     over_kernel_ds  ! skip over the next few bytes
36227           .data2  CLICK_SHIFT     ! for the monitor: memory granularity
36228   kernel_ds:
36229           .data2  0x0024          ! boot monitor flags:  (later kernel DS)
36230                                   !       make stack, will return
36231   over_kernel_ds:
36232
36233   ! Set up a C stack frame on the monitor stack.  (The monitor sets cs and ds
36234   ! right.  The ss register still references the monitor data segment.)
36235           push    bp
36236           mov     bp, sp
36237           push    si
36238           push    di
36239           mov     cx, 4(bp)       ! monitor code segment
```

```
36240           test    cx, cx          ! nonzero if return possible
36241           jz      noret
36242           inc     _mon_return
36243   noret:  mov     _mon_ss, ss     ! save stack location for later return
36244           mov     _mon_sp, sp
36245
36246   ! Locate boot parameters, set up kernel segment registers and stack.
36247           mov     bx, 6(bp)       ! boot parameters offset
36248           mov     dx, 8(bp)       ! boot parameters length
36249           mov     ax, ds          ! kernel data
36250           mov     es, ax
36251           mov     ss, ax
36252           mov     sp, #k_stktop   ! set sp to point to the top of kernel stack
36253
36254   ! Real mode needs to get kernel DS from the code segment.  Protected mode
36255   ! needs CS in the jump back to real mode.
36256
36257     cseg  mov     kernel_cs, cs
36258     cseg  mov     kernel_ds, ds
36259
36260   ! Call C startup code to set up a proper environment to run main().
36261           push    dx
36262           push    bx
36263           push    _mon_ss
36264           push    cx
36265           push    ds
36266           push    cs
36267           call    _cstart         ! cstart(cs, ds, mcs, mds, parmoff, parmlen)
36268           add     sp, #6*2
36269
36270           cmp     _protected_mode, #0
36271           jz      nosw            ! ok to switch to protected mode?
36272
36273           call    klib_init_prot  ! initialize klib functions for protected mode
36274           call    real2prot       ! switch to protected mode
36275
36276           push    #0              ! set flags to known good state
36277           popf                    ! especially, clear nested task and int enable
36278   nosw:
36279           jmp     _main           ! main()
36280
36281
36282   !*===========================================================================*
36283   !*                              interrupt handlers                           *
36284   !*===========================================================================*
36285
36286
36287   !*===========================================================================*
36288   !*                              hwint00 - 07                                 *
36289   !*===========================================================================*
36290   ! Note this is a macro, it looks like a subroutine.
36291   #define hwint_master(irq)       \
36292           call    save                    /* save interrupted process state */;\
36293           inb     INT_CTLMASK                                             ;\
36294           orb     al, *[1<<irq]                                           ;\
36295           outb    INT_CTLMASK             /* disable the irq              */;\
36296           movb    al, *ENABLE                                             ;\
36297           outb    INT_CTL                 /* reenable master 8259         */;\
36298           sti                             /* enable interrupts            */;\
36299           mov     ax, *irq                                                ;\
```

```
36300           push    ax                      /* irq                              */;\
36301           call    @_irq_table + 2*irq     /* ax = (*irq_table[irq])(irq)      */;\
36302           pop     cx                                                          ;\
36303           cli                             /* disable interrupts               */;\
36304           test    ax, ax                  /* need to reenable irq?            */;\
36305           jz      0f                                                          ;\
36306           inb     INT_CTLMASK                                                 ;\
36307           andb    al, *~[1<<irq]                                              ;\
36308           outb    INT_CTLMASK             /* enable the irq                   */;\
36309   0:      ret                             /* restart (another) process        */
36310
36311   ! Each of these entry points is an expansion of the hwint_master macro
36312
36313   _hwint00:               ! Interrupt routine for irq 0 (the clock).
36314           hwint_master(0)
36315
36316
36317   _hwint01:               ! Interrupt routine for irq 1 (keyboard)
36318           hwint_master(1)
36319
36320
36321   _hwint02:               ! Interrupt routine for irq 2 (cascade!)
36322           hwint_master(2)
36323
36324
36325   _hwint03:               ! Interrupt routine for irq 3 (second serial)
36326           hwint_master(3)
36327
36328
36329   _hwint04:               ! Interrupt routine for irq 4 (first serial)
36330           hwint_master(4)
36331
36332
36333   _hwint05:               ! Interrupt routine for irq 5 (XT winchester)
36334           hwint_master(5)
36335
36336
36337   _hwint06:               ! Interrupt routine for irq 6 (floppy)
36338           hwint_master(6)
36339
36340
36341   _hwint07:               ! Interrupt routine for irq 7 (printer)
36342           hwint_master(7)
36343
36344
36345   !*===========================================================================*
36346   !*                           hwint08 - 15                                    *
36347   !*===========================================================================*
36348   ! Note this is a macro, it looks like a subroutine.
36349   #define hwint_slave(irq)        \
36350           call    save                    /* save interrupted process state */;\
36351           inb     INT2_CTLMASK                                                ;\
36352           orb     al, *[1<<[irq-8]]                                           ;\
36353           outb    INT2_CTLMASK            /* disable the irq                  */;\
36354           movb    al, *ENABLE                                                 ;\
36355           outb    INT_CTL                 /* reenable master 8259             */;\
36356           jmp     .+2                     /* delay                            */;\
36357           outb    INT2_CTL                /* reenable slave 8259              */;\
36358           sti                             /* enable interrupts                */;\
36359           mov     ax, *irq                                                    ;\
```

```
36360           push    ax                      /* irq                              */;\
36361           call    @_irq_table + 2*irq     /* eax = (*irq_table[irq])(irq)     */;\
36362           pop     cx                                                           ;\
36363           cli                             /* disable interrupts               */;\
36364           test    ax, ax                  /* need to reenable irq?            */;\
36365           jz      0f                                                           ;\
36366           inb     INT2_CTLMASK                                                 ;\
36367           andb    al, *~[1<<[irq-8]]                                           ;\
36368           outb    INT2_CTLMASK            /* enable the irq                   */;\
36369   0:      ret                             /* restart (another) process        */
36370
36371   ! Each of these entry points is an expansion of the hwint_slave macro
36372
36373   _hwint08:               ! Interrupt routine for irq 8 (realtime clock)
36374           hwint_slave(8)
36375
36376
36377   _hwint09:               ! Interrupt routine for irq 9 (irq 2 redirected)
36378           hwint_slave(9)
36379
36380
36381   _hwint10:               ! Interrupt routine for irq 10
36382           hwint_slave(10)
36383
36384
36385   _hwint11:               ! Interrupt routine for irq 11
36386           hwint_slave(11)
36387
36388
36389   _hwint12:               ! Interrupt routine for irq 12
36390           hwint_slave(12)
36391
36392
36393   _hwint13:               ! Interrupt routine for irq 13 (FPU exception)
36394           hwint_slave(13)
36395
36396
36397   _hwint14:               ! Interrupt routine for irq 14 (AT winchester)
36398           hwint_slave(14)
36399
36400
36401   _hwint15:               ! Interrupt routine for irq 15
36402           hwint_slave(15)
36403
36404
36405   !*===========================================================================*
36406   !*                              save                                         *
36407   !*===========================================================================*
36408   save:                           ! save the machine state in the proc table.
36409
36410   ! In protected mode a jump to p_save is patched over the following
36411   ! code during initialization.
36412
36413           cld                     ! set direction flag to a known value
36414           push    ds
36415           push    si
36416     cseg  mov     ds,kernel_ds
36417           incb    _k_reenter      ! from -1 if not reentering
36418           jnz     push_current_stack      ! stack is already kernel stack
36419
```

```
36420           mov     si,_proc_ptr
36421           mov     AXREG(si),ax
36422           mov     BXREG(si),bx
36423           mov     CXREG(si),cx
36424           mov     DXREG(si),dx
36425           pop     SIREG(si)
36426           mov     DIREG(si),di
36427           mov     BPREG(si),bp
36428           mov     ESREG(si),es
36429           pop     DSREG(si)
36430           pop     bx              ! return adr
36431           pop     PCREG(si)
36432           pop     CSREG(si)
36433           pop     PSWREG(si)
36434           mov     SPREG(si),sp
36435           mov     SSREG(si),ss
36436
36437           mov     dx,ds
36438           mov     ss,dx
36439           mov     sp,#k_stktop
36440           mov     ax,#_restart    ! build return address for interrupt handler
36441           push    ax
36442
36443   stack_switched:
36444           mov     es,dx
36445           jmp     (bx)
36446
36447   push_current_stack:
36448           push    es
36449           push    bp
36450           push    di
36451           push    dx
36452           push    cx
36453           push    bx
36454           push    ax
36455           mov     bp,sp
36456           mov     bx,18(bp)       ! get the return adr; it becomes junk on stack
36457           mov     ax,#restart1
36458           push    ax
36459           mov     dx,ss
36460           mov     ds,dx
36461           jmp     stack_switched
36462
36463
36464   !*===========================================================================*
36465   !*                              s_call                                       *
36466   !*===========================================================================*
36467   ! This is real mode version. Alternate (_p_s_call) will be used in
36468   ! protected mode
36469
36470   _s_call:                        ! System calls are vectored here.
36471                                   ! Do save routine inline for speed,
36472                                   ! but do not save ax, bx, cx, dx,
36473                                   ! since C does not require preservation,
36474                                   ! and ax returns the result code anyway.
36475                                   ! Regs bp, si, di get saved by sys_call as
36476                                   ! well, but it is impractical not to preserve
36477                                   ! them here, in case context gets switched.
36478                                   ! Some special-case code in pick_proc()
36479                                   ! could avoid this.
```

```
36480           cld                     ! set direction flag to a known value
36481           push    ds
36482           push    si
36483     cseg  mov     ds,kernel_ds
36484           incb    _k_reenter
36485           mov     si,_proc_ptr
36486           pop     SIREG(si)
36487           mov     DIREG(si),di
36488           mov     BPREG(si),bp
36489           mov     ESREG(si),es
36490           pop     DSREG(si)
36491           pop     PCREG(si)
36492           pop     CSREG(si)
36493           pop     PSWREG(si)
36494           mov     SPREG(si),sp
36495           mov     SSREG(si),ss
36496           mov     dx,ds
36497           mov     es,dx
36498           mov     ss,dx           ! interrupt handlers may not make system calls
36499           mov     sp,#k_stktop    ! so stack is not already switched
36500                                   ! end of inline save
36501                                   ! now set up parameters for C routine sys_call
36502           push    bx              ! pointer to user message
36503           push    ax              ! src/dest
36504           push    cx              ! SEND/RECEIVE/BOTH
36505           sti                     ! allow SWITCHER to be interrupted
36506           call    _sys_call       ! sys_call(function, src_dest, m_ptr)
36507                                   ! caller is now explicitly in proc_ptr
36508           mov     AXREG(si),ax    ! sys_call MUST PRESERVE si
36509           cli
36510
36511   ! Fall into code to restart proc/task running.
36512
36513
36514   !*===========================================================================*
36515   !*                              restart                                      *
36516   !*===========================================================================*
36517   _restart:
36518
36519   ! Flush any held-up interrupts.
36520   ! This reenables interrupts, so the current interrupt handler may reenter.
36521   ! This does not matter, because the current handler is about to exit and no
36522   ! other handlers can reenter since flushing is only done when k_reenter == 0.
36523
36524   ! In protected mode a jump to p_restart is patched over the following
36525   ! code during initialization.
36526
36527           cmp     _held_head,#0   ! do fast test to usually avoid function call
36528           jz      over_call_unhold
36529           call    _unhold         ! this is rare so overhead is acceptable
36530   over_call_unhold:
36531
36532           mov     si,_proc_ptr
36533           decb    _k_reenter
36534           mov     ax,AXREG(si)    ! start restoring registers from proc table
36535                                   ! could make AXREG == 0 to use lodw here
36536           mov     bx,BXREG(si)
36537           mov     cx,CXREG(si)
36538           mov     dx,DXREG(si)
36539           mov     di,DIREG(si)
```

```
36540           mov     bp,BPREG(si)
36541           mov     es,ESREG(si)
36542           mov     ss,SSREG(si)
36543           mov     sp,SPREG(si)
36544           push    PSWREG(si)      ! fake interrupt stack frame
36545           push    CSREG(si)
36546           push    PCREG(si)
36547                                   ! could put si:ds together to use
36548                                   ! lds si,SIREG(si)
36549           push    DSREG(si)
36550           mov     si,SIREG(si)
36551           pop     ds
36552           iret
36553
36554   restart1:
36555           decb    _k_reenter
36556           pop     ax
36557           pop     bx
36558           pop     cx
36559           pop     dx
36560           pop     di
36561           pop     bp
36562           pop     es
36563           pop     si
36564           pop     ds
36565           add     sp,#2           ! skip return adr
36566           iret
36567
36568
36569   !*===========================================================================*
36570   !*                              int00-07                                     *
36571   !*===========================================================================*
36572   ! These are entry points for exceptions (processor generated interrupts,
36573   ! usually caused by error conditions such as an attempt to divide by zero)
36574
36575   _int00:                         ! interrupt through vector 0
36576           push    ax
36577           movb    al,#0
36578           jmp     exception
36579
36580   _int01:                         ! interrupt through vector 1, etc
36581           push    ax
36582           movb    al,#1
36583           jmp     exception
36584
36585   _int02:
36586           push    ax
36587           movb    al,#2
36588           jmp     exception
36589
36590   _int03:
36591           push    ax
36592           movb    al,#3
36593           jmp     exception
36594
36595   _int04:
36596           push    ax
36597           movb    al,#4
36598           jmp     exception
36599
```

```
36600   _int05:
36601           push    ax
36602           movb    al,#5
36603           jmp     exception
36604
36605   _int06:
36606           push    ax
36607           movb    al,#6
36608           jmp     exception
36609
36610   _int07:
36611           push    ax
36612           movb    al,#7
36613           !jmp    exception
36614
36615   exception:
36616     cseg  movb    ex_number,al    ! it is cumbersome to get this into dseg
36617           pop     ax
36618           call    save
36619     cseg  push    ex_number       ! high byte is constant 0
36620           call    _exception      ! do whatever is necessary (sti only if safe)
36621           add     sp,#2
36622           cli
36623           ret
36624
36625
36626   !*===========================================================================*
36627   !*                              level0_call                                  *
36628   !*===========================================================================*
36629   _level0_call:
36630           call    save
36631           jmp     @_level0_func
36632
36633
36634   !*===========================================================================*
36635   !*                              idle_task                                    *
36636   !*===========================================================================*
36637   _idle_task:                     ! executed when there is no work
36638           jmp     _idle_task      ! a "hlt" before this fails in protected mode
36639
36640
36641   !*===========================================================================*
36642   !*                              data                                         *
36643   !*===========================================================================*
36644   ! NB some variables are stored in code segment.
36645
36646   ex_number:                      ! exception number
36647           .space  2
36648
36649
36650   !*===========================================================================*
36651   !*                      variants for 286 protected mode                      *
36652   !*===========================================================================*
36653
36654   ! Most routines are different in 286 protected mode.
36655   ! The only essential difference is that an interrupt in protected mode
36656   ! (usually) switches the stack, so there is less to do in software.
36657
36658   ! These functions are reached along jumps patched in by klib_init_prot():
36659
```

```
36660           .define         p_restart       ! replaces _restart
36661           .define         p_save          ! replaces save
36662
36663   ! These exception and software-interrupt handlers are enabled by the new
36664   ! interrupt vector table set up in protect.c:
36665
36666           .define         _divide_error           ! _int00
36667           .define         _single_step_exception  ! _int01
36668           .define         _nmi                    ! _int02
36669           .define         _breakpoint_exception   ! _int03
36670           .define         _overflow               ! _int04
36671           .define         _bounds_check           ! _int05
36672           .define         _inval_opcode           ! _int06
36673           .define         _copr_not_available     ! _int07
36674           .define         _double_fault           ! (286 trap)
36675           .define         _copr_seg_overrun       ! (etc)
36676           .define         _inval_tss
36677           .define         _segment_not_present
36678           .define         _stack_exception
36679           .define         _general_protection
36680           .define         _p_s_call               ! _s_call
36681           .define         _level0_call
36682
36683   ! The hardware interrupt handlers need not be altered apart from putting
36684   ! them in the new table (save() handles the differences).
36685   ! Some of the intxx handlers (those for exceptions which do not push an
36686   ! error code) need not have been replaced, but the names here are better.
36687
36688   #include "protect.h"
36689
36690   /* Selected 286 tss offsets. */
36691   #define TSS2_S_SP0      2
36692
36693   ! imported variables
36694
36695           .extern         _tss
36696           .extern         _level0_func
36697
36698   !*===========================================================================*
36699   !*                              p_save                                       *
36700   !*===========================================================================*
36701   ! Save for 286 protected mode.
36702   ! This is much simpler than for 8086 mode, because the stack already points
36703   ! into process table, or has already been switched to the kernel stack.
36704
36705   p_save:
36706           cld                     ! set direction flag to a known value
36707           pusha                   ! save "general" registers
36708           push    ds              ! save ds
36709           push    es              ! save es
36710           mov     dx,ss           ! ss is kernel data segment
36711           mov     ds,dx           ! load rest of kernel segments
36712           mov     es,dx
36713           mov     bp,sp           ! prepare to return
36714           incb    _k_reenter      ! from -1 if not reentering
36715           jnz     set_p1_restart  ! stack is already kernel stack
36716           mov     sp,#k_stktop
36717           push    #p_restart      ! build return address for interrupt handler
36718           jmp     @RETADR-P_STACKBASE(bp)
36719
```

```
36720   set_p1_restart:
36721           push    #p1_restart
36722           jmp     @RETADR-P_STACKBASE(bp)
36723
36724
36725   !*===========================================================================*
36726   !*                              p_s_call                                     *
36727   !*===========================================================================*
36728   _p_s_call:
36729           cld                     ! set direction flag to a known value
36730           sub     sp,#6*2         ! skip RETADR, ax, cx, dx, bx, st
36731           push    bp              ! stack already points into process table
36732           push    si
36733           push    di
36734           push    ds
36735           push    es
36736           mov     dx,ss
36737           mov     ds,dx
36738           mov     es,dx
36739           incb    _k_reenter
36740           mov     si,sp           ! assumes P_STACKBASE == 0
36741           mov     sp,#k_stktop
36742                                   ! end of inline save
36743           sti                     ! allow SWITCHER to be interrupted
36744                                   ! now set up parameters for C routine sys_call
36745           push    bx              ! pointer to user message
36746           push    ax              ! src/dest
36747           push    cx              ! SEND/RECEIVE/BOTH
36748           call    _sys_call       ! sys_call(function, src_dest, m_ptr)
36749                                   ! caller is now explicitly in proc_ptr
36750           mov     AXREG(si),ax    ! sys_call MUST PRESERVE si
36751           cli
36752
36753   ! Fall into code to restart proc/task running.
36754
36755   p_restart:
36756
36757   ! Flush any held-up interrupts.
36758   ! This reenables interrupts, so the current interrupt handler may reenter.
36759   ! This does not matter, because the current handler is about to exit and no
36760   ! other handlers can reenter since flushing is only done when k_reenter == 0.
36761
36762           cmp     _held_head,#0   ! do fast test to usually avoid function call
36763           jz      p_over_call_unhold
36764           call    _unhold         ! this is rare so overhead is acceptable
36765   p_over_call_unhold:
36766           mov     si,_proc_ptr
36767           lldt    P_LDT_SEL(si)           ! enable segment descriptors for task
36768           lea     ax,P_STACKTOP(si)       ! arrange for next interrupt
36769           mov     _tss+TSS2_S_SP0,ax      ! to save state in process table
36770           mov     sp,si           ! assumes P_STACKBASE == 0
36771   p1_restart:
36772           decb    _k_reenter
36773           pop     es
36774           pop     ds
36775           popa
36776           add     sp,#2           ! skip return adr
36777           iret                    ! continue process
36778
36779
```

```
36780   !*===========================================================================*
36781   !*                                exception handlers                         *
36782   !*===========================================================================*
36783   _divide_error:
36784           push    #DIVIDE_VECTOR
36785           jmp     p_exception
36786
36787   _single_step_exception:
36788           push    #DEBUG_VECTOR
36789           jmp     p_exception
36790
36791   _nmi:
36792           push    #NMI_VECTOR
36793           jmp     p_exception
36794
36795   _breakpoint_exception:
36796           push    #BREAKPOINT_VECTOR
36797           jmp     p_exception
36798
36799   _overflow:
36800           push    #OVERFLOW_VECTOR
36801           jmp     p_exception
36802
36803   _bounds_check:
36804           push    #BOUNDS_VECTOR
36805           jmp     p_exception
36806
36807   _inval_opcode:
36808           push    #INVAL_OP_VECTOR
36809           jmp     p_exception
36810
36811   _copr_not_available:
36812           push    #COPROC_NOT_VECTOR
36813           jmp     p_exception
36814
36815   _double_fault:
36816           push    #DOUBLE_FAULT_VECTOR
36817           jmp     errexception
36818
36819   _copr_seg_overrun:
36820           push    #COPROC_SEG_VECTOR
36821           jmp     p_exception
36822
36823   _inval_tss:
36824           push    #INVAL_TSS_VECTOR
36825           jmp     errexception
36826
36827   _segment_not_present:
36828           push    #SEG_NOT_VECTOR
36829           jmp     errexception
36830
36831   _stack_exception:
36832           push    #STACK_FAULT_VECTOR
36833           jmp     errexception
36834
36835   _general_protection:
36836           push    #PROTECTION_VECTOR
36837           jmp     errexception
36838
36839
```

```
36840   !*===========================================================================*
36841   !*                              p_exception                                  *
36842   !*===========================================================================*
36843   ! This is called for all exceptions which do not push an error code.
36844
36845   p_exception:
36846     sseg  pop     ds_ex_number
36847           call    p_save
36848           jmp     p1_exception
36849
36850
36851   !*===========================================================================*
36852   !*                              errexception                                 *
36853   !*===========================================================================*
36854   ! This is called for all exceptions which push an error code.
36855
36856   errexception:
36857     sseg  pop     ds_ex_number
36858     sseg  pop     trap_errno
36859           call    p_save
36860   p1_exception:                   ! Common for all exceptions.
36861           push    ds_ex_number
36862           call    _exception
36863           add     sp,#2
36864           cli
36865           ret
36866
36867
36868   !*===========================================================================*
36869   !*                              data                                         *
36870   !*===========================================================================*
36871   ! These declarations assure that storage will be allocated at the very
36872   ! beginning of the kernel data section, so the boot monitor can be easily
36873   ! told how to patch these locations. Note that the magic number is put
36874   ! here by the compiler, but will be read by, and then overwritten by,
36875   ! the boot monitor. When the kernel starts the sizes array will be
36876   ! found here, as if it had been initialized by the compiler.
36877
36878           .data
36879   begdata:
36880   _sizes:                         ! sizes of kernel, mm, fs filled in by boot
36881           .data2  0x526F          ! this must be the first data entry (magic #)
36882           .space  16*2*2-2        ! monitor uses previous 2 words and this space
36883                                   ! extra space allows for additional servers
36884           .bss
36885   begbss:
36886   k_stack:
36887           .space  K_STACK_BYTES   ! kernel stack
36888   k_stktop:                       ! top of kernel stack
36889   ds_ex_number:
36890           .space  2
36891   trap_errno:
36892           .space  2
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/keymaps/genmap.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

36900   /*      genmap - output binary keymap                   Author: Marcus Hampel
36901    */
36902   #include <sys/types.h>
36903   #include <minix/keymap.h>
36904   #include <fcntl.h>
36905   #include <unistd.h>
36906   #include <stdlib.h>
36907   #include <string.h>
36908   #include <errno.h>
36909
36910   #include KEYSRC
36911
36912   u8_t comprmap[4 + NR_SCAN_CODES * MAP_COLS * 9/8 * 2 + 1];
36913
36914   void tell(const char *s)
36915   {
36916     write(2, s, strlen(s));
36917   }

36919   int main(void)
36920   {
36921     u8_t *cm, *fb;
36922     u16_t *km;
36923     int n;
36924
36925     /* Compress the keymap. */
36926     memcpy(comprmap, KEY_MAGIC, 4);
36927     cm = comprmap + 4;
36928     n = 8;
36929     for (km = keymap; km < keymap + NR_SCAN_CODES * MAP_COLS; km++) {
36930           if (n == 8) {
36931                   /* Allocate a new flag byte. */
36932                   fb = cm;
36933                   *cm++ = 0;
36934                   n= 0;
36935           }
36936           *cm++ = (*km & 0x00FF);         /* Low byte. */
36937           if (*km & 0xFF00) {
36938                   *cm++ = (*km >> 8);     /* High byte only when set. */
36939                   *fb |= (1 << n);        /* Set a flag if so. */
36940           }
36941           n++;
36942     }
36943
36944     /* Don't store trailing zeros. */
36945     while (cm > comprmap && cm[-1] == 0) cm--;
36946
36947     /* Emit the compressed keymap. */
36948     if (write(1, comprmap, cm - comprmap) < 0) {
36949           int err = errno;
36950
36951           tell("genmap: ");
36952           tell(strerror(err));
36953           tell("\n");
36954           exit(1);
```

```
36955       }
36956       exit(0);
36957     }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/keymaps/french.src
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

37000     /* Keymap for the French keyboard. */
37001
37002     u16_t keymap[NR_SCAN_CODES * MAP_COLS] = {
37003
37004     /* scan-code            !Shift  Shift   Alt     AltGr   Alt+Sh  Ctrl    */
37005     /* ========================================================================= */
37006     /* 00 - none     */      0,      0,      0,      0,      0,      0,
37007     /* 01 - ESC      */      C('['), C('['), CA('['),C('['), C('['), C('['),
37008     /* 02 - '1'      */      '&',    '1',    A('1'), '&',    '1',    C('A'),
37009     /* 03 - '2'      */      0202,   '2',    A('2'), '~',    '2',    C('B'),
37010     /* 04 - '3'      */      '"',    '3',    A('3'), '#',    '3',    C('C'),
37011     /* 05 - '4'      */      '\'',   '4',    A('4'), '{',    '4',    C('D'),
37012     /* 06 - '5'      */      '(',    '5',    A('5'), '[',    '5',    C('E'),
37013     /* 07 - '6'      */      '-',    '6',    A('6'), '|',    '6',    C('F'),
37014     /* 08 - '7'      */      0212,   '7',    A('7'), '´',    '7',    C('G'),
37015     /* 09 - '8'      */      '_',    '8',    A('8'), '\\',   '8',    C('H'),
37016     /* 10 - '9'      */      0207,   '9',    A('9'), '^',    '9',    C('I'),
37017     /* 11 - '0'      */      0205,   '0',    A('0'), '@',    '0',    C('J'),
37018     /* 12 - '-'      */      ')',    0370,   A(')'), ']',    '-',    C('K'),
37019     /* 13 - '='      */      '=',    '+',    A('='), '}',    '=',    C('L'),
37020     /* 14 - BS       */      C('H'), C('H'), CA('H'),C('H'), C('H'), 0177,
37021     /* 15 - TAB      */      C('I'), C('I'), CA('I'),C('I'), C('I'), C('I'),
37022     /* 16 - 'q'      */      L('a'), 'A',    A('a'), 'a',    'Q',    C('A'),
37023     /* 17 - 'w'      */      L('z'), 'Z',    A('z'), 'z',    'W',    C('Z'),
37024     /* 18 - 'e'      */      L('e'), 'E',    A('e'), 'e',    'E',    C('E'),
37025     /* 19 - 'r'      */      L('r'), 'R',    A('r'), 'r',    'R',    C('R'),
37026     /* 20 - 't'      */      L('t'), 'T',    A('t'), 't',    'T',    C('T'),
37027     /* 21 - 'y'      */      L('y'), 'Y',    A('y'), 'y',    'Y',    C('Y'),
37028     /* 22 - 'u'      */      L('u'), 'U',    A('u'), 'u',    'U',    C('U'),
37029     /* 23 - 'i'      */      L('i'), 'I',    A('i'), 'i',    'I',    C('I'),
37030     /* 24 - 'o'      */      L('o'), 'O',    A('o'), 'o',    'O',    C('O'),
37031     /* 25 - 'p'      */      L('p'), 'P',    A('p'), 'p',    'P',    C('P'),
37032     /* 26 - '['      */      '^',    '"',    A('^'), '^',    '[',    C('^'),
37033     /* 27 - ']'      */      '$',    0234,   A('$'), '$',    ']',    C('$'),
37034     /* 28 - CR/LF    */      C('M'), C('M'), CA('M'),C('M'), C('M'), C('J'),
37035     /* 29 - Ctrl     */      CTRL,   CTRL,   CTRL,   CTRL,   CTRL,   CTRL,
37036     /* 30 - 'a'      */      L('q'), 'Q',    A('q'), 'q',    'A',    C('Q'),
37037     /* 31 - 's'      */      L('s'), 'S',    A('s'), 's',    'S',    C('S'),
37038     /* 32 - 'd'      */      L('d'), 'D',    A('d'), 'd',    'D',    C('D'),
37039     /* 33 - 'f'      */      L('f'), 'F',    A('f'), 'f',    'F',    C('F'),
37040     /* 34 - 'g'      */      L('g'), 'G',    A('g'), 'g',    'G',    C('G'),
37041     /* 35 - 'h'      */      L('h'), 'H',    A('h'), 'h',    'H',    C('H'),
37042     /* 36 - 'j'      */      L('j'), 'J',    A('j'), 'j',    'J',    C('J'),
37043     /* 37 - 'k'      */      L('k'), 'K',    A('k'), 'k',    'K',    C('K'),
37044     /* 38 - 'l'      */      L('l'), 'L',    A('l'), 'l',    'L',    C('L'),
37045     /* 39 - ';'      */      L('m'), 'M',    A('m'), 'm',    'M',    C('M'),
37046     /* 40 - '\''     */      0227,   '%',    A('%'), 0227,   '\\',   C('G'),
37047     /* 41 - '´       */    0375,   0375,   0375,   0375,   '´',    C('['),
37048     /* 42 - l. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
37049     /* 43 - '´       */    '*',    0346,   A('*'), '*',    '´',    C('*'),
```

```
37050   /* 44 - 'z'     */      L('w'), 'W',    A('w'), 'w',    'Z',    C('W'),
37051   /* 45 - 'x'     */      L('x'), 'X',    A('x'), 'x',    'X',    C('X'),
37052   /* 46 - 'c'     */      L('c'), 'C',    A('c'), 'c',    'C',    C('C'),
37053   /* 47 - 'v'     */      L('v'), 'V',    A('v'), 'v',    'V',    C('V'),
37054   /* 48 - 'b'     */      L('b'), 'B',    A('b'), 'b',    'B',    C('B'),
37055   /* 49 - 'n'     */      L('n'), 'N',    A('n'), 'n',    'N',    C('N'),
37056   /* 50 - 'm'     */      ',',    '?',    A(','), ',',    'm',    C('@'),
37057   /* 51 - ','     */      ';',    '.',    A(';'), ';',    ',',    C('@'),
37058   /* 52 - '.'     */      ':',    '/',    A(':'), ':',    '.',    C('@'),
37059   /* 53 - '/'     */      '!', '$'/*025*/,A('!'), '!',    '/',    C('@'),
37060   /* 54 - r. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
37061   /* 55 - '*'     */      '*',    '*',    A('*'), '*',    '*',    C('@'),
37062   /* 56 - ALT     */      ALT,    ALT,    ALT,    ALT,    ALT,    ALT,
37063   /* 57 - ' '     */      ' ',    ' ',    A(' '), ' ',    ' ',    C('@'),
37064   /* 58 - CapsLck */      CALOCK, CALOCK, CALOCK, CALOCK, CALOCK, CALOCK,
37065   /* 59 - F1      */      F1,     SF1,    AF1,    AF1,    ASF1,   CF1,
37066   /* 60 - F2      */      F2,     SF2,    AF2,    AF2,    ASF2,   CF2,
37067   /* 61 - F3      */      F3,     SF3,    AF3,    AF3,    ASF3,   CF3,
37068   /* 62 - F4      */      F4,     SF4,    AF4,    AF4,    ASF4,   CF4,
37069   /* 63 - F5      */      F5,     SF5,    AF5,    AF5,    ASF5,   CF5,
37070   /* 64 - F6      */      F6,     SF6,    AF6,    AF6,    ASF6,   CF6,
37071   /* 65 - F7      */      F7,     SF7,    AF7,    AF7,    ASF7,   CF7,
37072   /* 66 - F8      */      F8,     SF8,    AF8,    AF8,    ASF8,   CF8,
37073   /* 67 - F9      */      F9,     SF9,    AF9,    AF9,    ASF9,   CF9,
37074   /* 68 - F10     */      F10,    SF10,   AF10,   AF10,   ASF10,  CF10,
37075   /* 69 - NumLock */      NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,
37076   /* 70 - ScrLock */      SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,
37077   /* 71 - Home    */      HOME,   '7',    AHOME,  AHOME,  '7',    CHOME,
37078   /* 72 - CurUp   */      UP,     '8',    AUP,    AUP,    '8',    CUP,
37079   /* 73 - PgUp    */      PGUP,   '9',    APGUP,  APGUP,  '9',    CPGUP,
37080   /* 74 - '-'     */      NMIN,   '-',    ANMIN,  ANMIN,  '-',    CNMIN,
37081   /* 75 - Left    */      LEFT,   '4',    ALEFT,  ALEFT,  '4',    CLEFT,
37082   /* 76 - MID     */      MID,    '5',    AMID,   AMID,   '5',    CMID,
37083   /* 77 - Right   */      RIGHT,  '6',    ARIGHT, ARIGHT, '6',    CRIGHT,
37084   /* 78 - '+'     */      PLUS,   '+',    APLUS,  APLUS,  '+',    CPLUS,
37085   /* 79 - End     */      END,    '1',    AEND,   AEND,   '1',    CEND,
37086   /* 80 - Down    */      DOWN,   '2',    ADOWN,  ADOWN,  '2',    CDOWN,
37087   /* 81 - PgDown  */      PGDN,   '3',    APGDN,  APGDN,  '3',    CPGDN,
37088   /* 82 - Insert  */      INSRT,  '0',    AINSRT, AINSRT, '0',    CINSRT,
37089   /* 83 - Delete  */      0177,   '.',    A(0177),0177,   '.',    0177,
37090   /* 84 - Enter   */      C('M'), C('M'), CA('M'),C('M'), C('M'), C('J'),
37091   /* 85 - ???     */      0,      0,      0,      0,      0,      0,
37092   /* 86 - ???     */      '<',    '>',    A('<'), '<',    '>',    C('@'),
37093   /* 87 - F11     */      F11,    SF11,   AF11,   AF11,   ASF11,  CF11,
37094   /* 88 - F12     */      F12,    SF12,   AF12,   AF12,   ASF12,  CF12,
37095   /* 89 - ???     */      0,      0,      0,      0,      0,      0,
37096   /* 90 - ???     */      0,      0,      0,      0,      0,      0,
37097   /* 91 - ???     */      0,      0,      0,      0,      0,      0,
37098   /* 92 - ???     */      0,      0,      0,      0,      0,      0,
37099   /* 93 - ???     */      0,      0,      0,      0,      0,      0,
37100   /* 94 - ???     */      0,      0,      0,      0,      0,      0,
37101   /* 95 - ???     */      0,      0,      0,      0,      0,      0,
37102   /* 96 - EXT_KEY */      EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY,
37103   /* 97 - ???     */      0,      0,      0,      0,      0,      0,
37104   /* 98 - ???     */      0,      0,      0,      0,      0,      0,
37105   /* 99 - ???     */      0,      0,      0,      0,      0,      0,
37106   /*100 - ???     */      0,      0,      0,      0,      0,      0,
37107   /*101 - ???     */      0,      0,      0,      0,      0,      0,
37108   /*102 - ???     */      0,      0,      0,      0,      0,      0,
37109   /*103 - ???     */      0,      0,      0,      0,      0,      0,
```

```
37110    /*104 - ???     */      0,      0,      0,      0,      0,      0,
37111    /*105 - ???     */      0,      0,      0,      0,      0,      0,
37112    /*106 - ???     */      0,      0,      0,      0,      0,      0,
37113    /*107 - ???     */      0,      0,      0,      0,      0,      0,
37114    /*108 - ???     */      0,      0,      0,      0,      0,      0,
37115    /*109 - ???     */      0,      0,      0,      0,      0,      0,
37116    /*110 - ???     */      0,      0,      0,      0,      0,      0,
37117    /*111 - ???     */      0,      0,      0,      0,      0,      0,
37118    /*112 - ???     */      0,      0,      0,      0,      0,      0,
37119    /*113 - ???     */      0,      0,      0,      0,      0,      0,
37120    /*114 - ???     */      0,      0,      0,      0,      0,      0,
37121    /*115 - ???     */      0,      0,      0,      0,      0,      0,
37122    /*116 - ???     */      0,      0,      0,      0,      0,      0,
37123    /*117 - ???     */      0,      0,      0,      0,      0,      0,
37124    /*118 - ???     */      0,      0,      0,      0,      0,      0,
37125    /*119 - ???     */      0,      0,      0,      0,      0,      0,
37126    /*120 - ???     */      0,      0,      0,      0,      0,      0,
37127    /*121 - ???     */      0,      0,      0,      0,      0,      0,
37128    /*122 - ???     */      0,      0,      0,      0,      0,      0,
37129    /*123 - ???     */      0,      0,      0,      0,      0,      0,
37130    /*124 - ???     */      0,      0,      0,      0,      0,      0,
37131    /*125 - ???     */      0,      0,      0,      0,      0,      0,
37132    /*126 - ???     */      0,      0,      0,      0,      0,      0,
37133    /*127 - ???     */      0,      0,      0,      0,      0,      0
37134    };
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/keymaps/german.src
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

37200    /* Keymap for German MF-2 keyboard. */
37201
37202    u16_t keymap[NR_SCAN_CODES * MAP_COLS] = {
37203
37204    /* scan-code            unsh    Shift   Alt     AltGr   Alt+Sh  Strg    */
37205    /* ============================================================== */
37206    /* 00 - none    */      0,      0,      0,      0,      0,      0,
37207    /* 01 - ESC     */      C('['), C('['), CA('['),C('['), C('['), C('['),
37208    /* 02 - '1'     */      '1',    '!',    A('1'), '1',    '!',    C('A'),
37209    /* 03 - '2'     */      '2',    '"',    A('2'), 0375,   '@',    C('@'),
37210    /* 04 - '3'     */      '3',    025,    A('3'), 0374,   '#',    C('C'),
37211    /* 05 - '4'     */      '4',    '$',    A('4'), '4',    '$',    C('D'),
37212    /* 06 - '5'     */      '5',    '%',    A('5'), '5',    '%',    C('E'),
37213    /* 07 - '6'     */      '6',    '&',    A('6'), '6',    '^',    C('^'),
37214    /* 08 - '7'     */      '7',    '/',    A('7'), '{',    '&',    C('G'),
37215    /* 09 - '8'     */      '8',    '(',    A('8'), '[',    '*',    C('H'),
37216    /* 10 - '9'     */      '9',    ')',    A('9'), ']',    '(',    C('I'),
37217    /* 11 - '0'     */      '0',    '=',    A('0'), '}',    ')',    C('@'),
37218    /* 12 - '-'     */      0341,   '?',    0341,   '\\',   '_',    C('_'),
37219    /* 13 - '='     */      '\'',   '´',    A('\''),'=',    '+',    C('@'),
37220    /* 14 - BS      */      C('H'), C('H'), CA('H'),C('H'), C('H'), 0177,
37221    /* 15 - TAB     */      C('I'), C('I'), CA('I'),C('I'), C('I'), C('I'),
37222    /* 16 - 'q'     */      L('q'), 'Q',    A('q'), '@',    'Q',    C('Q'),
37223    /* 17 - 'w'     */      L('w'), 'W',    A('w'), 'w',    'W',    C('W'),
37224    /* 18 - 'e'     */      L('e'), 'E',    A('e'), 'e',    'E',    C('E'),
```

```
37225    /* 19 - 'r'     */      L('r'), 'R',    A('r'), 'r',    'R',    C('R'),
37226    /* 20 - 't'     */      L('t'), 'T',    A('t'), 't',    'T',    C('T'),
37227    /* 21 - 'y'     */      L('z'), 'Z',    A('z'), 'z',    'Z',    C('Z'),
37228    /* 22 - 'u'     */      L('u'), 'U',    A('u'), 'u',    'U',    C('U'),
37229    /* 23 - 'i'     */      L('i'), 'I',    A('i'), 'i',    'I',    C('I'),
37230    /* 24 - 'o'     */      L('o'), 'O',    A('o'), 'o',    'O',    C('O'),
37231    /* 25 - 'p'     */      L('p'), 'P',    A('p'), 'p',    'P',    C('P'),
37232    /* 26 - '['     */      L(0201),0232,   0201,   '[',    '{',    C('['),
37233    /* 27 - ']'     */      '+',    '*',    A('+'), '~',    ']',    C(']'),
37234    /* 28 - CR/LF   */      C('M'), C('M'), CA('M'),C('M'), C('M'), C('J'),
37235    /* 29 - Strg;-) */      CTRL,   CTRL,   CTRL,   CTRL,   CTRL,   CTRL,
37236    /* 30 - 'a'     */      L('a'), 'A',    A('a'), 'a',    'A',    C('A'),
37237    /* 31 - 's'     */      L('s'), 'S',    A('s'), 's',    'S',    C('S'),
37238    /* 32 - 'd'     */      L('d'), 'D',    A('d'), 'd',    'D',    C('D'),
37239    /* 33 - 'f'     */      L('f'), 'F',    A('f'), 'f',    'F',    C('F'),
37240    /* 34 - 'g'     */      L('g'), 'G',    A('g'), 'g',    'G',    C('G'),
37241    /* 35 - 'h'     */      L('h'), 'H',    A('h'), 'h',    'H',    C('H'),
37242    /* 36 - 'j'     */      L('j'), 'J',    A('j'), 'j',    'J',    C('J'),
37243    /* 37 - 'k'     */      L('k'), 'K',    A('k'), 'k',    'K',    C('K'),
37244    /* 38 - 'l'     */      L('l'), 'L',    A('l'), 'l',    'L',    C('L'),
37245    /* 39 - ';'     */      L(0224),0231,   0224,   ';',    ':',    C('@'),
37246    /* 40 - '\''    */      L(0204),0216,   0204,   '\'',   '"',    C('@'),
37247    /* 41 - '´      */     '^',    0370,   A('^'), '´',    '~',    C('^'),
37248    /* 42 - SHIFT   */      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
37249    /* 43 - '\\'    */      '#',    '\'',   A('#'), '\\',   '|',    C('\\'),
37250    /* 44 - 'z'     */      L('y'), 'Y',    A('y'), 'y',    'Y',    C('Y'),
37251    /* 45 - 'x'     */      L('x'), 'X',    A('x'), 'x',    'X',    C('X'),
37252    /* 46 - 'c'     */      L('c'), 'C',    A('c'), 'c',    'C',    C('C'),
37253    /* 47 - 'v'     */      L('v'), 'V',    A('v'), 'v',    'V',    C('V'),
37254    /* 48 - 'b'     */      L('b'), 'B',    A('b'), 'b',    'B',    C('B'),
37255    /* 49 - 'n'     */      L('n'), 'N',    A('n'), 'n',    'N',    C('N'),
37256    /* 50 - 'm'     */      L('m'), 'M',    A('m'), 0346,   'M',    C('M'),
37257    /* 51 - ','     */      ',',    ';',    A(','), ',',    '<',    C('@'),
37258    /* 52 - '.'     */      '.',    ':',    A('.'), '.',    '>',    C('@'),
37259    /* 53 - '/'     */      '-',    '_',    A('-'), '/',    '?',    C('_'),
37260    /* 54 - SHIFT   */      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
37261    /* 55 - '*'     */      '*',    '*',    A('*'), '*',    '*',    C('@'),
37262    /* 56 - ALT     */      ALT,    ALT,    ALT,    ALT,    ALT,    ALT,
37263    /* 57 - ' '     */      ' ',    ' ',    A(' '), ' ',    ' ',    C('@'),
37264    /* 58 - CapsLck */      CALOCK, CALOCK, CALOCK, CALOCK, CALOCK, CALOCK,
37265    /* 59 - F1      */      F1,     SF1,    AF1,    AF1,    ASF1,   CF1,
37266    /* 60 - F2      */      F2,     SF2,    AF2,    AF2,    ASF2,   CF2,
37267    /* 61 - F3      */      F3,     SF3,    AF3,    AF3,    ASF3,   CF3,
37268    /* 62 - F4      */      F4,     SF4,    AF4,    AF4,    ASF4,   CF4,
37269    /* 63 - F5      */      F5,     SF5,    AF5,    AF5,    ASF5,   CF5,
37270    /* 64 - F6      */      F6,     SF6,    AF6,    AF6,    ASF6,   CF6,
37271    /* 65 - F7      */      F7,     SF7,    AF7,    AF7,    ASF7,   CF7,
37272    /* 66 - F8      */      F8,     SF8,    AF8,    AF8,    ASF8,   CF8,
37273    /* 67 - F9      */      F9,     SF9,    AF9,    AF9,    ASF9,   CF9,
37274    /* 68 - F10     */      F10,    SF10,   AF10,   AF10,   ASF10,  CF10,
37275    /* 69 - NumLock */      NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,
37276    /* 70 - ScrLock */      SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,
37277    /* 71 - Home    */      HOME,   '7',    AHOME,  AHOME,  '7',    CHOME,
37278    /* 72 - CurUp   */      UP,     '8',    AUP,    AUP,    '8',    CUP,
37279    /* 73 - PgUp    */      PGUP,   '9',    APGUP,  APGUP,  '9',    CPGUP,
37280    /* 74 - '-'     */      NMIN,   '-',    ANMIN,  ANMIN,  '-',    CNMIN,
37281    /* 75 - Left    */      LEFT,   '4',    ALEFT,  ALEFT,  '4',    CLEFT,
37282    /* 76 - MID     */      MID,    '5',    AMID,   AMID,   '5',    CMID,
37283    /* 77 - Right   */      RIGHT,  '6',    ARIGHT, ARIGHT, '6',    CRIGHT,
37284    /* 78 - '+'     */      PLUS,   '+',    APLUS,  APLUS,  '+',    CPLUS,
```

```
37285   /* 79 - End     */      END,    '1',    AEND,   AEND,   '1',    CEND,
37286   /* 80 - Down    */      DOWN,   '2',    ADOWN,  ADOWN,  '2',    CDOWN,
37287   /* 81 - PgDown  */      PGDN,   '3',    APGDN,  APGDN,  '3',    CPGDN,
37288   /* 82 - Insert  */      INSRT,  '0',    AINSRT, AINSRT, '0',    CINSRT,
37289   /* 83 - Delete  */      0177,   '.',    A(0177),0177,   '.',    0177,
37290   /* 84 - Enter   */      C('M'), C('M'), CA('M'),C('M'), C('M'), C('J'),
37291   /* 85 - ???     */      0,      0,      0,      0,      0,      0,
37292   /* 86 - ???     */      '<',    '>',    A('<'), '|',    '>',    C('@'),
37293   /* 87 - F11     */      F11,    SF11,   AF11,   AF11,   ASF11,  CF11,
37294   /* 88 - F12     */      F12,    SF12,   AF12,   AF12,   ASF12,  CF12,
37295   /* 89 - ???     */      0,      0,      0,      0,      0,      0,
37296   /* 90 - ???     */      0,      0,      0,      0,      0,      0,
37297   /* 91 - ???     */      0,      0,      0,      0,      0,      0,
37298   /* 92 - ???     */      0,      0,      0,      0,      0,      0,
37299   /* 93 - ???     */      0,      0,      0,      0,      0,      0,
37300   /* 94 - ???     */      0,      0,      0,      0,      0,      0,
37301   /* 95 - ???     */      0,      0,      0,      0,      0,      0,
37302   /* 96 - EXT_KEY */      EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY,
37303   /* 97 - ???     */      0,      0,      0,      0,      0,      0,
37304   /* 98 - ???     */      0,      0,      0,      0,      0,      0,
37305   /* 99 - ???     */      0,      0,      0,      0,      0,      0,
37306   /*100 - ???     */      0,      0,      0,      0,      0,      0,
37307   /*101 - ???     */      0,      0,      0,      0,      0,      0,
37308   /*102 - ???     */      0,      0,      0,      0,      0,      0,
37309   /*103 - ???     */      0,      0,      0,      0,      0,      0,
37310   /*104 - ???     */      0,      0,      0,      0,      0,      0,
37311   /*105 - ???     */      0,      0,      0,      0,      0,      0,
37312   /*106 - ???     */      0,      0,      0,      0,      0,      0,
37313   /*107 - ???     */      0,      0,      0,      0,      0,      0,
37314   /*108 - ???     */      0,      0,      0,      0,      0,      0,
37315   /*109 - ???     */      0,      0,      0,      0,      0,      0,
37316   /*110 - ???     */      0,      0,      0,      0,      0,      0,
37317   /*111 - ???     */      0,      0,      0,      0,      0,      0,
37318   /*112 - ???     */      0,      0,      0,      0,      0,      0,
37319   /*113 - ???     */      0,      0,      0,      0,      0,      0,
37320   /*114 - ???     */      0,      0,      0,      0,      0,      0,
37321   /*115 - ???     */      0,      0,      0,      0,      0,      0,
37322   /*116 - ???     */      0,      0,      0,      0,      0,      0,
37323   /*117 - ???     */      0,      0,      0,      0,      0,      0,
37324   /*118 - ???     */      0,      0,      0,      0,      0,      0,
37325   /*119 - ???     */      0,      0,      0,      0,      0,      0,
37326   /*120 - ???     */      0,      0,      0,      0,      0,      0,
37327   /*121 - ???     */      0,      0,      0,      0,      0,      0,
37328   /*122 - ???     */      0,      0,      0,      0,      0,      0,
37329   /*123 - ???     */      0,      0,      0,      0,      0,      0,
37330   /*124 - ???     */      0,      0,      0,      0,      0,      0,
37331   /*125 - ???     */      0,      0,      0,      0,      0,      0,
37332   /*126 - ???     */      0,      0,      0,      0,      0,      0,
37333   /*127 - ???     */      0,      0,      0,      0,      0,      0
37334   };
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/keymaps/italian.src
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

37400    /*unchecked!*/
37401    /* Keymap for Italian MF-2 keyboard. */
37402
37403    u16_t keymap[NR_SCAN_CODES * MAP_COLS] = {
37404
37405    /* scan-code            !Shift  Shift   Alt     AltGr   Alt+Sh  Ctrl    */
37406    /* ============================================================== */
37407    /* 00 - none    */      0,      0,      0,      0,      0,      0,
37408    /* 01 - ESC     */      C('['), C('['), CA('['),C('['), C('['), C('['),
37409    /* 02 - '1'     */      '1',    '!',    A('1'), '1',    '!',    C('A'),
37410    /* 03 - '2'     */      '2',    '"',    A('2'), '2',    '@',    C('@'),
37411    /* 04 - '3'     */      '3',    0234,   A('3'), '3',    0234,   C('C'),
37412    /* 05 - '4'     */      '4',    '$',    A('4'), '4',    '$',    C('D'),
37413    /* 06 - '5'     */      '5',    '%',    A('5'), '5',    '%',    C('E'),
37414    /* 07 - '6'     */      '6',    '&',    A('6'), '6',    '&',    C('F'),
37415    /* 08 - '7'     */      '7',    '/',    A('7'), '7',    '/',    C('G'),
37416    /* 09 - '8'     */      '8',    '(',    A('8'), '8',    '(',    C('H'),
37417    /* 10 - '9'     */      '9',    ')',    A('9'), '8',    ')',    C('I'),
37418    /* 11 - '0'     */      '0',    '=',    A('0'), '0',    '=',    C('@'),
37419    /* 12 - '-'     */      '\'',   '?',    A('\''),'\'',   '?',    C('@'),
37420    /* 13 - '='     */      '|',    '^',    A('|'), '|',    '^',    C('^'),
37421    /* 14 - BS      */      C('H'), C('H'), CA('H'),C('H'), C('H'), 0177,
37422    /* 15 - TAB     */      C('I'), C('I'), CA('I'),C('I'), C('I'), C('I'),
37423    /* 16 - 'q'     */      L('q'), 'Q',    A('q'), 'q',    'Q',    C('Q'),
37424    /* 17 - 'w'     */      L('w'), 'W',    A('w'), 'w',    'W',    C('W'),
37425    /* 18 - 'e'     */      L('e'), 'E',    A('e'), 'e',    'E',    C('E'),
37426    /* 19 - 'r'     */      L('r'), 'R',    A('r'), 'r',    'R',    C('R'),
37427    /* 20 - 't'     */      L('t'), 'T',    A('t'), 't',    'T',    C('T'),
37428    /* 21 - 'y'     */      L('y'), 'Y',    A('y'), 'y',    'Y',    C('Y'),
37429    /* 22 - 'u'     */      L('u'), 'U',    A('u'), 'u',    'U',    C('U'),
37430    /* 23 - 'i'     */      L('i'), 'I',    A('i'), 'i',    'I',    C('I'),
37431    /* 24 - 'o'     */      L('o'), 'O',    A('o'), 'o',    'O',    C('O'),
37432    /* 25 - 'p'     */      L('p'), 'P',    A('p'), 'p',    'P',    C('P'),
37433    /* 26 - '['     */      0212,   0202,   0212,   '[',    '{',    C('['),
37434    /* 27 - ']'     */      '+',    '*',    A('+'), ']',    '}',    C(']'),
37435    /* 28 - CR/LF   */      C('M'), C('M'), CA('M'),C('M'), C('M'), C('J'),
37436    /* 29 - Ctrl    */      CTRL,   CTRL,   CTRL,   CTRL,   CTRL,   CTRL,
37437    /* 30 - 'a'     */      L('a'), 'A',    A('a'), 'a',    'A',    C('A'),
37438    /* 31 - 's'     */      L('s'), 'S',    A('s'), 's',    'S',    C('S'),
37439    /* 32 - 'd'     */      L('d'), 'D',    A('d'), 'd',    'D',    C('D'),
37440    /* 33 - 'f'     */      L('f'), 'F',    A('f'), 'f',    'F',    C('F'),
37441    /* 34 - 'g'     */      L('g'), 'G',    A('g'), 'g',    'G',    C('G'),
37442    /* 35 - 'h'     */      L('h'), 'H',    A('h'), 'h',    'H',    C('H'),
37443    /* 36 - 'j'     */      L('j'), 'J',    A('j'), 'j',    'J',    C('J'),
37444    /* 37 - 'k'     */      L('k'), 'K',    A('k'), 'k',    'K',    C('K'),
37445    /* 38 - 'l'     */      L('l'), 'L',    A('l'), 'l',    'L',    C('L'),
37446    /* 39 - ';'     */      0225,   '@',    0225,   0225,   '@',    C('@'),
37447    /* 40 - '\''    */      0205,   '#',    0205,   0205,   '#',    C('@'),
37448    /* 41 - '´     */      '<',    '>',    A('<'), '\\',   '|',    C('\\'),
37449    /* 42 - l. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
37450    /* 43 - '\\'    */      0227,   025,    0227,   0227,   025,    C('@'),
37451    /* 44 - 'z'     */      L('z'), 'Z',    A('z'), 'z',    'Z',    C('Z'),
37452    /* 45 - 'x'     */      L('x'), 'X',    A('x'), 'x',    'X',    C('X'),
37453    /* 46 - 'c'     */      L('c'), 'C',    A('c'), 'c',    'C',    C('C'),
37454    /* 47 - 'v'     */      L('v'), 'V',    A('v'), 'v',    'V',    C('V'),
```

```
37455   /* 48 - 'b'     */      L('b'), 'B',    A('b'), 'b',    'B',    C('B'),
37456   /* 49 - 'n'     */      L('n'), 'N',    A('n'), 'n',    'N',    C('N'),
37457   /* 50 - 'm'     */      L('m'), 'M',    A('m'), 'm',    'M',    C('M'),
37458   /* 51 - ','     */      ',',    ';',    A(','), ',',    ';',    C('@'),
37459   /* 52 - '.'     */      '.',    ':',    A('.'), '.',    ':',    C('@'),
37460   /* 53 - '/'     */      '-',    '_',    A('-'), '-',    '_',    C('_'),
37461   /* 54 - r. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
37462   /* 55 - '*'     */      '*',    '*',    A('*'), '*',    '*',    C('M'),
37463   /* 56 - ALT     */      ALT,    ALT,    ALT,    ALT,    ALT,    ALT,
37464   /* 57 - ' '     */      ' ',    ' ',    A(' '), ' ',    ' ',    C('@'),
37465   /* 58 - CapsLck */      CALOCK, CALOCK, CALOCK, CALOCK, CALOCK, CALOCK,
37466   /* 59 - F1      */      F1,     SF1,    AF1,    AF1,    ASF1,   CF1,
37467   /* 60 - F2      */      F2,     SF2,    AF2,    AF2,    ASF2,   CF2,
37468   /* 61 - F3      */      F3,     SF3,    AF3,    AF3,    ASF3,   CF3,
37469   /* 62 - F4      */      F4,     SF4,    AF4,    AF4,    ASF4,   CF4,
37470   /* 63 - F5      */      F5,     SF5,    AF5,    AF5,    ASF5,   CF5,
37471   /* 64 - F6      */      F6,     SF6,    AF6,    AF6,    ASF6,   CF6,
37472   /* 65 - F7      */      F7,     SF7,    AF7,    AF7,    ASF7,   CF7,
37473   /* 66 - F8      */      F8,     SF8,    AF8,    AF8,    ASF8,   CF8,
37474   /* 67 - F9      */      F9,     SF9,    AF9,    AF9,    ASF9,   CF9,
37475   /* 68 - F10     */      F10,    SF10,   AF10,   AF10,   ASF10,  CF10,
37476   /* 69 - NumLock */      NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,
37477   /* 70 - ScrLock */      SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,
37478   /* 71 - Home    */      HOME,   '7',    AHOME,  AHOME,  '7',    CHOME,
37479   /* 72 - CurUp   */      UP,     '8',    AUP,    AUP,    '8',    CUP,
37480   /* 73 - PgUp    */      PGUP,   '9',    APGUP,  APGUP,  '9',    CPGUP,
37481   /* 74 - '-'     */      NMIN,   '-',    ANMIN,  ANMIN,  '-',    CNMIN,
37482   /* 75 - Left    */      LEFT,   '4',    ALEFT,  ALEFT,  '4',    CLEFT,
37483   /* 76 - MID     */      MID,    '5',    AMID,   AMID,   '5',    CMID,
37484   /* 77 - Right   */      RIGHT,  '6',    ARIGHT, ARIGHT, '6',    CRIGHT,
37485   /* 78 - '+'     */      PLUS,   '+',    APLUS,  APLUS,  '+',    CPLUS,
37486   /* 79 - End     */      END,    '1',    AEND,   AEND,   '1',    CEND,
37487   /* 80 - Down    */      DOWN,   '2',    ADOWN,  ADOWN,  '2',    CDOWN,
37488   /* 81 - PgDown  */      PGDN,   '3',    APGDN,  APGDN,  '3',    CPGDN,
37489   /* 82 - Insert  */      INSRT,  '0',    AINSRT, AINSRT, '0',    CINSRT,
37490   /* 83 - Delete  */      0177,   '.',    A(0177),0177,   '.',    0177,
37491   /* 84 - Enter   */      C('M'), C('M'), CA('M'),C('M'), C('M'), C('J'),
37492   /* 85 - ???     */      0,      0,      0,      0,      0,      0,
37493   /* 86 - ???     */      '<',    '>',    A('<'), '|',    '>',    C('@'),
37494   /* 87 - F11     */      F11,    SF11,   AF11,   AF11,   ASF11,  CF11,
37495   /* 88 - F12     */      F12,    SF12,   AF12,   AF12,   ASF12,  CF12,
37496   /* 89 - ???     */      0,      0,      0,      0,      0,      0,
37497   /* 90 - ???     */      0,      0,      0,      0,      0,      0,
37498   /* 91 - ???     */      0,      0,      0,      0,      0,      0,
37499   /* 92 - ???     */      0,      0,      0,      0,      0,      0,
37500   /* 93 - ???     */      0,      0,      0,      0,      0,      0,
37501   /* 94 - ???     */      0,      0,      0,      0,      0,      0,
37502   /* 95 - ???     */      0,      0,      0,      0,      0,      0,
37503   /* 96 - EXT_KEY */      EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY,
37504   /* 97 - ???     */      0,      0,      0,      0,      0,      0,
37505   /* 98 - ???     */      0,      0,      0,      0,      0,      0,
37506   /* 99 - ???     */      0,      0,      0,      0,      0,      0,
37507   /*100 - ???     */      0,      0,      0,      0,      0,      0,
37508   /*101 - ???     */      0,      0,      0,      0,      0,      0,
37509   /*102 - ???     */      0,      0,      0,      0,      0,      0,
37510   /*103 - ???     */      0,      0,      0,      0,      0,      0,
37511   /*104 - ???     */      0,      0,      0,      0,      0,      0,
37512   /*105 - ???     */      0,      0,      0,      0,      0,      0,
37513   /*106 - ???     */      0,      0,      0,      0,      0,      0,
37514   /*107 - ???     */      0,      0,      0,      0,      0,      0,
```

```
37515    /*108 - ???     */      0,      0,      0,      0,      0,      0,
37516    /*109 - ???     */      0,      0,      0,      0,      0,      0,
37517    /*110 - ???     */      0,      0,      0,      0,      0,      0,
37518    /*111 - ???     */      0,      0,      0,      0,      0,      0,
37519    /*112 - ???     */      0,      0,      0,      0,      0,      0,
37520    /*113 - ???     */      0,      0,      0,      0,      0,      0,
37521    /*114 - ???     */      0,      0,      0,      0,      0,      0,
37522    /*115 - ???     */      0,      0,      0,      0,      0,      0,
37523    /*116 - ???     */      0,      0,      0,      0,      0,      0,
37524    /*117 - ???     */      0,      0,      0,      0,      0,      0,
37525    /*118 - ???     */      0,      0,      0,      0,      0,      0,
37526    /*119 - ???     */      0,      0,      0,      0,      0,      0,
37527    /*120 - ???     */      0,      0,      0,      0,      0,      0,
37528    /*121 - ???     */      0,      0,      0,      0,      0,      0,
37529    /*122 - ???     */      0,      0,      0,      0,      0,      0,
37530    /*123 - ???     */      0,      0,      0,      0,      0,      0,
37531    /*124 - ???     */      0,      0,      0,      0,      0,      0,
37532    /*125 - ???     */      0,      0,      0,      0,      0,      0,
37533    /*126 - ???     */      0,      0,      0,      0,      0,      0,
37534    /*127 - ???     */      0,      0,      0,      0,      0,      0
37535    };
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/keymaps/japanese.src
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
37600    /*
37601     *      Keymap for Japanese 106 keyboard.
37602     *              Dec. 31 1995
37603     *      Toshiya Ogawa   <ogw@shizuokanet.or.jp>
37604     *                      <GCD02425@niftyserve.or.jp>
37605     */
37606
37607    /*
37608     * Japanese 106 keyboard has following additional 5 scan codes
37609     * compared to US standard 101 keyboard.
37610     *
37611     *      scan-code       keytop       effect in this keymap
37612     *      ---------------------------------------------------
37613     *      112(0x70)       KANA            (ignored)
37614     *      115(0x73)       BackSlash       mapped to '\\' and '_'
37615     *      121(0x79)       HENKAN          (ignored)
37616     *      123(0x7B)       MU-HENKAN       (ignored)
37617     *      125(0x7D)       YEN             mapped to '\\' and '|'
37618     */
37619
37620    #if (NR_SCAN_CODES != 0x80)
37621    #error NR_SCAN_CODES mis-match
37622    #endif
37623
37624    u16_t keymap[NR_SCAN_CODES * MAP_COLS] = {
37625
37626    /* scan-code            !Shift  Shift   Alt1    Alt2    Alt+Sh  Ctrl    */
37627    /* ============================================================================ */
37628    /* 00 - none    */      0,      0,      0,      0,      0,      0,
37629    /* 01 - ESC     */      C('['), C('['), CA('['),CA('['),CA('['),C('['),
```

```
37630   /* 02 - '1'     */      '1',    '!',    A('1'), A('1'), A('!'), C('A'),
37631   /* 03 - '2'     */      '2',    '"',    A('2'), A('2'), A('"'), C('B'),
37632   /* 04 - '3'     */      '3',    '#',    A('3'), A('3'), A('#'), C('C'),
37633   /* 05 - '4'     */      '4',    '$',    A('4'), A('4'), A('$'), C('D'),
37634   /* 06 - '5'     */      '5',    '%',    A('5'), A('5'), A('%'), C('E'),
37635   /* 07 - '6'     */      '6',    '&',    A('6'), A('6'), A('&'), C('F'),
37636   /* 08 - '7'     */      '7',    '\'',   A('7'), A('7'), A('\''),C('G'),
37637   /* 09 - '8'     */      '8',    '(',    A('8'), A('8'), A('('), C('H'),
37638   /* 10 - '9'     */      '9',    ')',    A('9'), A('9'), A(')'), C('I'),
37639   /* 11 - '0'     */      '0',    '~',    A('0'), A('0'), A('~'), C('@'),
37640   /* 12 - '-'     */      '-',    '=',    A('-'), A('-'), A('='), C('@'),
37641   /* 13 - '^'     */      '^',    '~',    A('^'), A('^'), A('~'), C('^'),
37642   /* 14 - BS      */      C('H'), C('H'), CA('H'),CA('H'),CA('H'),0177,
37643   /* 15 - TAB     */      C('I'), C('I'), CA('I'),CA('I'),CA('I'),C('I'),
37644   /* 16 - 'q'     */      L('q'), 'Q',    A('q'), A('q'), A('Q'), C('Q'),
37645   /* 17 - 'w'     */      L('w'), 'W',    A('w'), A('w'), A('W'), C('W'),
37646   /* 18 - 'e'     */      L('e'), 'E',    A('e'), A('e'), A('E'), C('E'),
37647   /* 19 - 'r'     */      L('r'), 'R',    A('r'), A('r'), A('R'), C('R'),
37648   /* 20 - 't'     */      L('t'), 'T',    A('t'), A('t'), A('T'), C('T'),
37649   /* 21 - 'y'     */      L('y'), 'Y',    A('y'), A('y'), A('Y'), C('Y'),
37650   /* 22 - 'u'     */      L('u'), 'U',    A('u'), A('u'), A('U'), C('U'),
37651   /* 23 - 'i'     */      L('i'), 'I',    A('i'), A('i'), A('I'), C('I'),
37652   /* 24 - 'o'     */      L('o'), 'O',    A('o'), A('o'), A('O'), C('O'),
37653   /* 25 - 'p'     */      L('p'), 'P',    A('p'), A('p'), A('P'), C('P'),
37654   /* 26 - '@'     */      '@',    '´',    A('@'), A('@'), A('´'), C('@'),
37655   /* 27 - '['     */      '[',    '{',    A('['), A('['), A('{'), C('['),
37656   /* 28 - Enter   */      C('M'), C('M'), CA('M'),CA('M'),CA('M'),C('J'),
37657   /* 29 - Ctrl    */      CTRL,   CTRL,   CTRL,   CTRL,   CTRL,   CTRL,
37658   /* 30 - 'a'     */      L('a'), 'A',    A('a'), A('a'), A('A'), C('A'),
37659   /* 31 - 's'     */      L('s'), 'S',    A('s'), A('s'), A('S'), C('S'),
37660   /* 32 - 'd'     */      L('d'), 'D',    A('d'), A('d'), A('D'), C('D'),
37661   /* 33 - 'f'     */      L('f'), 'F',    A('f'), A('f'), A('F'), C('F'),
37662   /* 34 - 'g'     */      L('g'), 'G',    A('g'), A('g'), A('G'), C('G'),
37663   /* 35 - 'h'     */      L('h'), 'H',    A('h'), A('h'), A('H'), C('H'),
37664   /* 36 - 'j'     */      L('j'), 'J',    A('j'), A('j'), A('J'), C('J'),
37665   /* 37 - 'k'     */      L('k'), 'K',    A('k'), A('k'), A('K'), C('K'),
37666   /* 38 - 'l'     */      L('l'), 'L',    A('l'), A('l'), A('L'), C('L'),
37667   /* 39 - ';'     */      ';',    '+',    A(';'), A(';'), A('+'), C('@'),
37668   /* 40 - ':'     */      ':',    '*',    A(':'), A(':'), A('*'), C('@'),
37669   /* 41 - KANJI   */      0,      0,      0,      0,      0,      0,
37670   /* 42 - l. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
37671   /* 43 - ']'     */      ']',    '}',    A(']'), A(']'), A('}'), C(']'),
37672   /* 44 - 'z'     */      L('z'), 'Z',    A('z'), A('z'), A('Z'), C('Z'),
37673   /* 45 - 'x'     */      L('x'), 'X',    A('x'), A('x'), A('X'), C('X'),
37674   /* 46 - 'c'     */      L('c'), 'C',    A('c'), A('c'), A('C'), C('C'),
37675   /* 47 - 'v'     */      L('v'), 'V',    A('v'), A('v'), A('V'), C('V'),
37676   /* 48 - 'b'     */      L('b'), 'B',    A('b'), A('b'), A('B'), C('B'),
37677   /* 49 - 'n'     */      L('n'), 'N',    A('n'), A('n'), A('N'), C('N'),
37678   /* 50 - 'm'     */      L('m'), 'M',    A('m'), A('m'), A('M'), C('M'),
37679   /* 51 - ','     */      ',',    '<',    A(','), A(','), A('<'), C('@'),
37680   /* 52 - '.'     */      '.',    '>',    A('.'), A('.'), A('>'), C('@'),
37681   /* 53 - '/'     */      '/',    '?',    A('/'), A('/'), A('?'), C('@'),
37682   /* 54 - r. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
37683   /* 55 - '*'     */      '*',    '*',    A('*'), A('*'), A('*'), C('@'),
37684   /* 56 - ALT     */      ALT,    ALT,    ALT,    ALT,    ALT,    ALT,
37685   /* 57 - ' '     */      ' ',    ' ',    A(' '), A(' '), A(' '), C('@'),
37686   /* 58 - CapsLck */      CALOCK, CALOCK, CALOCK, CALOCK, CALOCK, CALOCK,
37687   /* 59 - F1      */      F1,     SF1,    AF1,    AF1,    ASF1,   CF1,
37688   /* 60 - F2      */      F2,     SF2,    AF2,    AF2,    ASF2,   CF2,
37689   /* 61 - F3      */      F3,     SF3,    AF3,    AF3,    ASF3,   CF3,
```

```
37690    /* 62 - F4      */      F4,     SF4,    AF4,    AF4,    ASF4,   CF4,
37691    /* 63 - F5      */      F5,     SF5,    AF5,    AF5,    ASF5,   CF5,
37692    /* 64 - F6      */      F6,     SF6,    AF6,    AF6,    ASF6,   CF6,
37693    /* 65 - F7      */      F7,     SF7,    AF7,    AF7,    ASF7,   CF7,
37694    /* 66 - F8      */      F8,     SF8,    AF8,    AF8,    ASF8,   CF8,
37695    /* 67 - F9      */      F9,     SF9,    AF9,    AF9,    ASF9,   CF9,
37696    /* 68 - F10     */      F10,    SF10,   AF10,   AF10,   ASF10,  CF10,
37697    /* 69 - NumLock */      NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,
37698    /* 70 - ScrLock */      SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,
37699    /* 71 - Home    */      HOME,   '7',    AHOME,  AHOME,  A('7'), CHOME,
37700    /* 72 - CurUp   */      UP,     '8',    AUP,    AUP,    A('8'), CUP,
37701    /* 73 - PgUp    */      PGUP,   '9',    APGUP,  APGUP,  A('9'), CPGUP,
37702    /* 74 - '-'     */      NMIN,   '-',    ANMIN,  ANMIN,  A('-'), CNMIN,
37703    /* 75 - Left    */      LEFT,   '4',    ALEFT,  ALEFT,  A('4'), CLEFT,
37704    /* 76 - MID     */      MID,    '5',    AMID,   AMID,   A('5'), CMID,
37705    /* 77 - Right   */      RIGHT,  '6',    ARIGHT, ARIGHT, A('6'), CRIGHT,
37706    /* 78 - '+'     */      PLUS,   '+',    APLUS,  APLUS,  A('+'), CPLUS,
37707    /* 79 - End     */      END,    '1',    AEND,   AEND,   A('1'), CEND,
37708    /* 80 - Down    */      DOWN,   '2',    ADOWN,  ADOWN,  A('2'), CDOWN,
37709    /* 81 - PgDown  */      PGDN,   '3',    APGDN,  APGDN,  A('3'), CPGDN,
37710    /* 82 - Insert  */      INSRT,  '0',    AINSRT, AINSRT, A('0'), CINSRT,
37711    /* 83 - Delete  */      0177,   '.',    A(0177),A(0177),A('.'), 0177,
37712    /* 84 - Enter   */      C('M'), C('M'), CA('M'),CA('M'),CA('M'),C('J'),
37713    /* 85 - ???     */      0,      0,      0,      0,      0,      0,
37714    /* 86 - ???     */      0,      0,      0,      0,      0,      0,
37715    /* 87 - F11     */      F11,    SF11,   AF11,   AF11,   ASF11,  CF11,
37716    /* 88 - F12     */      F12,    SF12,   AF12,   AF12,   ASF12,  CF12,
37717    /* 89 - ???     */      0,      0,      0,      0,      0,      0,
37718    /* 90 - ???     */      0,      0,      0,      0,      0,      0,
37719    /* 91 - ???     */      0,      0,      0,      0,      0,      0,
37720    /* 92 - ???     */      0,      0,      0,      0,      0,      0,
37721    /* 93 - ???     */      0,      0,      0,      0,      0,      0,
37722    /* 94 - ???     */      0,      0,      0,      0,      0,      0,
37723    /* 95 - ???     */      0,      0,      0,      0,      0,      0,
37724    /* 96 - EXT_KEY */      EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY,
37725    /* 97 - ???     */      0,      0,      0,      0,      0,      0,
37726    /* 98 - ???     */      0,      0,      0,      0,      0,      0,
37727    /* 99 - ???     */      0,      0,      0,      0,      0,      0,
37728    /*100 - ???     */      0,      0,      0,      0,      0,      0,
37729    /*101 - ???     */      0,      0,      0,      0,      0,      0,
37730    /*102 - ???     */      0,      0,      0,      0,      0,      0,
37731    /*103 - ???     */      0,      0,      0,      0,      0,      0,
37732    /*104 - ???     */      0,      0,      0,      0,      0,      0,
37733    /*105 - ???     */      0,      0,      0,      0,      0,      0,
37734    /*106 - ???     */      0,      0,      0,      0,      0,      0,
37735    /*107 - ???     */      0,      0,      0,      0,      0,      0,
37736    /*108 - ???     */      0,      0,      0,      0,      0,      0,
37737    /*109 - ???     */      0,      0,      0,      0,      0,      0,
37738    /*110 - ???     */      0,      0,      0,      0,      0,      0,
37739    /*111 - ???     */      0,      0,      0,      0,      0,      0,
37740    /*112 - KANA    */      0,      0,      0,      0,      0,      0,
37741    /*113 - ???     */      0,      0,      0,      0,      0,      0,
37742    /*114 - ???     */      0,      0,      0,      0,      0,      0,
37743    /*115 - '\\'    */      '\\',   '_',    A('\\'),A('\\'),A('_'), C('_'),
37744    /*116 - ???     */      0,      0,      0,      0,      0,      0,
37745    /*117 - ???     */      0,      0,      0,      0,      0,      0,
37746    /*118 - ???     */      0,      0,      0,      0,      0,      0,
37747    /*119 - ???     */      0,      0,      0,      0,      0,      0,
37748    /*120 - ???     */      0,      0,      0,      0,      0,      0,
37749    /*121 - HENKAN  */      0,      0,      0,      0,      0,      0,
```

```
37750    /*122 - ???     */      0,      0,      0,      0,      0,      0,
37751    /*123 - MU-HENKAN*/     0,      0,      0,      0,      0,      0,
37752    /*124 - ???     */      0,      0,      0,      0,      0,      0,
37753    /*125 - YEN     */      '\\',   '|',    A('\\'),A('\\'),A('|'), C('\\'),
37754    /*126 - ???     */      0,      0,      0,      0,      0,      0,
37755    /*127 - ???     */      0,      0,      0,      0,      0,      0
37756    };
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/keymaps/latin-am.src
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
37800    /***
37801       Keymap for Latin American keyboard. v1.02
37802       Victor A. Rodriguez - El bit Fantasma - Bit-Man@Tasa.Com.AR
37803
37804       The Latin American keyboard makes differences between the left and
37805       right ALT keys (the right one is so called ALT GR), and uses accent.
37806
37807       Release History
37808       ===============
37809       v1.00     Initial version
37810       v1.01     Extended ASCII characters replaced by hex. equivalents
37811       v1.02     NR_SCAN_CODES has grown to 0x80, required by Toshiya Ogawa
37812                 (ogw@shizuokanet.or.jp) and added by Kees J.Bot (kjb@cs.vu.nl)
37813                 in MINIX 1.7.2
37814    ***/
37815
37816    #if (NR_SCAN_CODES != 0x80)
37817    #error NR_SCAN_CODES mis-match
37818    #endif
37819
37820    u16_t keymap[NR_SCAN_CODES * MAP_COLS] = {
37821
37822    /* scan-code            !Shift  Shift   Alt1    Alt2    Alt+Sh  Ctrl    */
37823    /* ============================================================== */
37824    /* 00 - none    */      0,      0,      0,      0,      0,      0,
37825    /* 01 - ESC     */      C('['), C('['), CA('['),CA('['),CA('['),C('['),
37826    /* 02 - '1'     */      '1',    '!',    A('1'), A('1'), A('!'), C('A'),
37827    /* 03 - '2'     */      '2',    '"',    A('2'), A('2'), A('"'), C('@'),
37828    /* 04 - '3'     */      '3',    '#',    A('3'), A('3'), A('#'), C('C'),
37829    /* 05 - '4'     */      '4',    '$',    A('4'), A('4'), A('$'), C('D'),
37830    /* 06 - '5'     */      '5',    '%',    A('5'), A('5'), A('%'), C('E'),
37831    /* 07 - '6'     */      '6',    '&',    A('6'), A('6'), A('$'), C('^'),
37832    /* 08 - '7'     */      '7',    '/',    A('7'), A('7'), A('/'), C('G'),
37833    /* 09 - '8'     */      '8',    '(',    A('8'), A('8'), A('('), C('H'),
37834    /* 10 - '9'     */      '9',    ')',    A('9'), A('9'), A(')'), C('I'),
37835    /* 11 - '0'     */      '0',    '=',    A('0'), A('0'), A('='), C('@'),
37836    /* 12 - '-'     */      '\'',   '?',    A('\''), 0x5c,   A('?'), C('?'),
37837    /* 13 - '('     */      0xa8,   0xad,   A(0xa8),A(0xa8),A(0xad), C('@'),
37838    /* 14 - BS      */      C('H'), C('H'), CA('H'),CA('H'),CA('H'),0177,
37839    /* 15 - TAB     */      C('I'), C('I'), CA('I'),CA('I'),CA('I'),C('I'),
37840    /* 16 - 'q'     */      L('q'), 'Q',    A('q'), 0x40,   A('Q'), C('Q'),
37841    /* 17 - 'w'     */      L('w'), 'W',    A('w'), A('w'), A('W'), C('W'),
37842    /* 18 - 'e'     */      L('e'), 'E',    A('e'), A('e'), A('E'), C('E'),
37843    /* 19 - 'r'     */      L('r'), 'R',    A('r'), A('r'), A('R'), C('R'),
37844    /* 20 - 't'     */      L('t'), 'T',    A('t'), A('t'), A('T'), C('T'),
```

```
37845   /* 21 - 'y'     */      L('y'), 'Y',    A('y'), A('y'), A('Y'), C('Y'),
37846   /* 22 - 'u'     */      L('u'), 'U',    A('u'), A('u'), A('U'), C('U'),
37847   /* 23 - 'i'     */      L('i'), 'I',    A('i'), A('i'), A('I'), C('I'),
37848   /* 24 - 'o'     */      L('o'), 'O',    A('o'), A('o'), A('O'), C('O'),
37849   /* 25 - 'p'     */      L('p'), 'P',    A('p'), A('p'), A('P'), C('P'),
37850   /* 26 - 'o'     */      0xef,   0xf9,   A(0xef),A(0xef),A(0xf9),C(0xef),
37851   /* 27 - '+'     */      '+',    '*',    A('+'), 0x7e,   A('*'), C('+'),
37852   /* 28 - CR/LF   */      C('M'), C('M'), CA('M'),CA('M'),CA('M'),C('J'),
37853   /* 29 - Ctrl    */      CTRL,   CTRL,   CTRL,   CTRL,   CTRL,   CTRL,
37854   /* 30 - 'a'     */      L('a'), 'A',    A('a'), A('a'), A('A'), C('A'),
37855   /* 31 - 's'     */      L('s'), 'S',    A('s'), A('s'), A('S'), C('S'),
37856   /* 32 - 'd'     */      L('d'), 'D',    A('d'), A('d'), A('D'), C('D'),
37857   /* 33 - 'f'     */      L('f'), 'F',    A('f'), A('f'), A('F'), C('F'),
37858   /* 34 - 'g'     */      L('g'), 'G',    A('g'), A('g'), A('G'), C('G'),
37859   /* 35 - 'h'     */      L('h'), 'H',    A('h'), A('h'), A('H'), C('H'),
37860   /* 36 - 'j'     */      L('j'), 'J',    A('j'), A('j'), A('J'), C('J'),
37861   /* 37 - 'k'     */      L('k'), 'K',    A('k'), A('k'), A('K'), C('K'),
37862   /* 38 - 'l'     */      L('l'), 'L',    A('l'), A('l'), A('L'), C('L'),
37863   /* 39 - '$'     */      L(0xa4),0xa5,   A(0xa4),A(0xa4),A(0xa5),C('@'),
37864   /* 40 - '{'     */      '{',    '[',    A('{'), 0x5e,   A('['), C('@'),
37865   /* 41 - '|'     */      '|',    0xf8,   A('|'), 0xaa,   A('\''),C('@'),
37866   /* 42 - l. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
37867   /* 43 - '}'     */      0x7d,   0x5d,   A('<'), 0x60,   A('>'), C('<'),
37868   /* 44 - 'z'     */      L('z'), 'Z',    A('z'), A('z'), A('Z'), C('Z'),
37869   /* 45 - 'x'     */      L('x'), 'X',    A('x'), A('x'), A('X'), C('X'),
37870   /* 46 - 'c'     */      L('c'), 'C',    A('c'), A('c'), A('C'), C('C'),
37871   /* 47 - 'v'     */      L('v'), 'V',    A('v'), A('v'), A('V'), C('V'),
37872   /* 48 - 'b'     */      L('b'), 'B',    A('b'), A('b'), A('B'), C('B'),
37873   /* 49 - 'n'     */      L('n'), 'N',    A('n'), A('n'), A('N'), C('N'),
37874   /* 50 - 'm'     */      L('m'), 'M',    A('m'), A('m'), A('M'), C('M'),
37875   /* 51 - ','     */      ',',    ';',    A(','), A(','), A(';'), C('@'),
37876   /* 52 - '.'     */      '.',    ':',    A('.'), A('.'), A(':'), C('@'),
37877   /* 53 - '/'     */      '-',    '_',    A('-'), A('-'), A('_'), C('@'),
37878   /* 54 - r. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
37879   /* 55 - '*'     */      '*',    '*',    A('*'), A('*'), A('*'), C('@'),
37880   /* 56 - ALT     */      ALT,    ALT,    ALT,    ALT,    ALT,    ALT,
37881   /* 57 - ' '     */      ' ',    ' ',    A(' '), A(' '), A(' '), C('@'),
37882   /* 58 - CapsLck */      CALOCK, CALOCK, CALOCK, CALOCK, CALOCK, CALOCK,
37883   /* 59 - F1      */      F1,     SF1,    AF1,    AF1,    ASF1,   CF1,
37884   /* 60 - F2      */      F2,     SF2,    AF2,    AF2,    ASF2,   CF2,
37885   /* 61 - F3      */      F3,     SF3,    AF3,    AF3,    ASF3,   CF3,
37886   /* 62 - F4      */      F4,     SF4,    AF4,    AF4,    ASF4,   CF4,
37887   /* 63 - F5      */      F5,     SF5,    AF5,    AF5,    ASF5,   CF5,
37888   /* 64 - F6      */      F6,     SF6,    AF6,    AF6,    ASF6,   CF6,
37889   /* 65 - F7      */      F7,     SF7,    AF7,    AF7,    ASF7,   CF7,
37890   /* 66 - F8      */      F8,     SF8,    AF8,    AF8,    ASF8,   CF8,
37891   /* 67 - F9      */      F9,     SF9,    AF9,    AF9,    ASF9,   CF9,
37892   /* 68 - F10     */      F10,    SF10,   AF10,   AF10,   ASF10,  CF10,
37893   /* 69 - NumLock */      NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,
37894   /* 70 - ScrLock */      SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,
37895   /* 71 - Home    */      HOME,   '7',    AHOME,  AHOME,  A('7'), CHOME,
37896   /* 72 - CurUp   */      UP,     '8',    AUP,    AUP,    A('8'), CUP,
37897   /* 73 - PgUp    */      PGUP,   '9',    APGUP,  APGUP,  A('9'), CPGUP,
37898   /* 74 - '-'     */      '-',    '-',    '-',    '-',    '-',    '-',
37899   /* 75 - Left    */      LEFT,   '4',    ALEFT,  ALEFT,  A('4'), CLEFT,
37900   /* 76 - MID     */      MID,    '5',    AMID,   AMID,   A('5'), CMID,
37901   /* 77 - Right   */      RIGHT,  '6',    ARIGHT, ARIGHT, A('6'), CRIGHT,
37902   /* 78 - '+'     */      '+',    '+',    '+',    '+',    '+',    '+',
37903   /* 79 - End     */      END,    '1',    AEND,   AEND,   A('1'), CEND,
37904   /* 80 - Down    */      DOWN,   '2',    ADOWN,  ADOWN,  A('2'), CDOWN,
```

```
37905    /* 81 - PgDown  */      PGDN,   '3',    APGDN,  APGDN,  A('3'), CPGDN,
37906    /* 82 - Insert  */      INSRT,  '0',    AINSRT, AINSRT, A('0'), CINSRT,
37907    /* 83 - Delete  */      0177,   '.',    A(0177),A(0177),A('.'), 0177,
37908    /* 84 - Enter   */      C('M'), C('M'), CA('M'),CA('M'),CA('M'),C('J'),
37909    /* 85 - ???     */      0,      0,      0,      0,      0,      0,
37910    /* 86 - ???     */      '<',    '>',   A('<'),  A('>'), A('>'), C('@'),
37911    /* 87 - F11     */      F11,    SF11,   AF11,   AF11,   ASF11,  CF11,
37912    /* 88 - F12     */      F12,    SF12,   AF12,   AF12,   ASF12,  CF12,
37913    /* 89 - ???     */      0,      0,      0,      0,      0,      0,
37914    /* 90 - ???     */      0,      0,      0,      0,      0,      0,
37915    /* 91 - ???     */      0,      0,      0,      0,      0,      0,
37916    /* 92 - ???     */      0,      0,      0,      0,      0,      0,
37917    /* 93 - ???     */      0,      0,      0,      0,      0,      0,
37918    /* 94 - ???     */      0,      0,      0,      0,      0,      0,
37919    /* 95 - ???     */      0,      0,      0,      0,      0,      0,
37920    /* 96 - EXT_KEY */      EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY,
37921    /* 97 - ???     */      0,      0,      0,      0,      0,      0,
37922    /* 98 - ???     */      0,      0,      0,      0,      0,      0,
37923    /* 99 - ???     */      0,      0,      0,      0,      0,      0,
37924    /*100 - ???     */      0,      0,      0,      0,      0,      0,
37925    /*101 - ???     */      0,      0,      0,      0,      0,      0,
37926    /*102 - ???     */      0,      0,      0,      0,      0,      0,
37927    /*103 - ???     */      0,      0,      0,      0,      0,      0,
37928    /*104 - ???     */      0,      0,      0,      0,      0,      0,
37929    /*105 - ???     */      0,      0,      0,      0,      0,      0,
37930    /*106 - ???     */      0,      0,      0,      0,      0,      0,
37931    /*107 - ???     */      0,      0,      0,      0,      0,      0,
37932    /*108 - ???     */      0,      0,      0,      0,      0,      0,
37933    /*109 - ???     */      0,      0,      0,      0,      0,      0,
37934    /*110 - ???     */      0,      0,      0,      0,      0,      0,
37935    /*111 - ???     */      0,      0,      0,      0,      0,      0,
37936    /*112 - ???     */      0,      0,      0,      0,      0,      0,
37937    /*113 - ???     */      0,      0,      0,      0,      0,      0,
37938    /*114 - ???     */      0,      0,      0,      0,      0,      0,
37939    /*115 - ???     */      0,      0,      0,      0,      0,      0,
37940    /*116 - ???     */      0,      0,      0,      0,      0,      0,
37941    /*117 - ???     */      0,      0,      0,      0,      0,      0,
37942    /*118 - ???     */      0,      0,      0,      0,      0,      0,
37943    /*119 - ???     */      0,      0,      0,      0,      0,      0,
37944    /*120 - ???     */      0,      0,      0,      0,      0,      0,
37945    /*121 - ???     */      0,      0,      0,      0,      0,      0,
37946    /*122 - ???     */      0,      0,      0,      0,      0,      0,
37947    /*123 - ???     */      0,      0,      0,      0,      0,      0,
37948    /*124 - ???     */      0,      0,      0,      0,      0,      0,
37949    /*125 - ???     */      0,      0,      0,      0,      0,      0,
37950    /*126 - ???     */      0,      0,      0,      0,      0,      0,
37951    /*127 - ???     */      0,      0,      0,      0,      0,      0
37952    };
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/keymaps/olivetti.src
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

38000    /* Keymap for the Olivetti M24. */
38001
38002    u16_t keymap[NR_SCAN_CODES * MAP_COLS] = {
38003
38004    /* scan-code            !Shift  Shift   Alt1    Alt2    Alt+Sh  Ctrl    */
38005    /* ============================================================== */
38006    /* 00 - none    */      0,      0,      0,      0,      0,      0,
38007    /* 01 - ESC     */      C('['), C('['), CA('['),CA('['),CA('['),C('['),
38008    /* 02 - '1'     */      '1',    '!',    A('1'), A('1'), A('!'), C('A'),
38009    /* 03 - '2'     */      '2',    '"',    A('2'), A('2'), A('"'), C('B'),
38010    /* 04 - '3'     */      '3',    '#',    A('3'), A('3'), A('#'), C('C'),
38011    /* 05 - '4'     */      '4',    '$',    A('4'), A('4'), A('$'), C('D'),
38012    /* 06 - '5'     */      '5',    '%',    A('5'), A('5'), A('%'), C('E'),
38013    /* 07 - '6'     */      '6',    '&',    A('6'), A('6'), A('&'), C('F'),
38014    /* 08 - '7'     */      '7',    '\'',   A('7'), A('7'), A('\''),C('G'),
38015    /* 09 - '8'     */      '8',    '(',    A('8'), A('8'), A('('), C('H'),
38016    /* 10 - '9'     */      '9',    ')',    A('9'), A('9'), A(')'), C('I'),
38017    /* 11 - '0'     */      '0',    '_',    A('0'), A('0'), A('_'), C('@'),
38018    /* 12 - '-'     */      '-',    '=',    A('-'), A('-'), A('='), C('@'),
38019    /* 13 - '='     */      '^',    '~',    A('^'), A('^'), A('~'), C('^'),
38020    /* 14 - BS      */      C('H'), C('H'), CA('H'),CA('H'),CA('H'),0177,
38021    /* 15 - TAB     */      C('I'), C('I'), CA('I'),CA('I'),CA('I'),C('I'),
38022    /* 16 - 'q'     */      L('q'), 'Q',    A('q'), A('q'), A('Q'), C('Q'),
38023    /* 17 - 'w'     */      L('w'), 'W',    A('w'), A('w'), A('W'), C('W'),
38024    /* 18 - 'e'     */      L('e'), 'E',    A('e'), A('e'), A('E'), C('E'),
38025    /* 19 - 'r'     */      L('r'), 'R',    A('r'), A('r'), A('R'), C('R'),
38026    /* 20 - 't'     */      L('t'), 'T',    A('t'), A('t'), A('T'), C('T'),
38027    /* 21 - 'y'     */      L('y'), 'Y',    A('y'), A('y'), A('Y'), C('Y'),
38028    /* 22 - 'u'     */      L('u'), 'U',    A('u'), A('u'), A('U'), C('U'),
38029    /* 23 - 'i'     */      L('i'), 'I',    A('i'), A('i'), A('I'), C('I'),
38030    /* 24 - 'o'     */      L('o'), 'O',    A('o'), A('o'), A('O'), C('O'),
38031    /* 25 - 'p'     */      L('p'), 'P',    A('p'), A('p'), A('P'), C('P'),
38032    /* 26 - '['     */      '@',    '´',    A('@'), A('@'), A('´'), C('@'),
38033    /* 27 - ']'     */      '[',    '{',    A('['), A('['), A('{'), C('['),
38034    /* 28 - CR/LF   */      C('M'), C('M'), CA('M'),CA('M'),CA('M'),C('J'),
38035    /* 29 - Ctrl    */      CTRL,   CTRL,   CTRL,   CTRL,   CTRL,   CTRL,
38036    /* 30 - 'a'     */      L('a'), 'A',    A('a'), A('a'), A('A'), C('A'),
38037    /* 31 - 's'     */      L('s'), 'S',    A('s'), A('s'), A('S'), C('S'),
38038    /* 32 - 'd'     */      L('d'), 'D',    A('d'), A('d'), A('D'), C('D'),
38039    /* 33 - 'f'     */      L('f'), 'F',    A('f'), A('f'), A('F'), C('F'),
38040    /* 34 - 'g'     */      L('g'), 'G',    A('g'), A('g'), A('G'), C('G'),
38041    /* 35 - 'h'     */      L('h'), 'H',    A('h'), A('h'), A('H'), C('H'),
38042    /* 36 - 'j'     */      L('j'), 'J',    A('j'), A('j'), A('J'), C('J'),
38043    /* 37 - 'k'     */      L('k'), 'K',    A('k'), A('k'), A('K'), C('K'),
38044    /* 38 - 'l'     */      L('l'), 'L',    A('l'), A('l'), A('L'), C('L'),
38045    /* 39 - ';'     */      ';',    '+',    A(';'), A(';'), A('+'), C('@'),
38046    /* 40 - '\''    */      ':',    '*',    A(':'), A(':'), A('*'), C('@'),
38047    /* 41 - '´      */     ']',    '}',    A(']'), A(']'), A('}'), C(']'),
38048    /* 42 - l. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
38049    /* 43 - '\\'    */      '\\',   '|',    A('\\'),A('\\'),A('|'), C('\\'),
38050    /* 44 - 'z'     */      L('z'), 'Z',    A('z'), A('z'), A('Z'), C('Z'),
38051    /* 45 - 'x'     */      L('x'), 'X',    A('x'), A('x'), A('X'), C('X'),
38052    /* 46 - 'c'     */      L('c'), 'C',    A('c'), A('c'), A('C'), C('C'),
38053    /* 47 - 'v'     */      L('v'), 'V',    A('v'), A('v'), A('V'), C('V'),
38054    /* 48 - 'b'     */      L('b'), 'B',    A('b'), A('b'), A('B'), C('B'),
```

```
38055   /* 49 - 'n'     */      L('n'), 'N',    A('n'), A('n'), A('N'), C('N'),
38056   /* 50 - 'm'     */      L('m'), 'M',    A('m'), A('m'), A('M'), C('M'),
38057   /* 51 - ','     */      ',',    '<',    A(','), A(','), A('<'), C('@'),
38058   /* 52 - '.'     */      '.',    '>',    A('.'), A('.'), A('>'), C('@'),
38059   /* 53 - '/'     */      '/',    '?',    A('/'), A('/'), A('?'), C('@'),
38060   /* 54 - r. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
38061   /* 55 - '*'     */      '*',    '*',    A('*'), A('*'), A('*'), C('@'),
38062   /* 56 - ALT     */      ALT,    ALT,    ALT,    ALT,    ALT,    ALT,
38063   /* 57 - ' '     */      ' ',    ' ',    A(' '), A(' '), A(' '), C('@'),
38064   /* 58 - CapsLck */      CALOCK, CALOCK, CALOCK, CALOCK, CALOCK, CALOCK,
38065   /* 59 - F1      */      F1,     SF1,    AF1,    AF1,    ASF1,   CF1,
38066   /* 60 - F2      */      F2,     SF2,    AF2,    AF2,    ASF2,   CF2,
38067   /* 61 - F3      */      F3,     SF3,    AF3,    AF3,    ASF3,   CF3,
38068   /* 62 - F4      */      F4,     SF4,    AF4,    AF4,    ASF4,   CF4,
38069   /* 63 - F5      */      F5,     SF5,    AF5,    AF5,    ASF5,   CF5,
38070   /* 64 - F6      */      F6,     SF6,    AF6,    AF6,    ASF6,   CF6,
38071   /* 65 - F7      */      F7,     SF7,    AF7,    AF7,    ASF7,   CF7,
38072   /* 66 - F8      */      F8,     SF8,    AF8,    AF8,    ASF8,   CF8,
38073   /* 67 - F9      */      F9,     SF9,    AF9,    AF9,    ASF9,   CF9,
38074   /* 68 - F10     */      F10,    SF10,   AF10,   AF10,   ASF10,  CF10,
38075   /* 69 - NumLock */      C('S'), C('S'), C('S'), C('S'), C('S'), C('S'),
38076   /* 70 - ScrLock */      SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,
38077   /* 71 - Home    */      HOME,   '7',    AHOME,  AHOME,  A('7'), CHOME,
38078   /* 72 - CurUp   */      UP,     '8',    AUP,    AUP,    A('8'), CUP,
38079   /* 73 - PgUp    */      PGUP,   '9',    APGUP,  APGUP,  A('9'), CPGUP,
38080   /* 74 - '-'     */      NMIN,   '-',    ANMIN,  ANMIN,  A('-'), CNMIN,
38081   /* 75 - Left    */      LEFT,   '4',    ALEFT,  ALEFT,  A('4'), CLEFT,
38082   /* 76 - MID     */      MID,    '5',    AMID,   AMID,   A('5'), CMID,
38083   /* 77 - Right   */      RIGHT,  '6',    ARIGHT, ARIGHT, A('6'), CRIGHT,
38084   /* 78 - '+'     */      PLUS,   '+',    APLUS,  APLUS,  A('+'), CPLUS,
38085   /* 79 - End     */      END,    '1',    AEND,   AEND,   A('1'), CEND,
38086   /* 80 - Down    */      DOWN,   '2',    ADOWN,  ADOWN,  A('2'), CDOWN,
38087   /* 81 - PgDown  */      PGDN,   '3',    APGDN,  APGDN,  A('3'), CPGDN,
38088   /* 82 - Insert  */      INSRT,  '0',    AINSRT, AINSRT, A('0'), CINSRT,
38089   /* 83 - Delete  */      0177,   '.',    A(0177),A(0177),A('.'), 0177,
38090   /* 84 - Enter   */      ' ',    ' ',    A(' '), A(' '), A(' '), C('@'),
38091   /* 85 - ???     */      LEFT,   014,    A(014), A(014), A(014), 014,
38092   /* 86 - ???     */      0212,   0212,   0212,   0212,   0212,   0212,
38093   /* 87 - F11     */      C('M'), C('M'), CA('M'),CA('M'),CA('M'),C('J'),
38094   /* 88 - F12     */      F12,    SF12,   AF12,   AF12,   ASF12,  CF12,
38095   /* 89 - ???     */      DOWN,   DOWN,   DOWN,   DOWN,   DOWN,   DOWN,
38096   /* 90 - ???     */      RIGHT,  RIGHT,  RIGHT,  RIGHT,  RIGHT,  RIGHT,
38097   /* 91 - ???     */      UP,     UP,     UP,     UP,     UP,     UP,
38098   /* 92 - ???     */      LEFT,   LEFT,   LEFT,   LEFT,   LEFT,   LEFT,
38099   /* 93 - ???     */      0,      0,      0,      0,      0,      0,
38100   /* 94 - ???     */      0,      0,      0,      0,      0,      0,
38101   /* 95 - ???     */      0,      0,      0,      0,      0,      0,
38102   /* 96 - EXT_KEY */      EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY,
38103   /* 97 - ???     */      0,      0,      0,      0,      0,      0,
38104   /* 98 - ???     */      0,      0,      0,      0,      0,      0,
38105   /* 99 - ???     */      0,      0,      0,      0,      0,      0,
38106   /*100 - ???     */      0,      0,      0,      0,      0,      0,
38107   /*101 - ???     */      0,      0,      0,      0,      0,      0,
38108   /*102 - ???     */      0,      0,      0,      0,      0,      0,
38109   /*103 - ???     */      0,      0,      0,      0,      0,      0,
38110   /*104 - ???     */      0,      0,      0,      0,      0,      0,
38111   /*105 - ???     */      0,      0,      0,      0,      0,      0,
38112   /*106 - ???     */      0,      0,      0,      0,      0,      0,
38113   /*107 - ???     */      0,      0,      0,      0,      0,      0,
38114   /*108 - ???     */      0,      0,      0,      0,      0,      0,
```

```
38115    /*109 - ???     */      0,      0,      0,      0,      0,      0,
38116    /*110 - ???     */      0,      0,      0,      0,      0,      0,
38117    /*111 - ???     */      0,      0,      0,      0,      0,      0,
38118    /*112 - ???     */      0,      0,      0,      0,      0,      0,
38119    /*113 - ???     */      0,      0,      0,      0,      0,      0,
38120    /*114 - ???     */      0,      0,      0,      0,      0,      0,
38121    /*115 - ???     */      0,      0,      0,      0,      0,      0,
38122    /*116 - ???     */      0,      0,      0,      0,      0,      0,
38123    /*117 - ???     */      0,      0,      0,      0,      0,      0,
38124    /*118 - ???     */      0,      0,      0,      0,      0,      0,
38125    /*119 - ???     */      0,      0,      0,      0,      0,      0,
38126    /*120 - ???     */      0,      0,      0,      0,      0,      0,
38127    /*121 - ???     */      0,      0,      0,      0,      0,      0,
38128    /*122 - ???     */      0,      0,      0,      0,      0,      0,
38129    /*123 - ???     */      0,      0,      0,      0,      0,      0,
38130    /*124 - ???     */      0,      0,      0,      0,      0,      0,
38131    /*125 - ???     */      0,      0,      0,      0,      0,      0,
38132    /*126 - ???     */      0,      0,      0,      0,      0,      0,
38133    /*127 - ???     */      0,      0,      0,      0,      0,      0
38134    };
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/keymaps/scandin.src
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

38200    /* Keymap for Scandinavian keyboard. */
38201
38202    u16_t keymap[NR_SCAN_CODES * MAP_COLS] = {
38203
38204    /* scan-code            unsh    Shift   Alt     AltGr   Alt+Sh  Strg    */
38205    /* ============================================================== */
38206    /* 00 - none    */      0,      0,      0,      0,      0,      0,
38207    /* 01 - ESC     */      C('['), C('['), CA('['),C('['), C('['), C('['),
38208    /* 02 - '1'     */      '1',    '!',    A('1'), '1',    '!',    C('A'),
38209    /* 03 - '2'     */      '2',    '"',    A('2'), '@',    '@',    C('@'),
38210    /* 04 - '3'     */      '3',    '#',    A('3'), 156,    '#',    C('C'),
38211    /* 05 - '4'     */      '4',    ' ',    A('4'), '$',    '$',    C('D'),
38212    /* 06 - '5'     */      '5',    '%',    A('5'), '5',    '%',    C('E'),
38213    /* 07 - '6'     */      '6',    '&',    A('6'), '6',    '^',    C('^'),
38214    /* 08 - '7'     */      '7',    '/',    A('7'), '{',    '&',    C('G'),
38215    /* 09 - '8'     */      '8',    '(',    A('8'), '[',    '*',    C('H'),
38216    /* 10 - '9'     */      '9',    ')',    A('9'), ']',    '(',    C('I'),
38217    /* 11 - '0'     */      '0',    '=',    A('0'), '}',    ')',    C('@'),
38218    /* 12 - '_'     */      '+',    '?',    0341,   '\\',   '_',    C('_'),
38219    /* 13 - '='     */      '\'',   '`',    A('\''),'=',    '+',    C('@'),
38220    /* 14 - BS      */      C('H'), C('H'), CA('H'),C('H'), C('H'), 0177,
38221    /* 15 - TAB     */      C('I'), C('I'), CA('I'),C('I'), C('I'), C('I'),
38222    /* 16 - 'q'     */      L('q'), 'Q',    A('q'), '@',    'Q',    C('Q'),
38223    /* 17 - 'w'     */      L('w'), 'W',    A('w'), 'w',    'W',    C('W'),
38224    /* 18 - 'e'     */      L('e'), 'E',    A('e'), 'e',    'E',    C('E'),
38225    /* 19 - 'r'     */      L('r'), 'R',    A('r'), 'r',    'R',    C('R'),
38226    /* 20 - 't'     */      L('t'), 'T',    A('t'), 't',    'T',    C('T'),
38227    /* 21 - 'y'     */      L('y'), 'Y',    A('y'), 'y',    'Y',    C('Y'),
38228    /* 22 - 'u'     */      L('u'), 'U',    A('u'), 'u',    'U',    C('U'),
38229    /* 23 - 'i'     */      L('i'), 'I',    A('i'), 'i',    'I',    C('I'),
```

```
38230   /* 24 - 'o'     */      L('o'), 'O',    A('o'), 'o',    'O',    C('O'),
38231   /* 25 - 'p'     */      L('p'), 'P',    A('p'), 'p',    'P',    C('P'),
38232   /* 26 - '['     */      L(134),  143,   0201,   '[',    '{',    C('['),
38233   /* 27 - ']'     */      '\"',   '^',    A('+'), '~',    ']',    C(']'),
38234   /* 28 - CR/LF   */      C('M'), C('M'), CA('M'),C('M'), C('M'), C('J'),
38235   /* 29 - Strg;-) */      CTRL,   CTRL,   CTRL,   CTRL,   CTRL,   CTRL,
38236   /* 30 - 'a'     */      L('a'), 'A',    A('a'), 'a',    'A',    C('A'),
38237   /* 31 - 's'     */      L('s'), 'S',    A('s'), 's',    'S',    C('S'),
38238   /* 32 - 'd'     */      L('d'), 'D',    A('d'), 'd',    'D',    C('D'),
38239   /* 33 - 'f'     */      L('f'), 'F',    A('f'), 'f',    'F',    C('F'),
38240   /* 34 - 'g'     */      L('g'), 'G',    A('g'), 'g',    'G',    C('G'),
38241   /* 35 - 'h'     */      L('h'), 'H',    A('h'), 'h',    'H',    C('H'),
38242   /* 36 - 'j'     */      L('j'), 'J',    A('j'), 'j',    'J',    C('J'),
38243   /* 37 - 'k'     */      L('k'), 'K',    A('k'), 'k',    'K',    C('K'),
38244   /* 38 - 'l'     */      L('l'), 'L',    A('l'), 'l',    'L',    C('L'),
38245   /* 39 - ';'     */      L(0224),0231,   0224,   ';',    ':',    C('@'),
38246   /* 40 - '\''    */      L(0204),0216,   0204,   '\'',   '"',    C('@'),
38247   /* 41 - '´     */      L(21),  171,    A('^'), '´',    '~',    C('^'),
38248   /* 42 - SHIFT   */      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
38249   /* 43 - '\\'    */      39,     '*',    A('#'), '\\',   '|',    C('\\'),
38250   /* 44 - 'z'     */      L('z'), 'Z',    A('z'), 'z',    'Z',    C('Z'),
38251   /* 45 - 'x'     */      L('x'), 'X',    A('x'), 'x',    'X',    C('X'),
38252   /* 46 - 'c'     */      L('c'), 'C',    A('c'), 'c',    'C',    C('C'),
38253   /* 47 - 'v'     */      L('v'), 'V',    A('v'), 'v',    'V',    C('V'),
38254   /* 48 - 'b'     */      L('b'), 'B',    A('b'), 'b',    'B',    C('B'),
38255   /* 49 - 'n'     */      L('n'), 'N',    A('n'), 'n',    'N',    C('N'),
38256   /* 50 - 'm'     */      L('m'), 'M',    A('m'), 0346,   'M',    C('M'),
38257   /* 51 - ','     */      ',',    ';',    A(','), ',',    '<',    C('@'),
38258   /* 52 - '.'     */      '.',    ':',    A('.'), '.',    '>',    C('@'),
38259   /* 53 - '/'     */      '-',    '_',    A('-'), '/',    '?',    C('_'),
38260   /* 54 - SHIFT   */      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
38261   /* 55 - '*'     */      '*',    '*',    A('*'), '*',    '*',    C('@'),
38262   /* 56 - ALT     */      ALT,    ALT,    ALT,    ALT,    ALT,    ALT,
38263   /* 57 - ' '     */      ' ',    ' ',    A(' '), ' ',    ' ',    C('@'),
38264   /* 58 - CapsLck */      CALOCK, CALOCK, CALOCK, CALOCK, CALOCK, CALOCK,
38265   /* 59 - F1      */      F1,     SF1,    AF1,    AF1,    ASF1,   CF1,
38266   /* 60 - F2      */      F2,     SF2,    AF2,    AF2,    ASF2,   CF2,
38267   /* 61 - F3      */      F3,     SF3,    AF3,    AF3,    ASF3,   CF3,
38268   /* 62 - F4      */      F4,     SF4,    AF4,    AF4,    ASF4,   CF4,
38269   /* 63 - F5      */      F5,     SF5,    AF5,    AF5,    ASF5,   CF5,
38270   /* 64 - F6      */      F6,     SF6,    AF6,    AF6,    ASF6,   CF6,
38271   /* 65 - F7      */      F7,     SF7,    AF7,    AF7,    ASF7,   CF7,
38272   /* 66 - F8      */      F8,     SF8,    AF8,    AF8,    ASF8,   CF8,
38273   /* 67 - F9      */      F9,     SF9,    AF9,    AF9,    ASF9,   CF9,
38274   /* 68 - F10     */      F10,    SF10,   AF10,   AF10,   ASF10,  CF10,
38275   /* 69 - NumLock */      NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,
38276   /* 70 - ScrLock */      SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,
38277   /* 71 - Home    */      HOME,   '7',    AHOME,  AHOME,  '7',    CHOME,
38278   /* 72 - CurUp   */      UP,     '8',    AUP,    AUP,    '8',    CUP,
38279   /* 73 - PgUp    */      PGUP,   '9',    APGUP,  APGUP,  '9',    CPGUP,
38280   /* 74 - '-'     */      NMIN,   '-',    ANMIN,  ANMIN,  '-',    CNMIN,
38281   /* 75 - Left    */      LEFT,   '4',    ALEFT,  ALEFT,  '4',    CLEFT,
38282   /* 76 - MID     */      MID,    '5',    AMID,   AMID,   '5',    CMID,
38283   /* 77 - Right   */      RIGHT,  '6',    ARIGHT, ARIGHT, '6',    CRIGHT,
38284   /* 78 - '+'     */      PLUS,   '+',    APLUS,  APLUS,  '+',    CPLUS,
38285   /* 79 - End     */      END,    '1',    AEND,   AEND,   '1',    CEND,
38286   /* 80 - Down    */      DOWN,   '2',    ADOWN,  ADOWN,  '2',    CDOWN,
38287   /* 81 - PgDown  */      PGDN,   '3',    APGDN,  APGDN,  '3',    CPGDN,
38288   /* 82 - Insert  */      INSRT,  '0',    AINSRT, AINSRT, '0',    CINSRT,
38289   /* 83 - Delete  */      0177,   '.',    A(0177),0177,   '.',    0177,
```

```
38290    /* 84 - Enter   */      C('M'), C('M'), CA('M'),C('M'), C('M'), C('J'),
38291    /* 85 - ???     */      0,      0,      0,      0,      0,      0,
38292    /* 86 - ???     */      '<',    '>',    A('<'), '|',    '>',    C('@'),
38293    /* 87 - F11     */      F11,    SF11,   AF11,   AF11,   ASF11,  CF11,
38294    /* 88 - F12     */      F12,    SF12,   AF12,   AF12,   ASF12,  CF12,
38295    /* 89 - ???     */      0,      0,      0,      0,      0,      0,
38296    /* 90 - ???     */      0,      0,      0,      0,      0,      0,
38297    /* 91 - ???     */      0,      0,      0,      0,      0,      0,
38298    /* 92 - ???     */      0,      0,      0,      0,      0,      0,
38299    /* 93 - ???     */      0,      0,      0,      0,      0,      0,
38300    /* 94 - ???     */      0,      0,      0,      0,      0,      0,
38301    /* 95 - ???     */      0,      0,      0,      0,      0,      0,
38302    /* 96 - EXT_KEY */      EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY,
38303    /* 97 - ???     */      0,      0,      0,      0,      0,      0,
38304    /* 98 - ???     */      0,      0,      0,      0,      0,      0,
38305    /* 99 - ???     */      0,      0,      0,      0,      0,      0,
38306    /*100 - ???     */      0,      0,      0,      0,      0,      0,
38307    /*101 - ???     */      0,      0,      0,      0,      0,      0,
38308    /*102 - ???     */      0,      0,      0,      0,      0,      0,
38309    /*103 - ???     */      0,      0,      0,      0,      0,      0,
38310    /*104 - ???     */      0,      0,      0,      0,      0,      0,
38311    /*105 - ???     */      0,      0,      0,      0,      0,      0,
38312    /*106 - ???     */      0,      0,      0,      0,      0,      0,
38313    /*107 - ???     */      0,      0,      0,      0,      0,      0,
38314    /*108 - ???     */      0,      0,      0,      0,      0,      0,
38315    /*109 - ???     */      0,      0,      0,      0,      0,      0,
38316    /*110 - ???     */      0,      0,      0,      0,      0,      0,
38317    /*111 - ???     */      0,      0,      0,      0,      0,      0,
38318    /*112 - ???     */      0,      0,      0,      0,      0,      0,
38319    /*113 - ???     */      0,      0,      0,      0,      0,      0,
38320    /*114 - ???     */      0,      0,      0,      0,      0,      0,
38321    /*115 - ???     */      0,      0,      0,      0,      0,      0,
38322    /*116 - ???     */      0,      0,      0,      0,      0,      0,
38323    /*117 - ???     */      0,      0,      0,      0,      0,      0,
38324    /*118 - ???     */      0,      0,      0,      0,      0,      0,
38325    /*119 - ???     */      0,      0,      0,      0,      0,      0,
38326    /*120 - ???     */      0,      0,      0,      0,      0,      0,
38327    /*121 - ???     */      0,      0,      0,      0,      0,      0,
38328    /*122 - ???     */      0,      0,      0,      0,      0,      0,
38329    /*123 - ???     */      0,      0,      0,      0,      0,      0,
38330    /*124 - ???     */      0,      0,      0,      0,      0,      0,
38331    /*125 - ???     */      0,      0,      0,      0,      0,      0,
38332    /*126 - ???     */      0,      0,      0,      0,      0,      0,
38333    /*127 - ???     */      0,      0,      0,      0,      0,      0
38334    };
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/keymaps/spanish.src
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

38400    /* Keymap for Spanish MF-2 keyboard. */
38401    /* Modified by Javier Garcia Martin jawa@inf.deusto.es */
38402
38403    u16_t keymap[NR_SCAN_CODES * MAP_COLS] = {
38404
38405    /* scan-code            !Shift  Shift   Alt     AltGr   Alt+Sh  Ctrl    */
38406    /*
38407    ================================================================
38408    */
38409    /* 00 - none    */      0,      0,      0,      0,      0,      0,
38410    /* 01 - ESC     */      C('['), C('['), CA('['),C('['), C('['), C('['),
38411    /* 02 - '1'     */      '1',    '!',    A('1'), '|',    '!',    C('A'),
38412    /* 03 - '2'     */      '2',    '"',    A('2'), '@',    '"',    C('@'),
38413    /* 04 - '3'     */      '3',    0372,   A('3'), '#',    0372,   C('C'),
38414    /* 05 - '4'     */      '4',    '$',    A('4'), '4',    '$',    C('D'),
38415    /* 06 - '5'     */      '5',    '%',    A('5'), '5',    '%',    C('E'),
38416    /* 07 - '6'     */      '6',    '&',    A('6'), 0252,   '&',    C('F'),
38417    /* 08 - '7'     */      '7',    '/',    A('7'), '{',    '/',    C('G'),
38418    /* 09 - '8'     */      '8',    '(',    A('8'), '(',    '(',    C('H'),
38419    /* 10 - '9'     */      '9',    ')',    A('9'), ')',    ')',    C('I'),
38420    /* 11 - '0'     */      '0',    '=',    A('0'), '=',    '=',    C('@'),
38421    /* 12 - '-'     */      '\'',   '?',    A('\''),'?',    '?',    C('_'),
38422    /* 13 - '='     */      0255,   0250,   A(0255),0250,   0250,   C('@'),
38423    /* 14 - BS      */      C('H'), C('H'), CA('H'),C('H'), C('H'), 0177,
38424    /* 15 - TAB     */      C('I'), C('I'), CA('I'),C('I'), C('I'), C('I'),
38425    /* 16 - 'q'     */      L('q'), 'Q',    A('q'), 'q',    'Q',    C('Q'),
38426    /* 17 - 'w'     */      L('w'), 'W',    A('w'), 'w',    'W',    C('W'),
38427    /* 18 - 'e'     */      L('e'), 'E',    A('e'), 'e',    'E',    C('E'),
38428    /* 19 - 'r'     */      L('r'), 'R',    A('r'), 'r',    'R',    C('R'),
38429    /* 20 - 't'     */      L('t'), 'T',    A('t'), 't',    'T',    C('T'),
38430    /* 21 - 'y'     */      L('y'), 'Y',    A('y'), 'y',    'Y',    C('Y'),
38431    /* 22 - 'u'     */      L('u'), 'U',    A('u'), 'u',    'U',    C('U'),
38432    /* 23 - 'i'     */      L('i'), 'I',    A('i'), 'i',    'I',    C('I'),
38433    /* 24 - 'o'     */      L('o'), 'O',    A('o'), 'o',    'O',    C('O'),
38434    /* 25 - 'p'     */      L('p'), 'P',    A('p'), 'p',    'P',    C('P'),
38435    /* 26 - '['     */      '´',    '^',    A('´'),'[',     '^',    C('['),
38436    /* 27 - ']'     */      '+',    '*',    A('+'), ']',    '*',    C(']'),
38437    /* 28 - CR/LF   */      C('M'), C('M'), CA('M'),C('M'), C('M'), C('J'),
38438    /* 29 - Ctrl    */      CTRL,   CTRL,   CTRL,   CTRL,   CTRL,   CTRL,
38439    /* 30 - 'a'     */      L('a'), 'A',    A('a'), 'a',    'A',    C('A'),
38440    /* 31 - 's'     */      L('s'), 'S',    A('s'), 's',    'S',    C('S'),
38441    /* 32 - 'd'     */      L('d'), 'D',    A('d'), 'd',    'D',    C('D'),
38442    /* 33 - 'f'     */      L('f'), 'F',    A('f'), 'f',    'F',    C('F'),
38443    /* 34 - 'g'     */      L('g'), 'G',    A('g'), 'g',    'G',    C('G'),
38444    /* 35 - 'h'     */      L('h'), 'H',    A('h'), 'h',    'H',    C('H'),
38445    /* 36 - 'j'     */      L('j'), 'J',    A('j'), 'j',    'J',    C('J'),
38446    /* 37 - 'k'     */      L('k'), 'K',    A('k'), 'k',    'K',    C('K'),
38447    /* 38 - 'l'     */      L('l'), 'L',    A('l'), 'l',    'L',    C('L'),
38448    /* 39 - ';'     */      L(0244),0245,   A(0244),0244,   0245,   C('@'),
38449    /* 40 - '\''    */      '\'',   '"',    A('\''),'{',    '"',    C('@'),
38450    /* 41 - '´     */      0247,   0246,   A(0247),'\\',   0246,   C('@'),
38451    /* 42 - l. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
38452    /* 43 - '\\'    */      L(0207),0200,   A(0207),'}',    0200,   C('@'),
38453    /* 44 - 'z'     */      L('z'), 'Z',    A('z'), 'z',    'Z',    C('Z'),
38454    /* 45 - 'x'     */      L('x'), 'X',    A('x'), 'x',    'X',    C('X'),
```

```
38455   /* 46 - 'c'     */      L('c'), 'C',    A('c'), 'c',    'C',    C('C'),
38456   /* 47 - 'v'     */      L('v'), 'V',    A('v'), 'v',    'V',    C('V'),
38457   /* 48 - 'b'     */      L('b'), 'B',    A('b'), 'b',    'B',    C('B'),
38458   /* 49 - 'n'     */      L('n'), 'N',    A('n'), 'n',    'N',    C('N'),
38459   /* 50 - 'm'     */      L('m'), 'M',    A('m'), 'm',    'M',    C('M'),
38460   /* 51 - ','     */      ',',    ';',    A(','), ',',    ';',    C('@'),
38461   /* 52 - '.'     */      '.',    ':',    A('.'), '.',    ':',    C('@'),
38462   /* 53 - '/'     */      '-',    '_',    A('-'), '-',    '_',    C('@'),
38463   /* 54 - r. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
38464   /* 55 - '*'     */      '*',    '*',    A('*'), '*',    '*',    C('M'),
38465   /* 56 - ALT     */      ALT,    ALT,    ALT,    ALT,    ALT,    ALT,
38466   /* 57 - ' '     */      ' ',    ' ',    A(' '), ' ',    ' ',    C('@'),
38467   /* 58 - CapsLck */      CALOCK, CALOCK, CALOCK, CALOCK, CALOCK, CALOCK,
38468   /* 59 - F1      */      F1,     SF1,    AF1,    AF1,    ASF1,   CF1,
38469   /* 60 - F2      */      F2,     SF2,    AF2,    AF2,    ASF2,   CF2,
38470   /* 61 - F3      */      F3,     SF3,    AF3,    AF3,    ASF3,   CF3,
38471   /* 62 - F4      */      F4,     SF4,    AF4,    AF4,    ASF4,   CF4,
38472   /* 63 - F5      */      F5,     SF5,    AF5,    AF5,    ASF5,   CF5,
38473   /* 64 - F6      */      F6,     SF6,    AF6,    AF6,    ASF6,   CF6,
38474   /* 65 - F7      */      F7,     SF7,    AF7,    AF7,    ASF7,   CF7,
38475   /* 66 - F8      */      F8,     SF8,    AF8,    AF8,    ASF8,   CF8,
38476   /* 67 - F9      */      F9,     SF9,    AF9,    AF9,    ASF9,   CF9,
38477   /* 68 - F10     */      F10,    SF10,   AF10,   AF10,   ASF10,  CF10,
38478   /* 69 - NumLock */      NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,
38479   /* 70 - ScrLock */      SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,
38480   /* 71 - Home    */      HOME,   '7',    AHOME,  AHOME,  '7',    CHOME,
38481   /* 72 - CurUp   */      UP,     '8',    AUP,    AUP,    '8',    CUP,
38482   /* 73 - PgUp    */      PGUP,   '9',    APGUP,  APGUP,  '9',    CPGUP,
38483   /* 74 - '-'     */      NMIN,   '-',    ANMIN,  ANMIN,  '-',    CNMIN,
38484   /* 75 - Left    */      LEFT,   '4',    ALEFT,  ALEFT,  '4',    CLEFT,
38485   /* 76 - MID     */      MID,    '5',    AMID,   AMID,   '5',    CMID,
38486   /* 77 - Right   */      RIGHT,  '6',    ARIGHT, ARIGHT, '6',    CRIGHT,
38487   /* 78 - '+'     */      PLUS,   '+',    APLUS,  APLUS,  '+',    CPLUS,
38488   /* 79 - End     */      END,    '1',    AEND,   AEND,   '1',    CEND,
38489   /* 80 - Down    */      DOWN,   '2',    ADOWN,  ADOWN,  '2',    CDOWN,
38490   /* 81 - PgDown  */      PGDN,   '3',    APGDN,  APGDN,  '3',    CPGDN,
38491   /* 82 - Insert  */      INSRT,  '0',    AINSRT, AINSRT, '0',    CINSRT,
38492   /* 83 - Delete  */      0177,   '.',    A(0177),0177,   '.',    0177,
38493   /* 84 - Enter   */      C('M'), C('M'), CA('M'),C('M'), C('M'), C('J'),
38494   /* 85 - ???     */      0,      0,      0,      0,      0,      0,
38495   /* 86 - ???     */      '<',    '>',    A('<'), '<',    '>',    C('@'),
38496   /* 87 - F11     */      F11,    SF11,   AF11,   AF11,   ASF11,  CF11,
38497   /* 88 - F12     */      F12,    SF12,   AF12,   AF12,   ASF12,  CF12,
38498   /* 89 - ???     */      0,      0,      0,      0,      0,      0,
38499   /* 90 - ???     */      0,      0,      0,      0,      0,      0,
38500   /* 91 - ???     */      0,      0,      0,      0,      0,      0,
38501   /* 92 - ???     */      0,      0,      0,      0,      0,      0,
38502   /* 93 - ???     */      0,      0,      0,      0,      0,      0,
38503   /* 94 - ???     */      0,      0,      0,      0,      0,      0,
38504   /* 95 - ???     */      0,      0,      0,      0,      0,      0,
38505   /* 96 - EXT_KEY */      EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY,
38506   /* 97 - ???     */      0,      0,      0,      0,      0,      0,
38507   /* 98 - ???     */      0,      0,      0,      0,      0,      0,
38508   /* 99 - ???     */      0,      0,      0,      0,      0,      0,
38509   /*100 - ???     */      0,      0,      0,      0,      0,      0,
38510   /*101 - ???     */      0,      0,      0,      0,      0,      0,
38511   /*102 - ???     */      0,      0,      0,      0,      0,      0,
38512   /*103 - ???     */      0,      0,      0,      0,      0,      0,
38513   /*104 - ???     */      0,      0,      0,      0,      0,      0,
38514   /*105 - ???     */      0,      0,      0,      0,      0,      0,
```

```
38515    /*106 - ???     */      0,      0,      0,      0,      0,      0,
38516    /*107 - ???     */      0,      0,      0,      0,      0,      0,
38517    /*108 - ???     */      0,      0,      0,      0,      0,      0,
38518    /*109 - ???     */      0,      0,      0,      0,      0,      0,
38519    /*110 - ???     */      0,      0,      0,      0,      0,      0,
38520    /*111 - ???     */      0,      0,      0,      0,      0,      0,
38521    /*112 - ???     */      0,      0,      0,      0,      0,      0,
38522    /*113 - ???     */      0,      0,      0,      0,      0,      0,
38523    /*114 - ???     */      0,      0,      0,      0,      0,      0,
38524    /*115 - ???     */      0,      0,      0,      0,      0,      0,
38525    /*116 - ???     */      0,      0,      0,      0,      0,      0,
38526    /*117 - ???     */      0,      0,      0,      0,      0,      0,
38527    /*118 - ???     */      0,      0,      0,      0,      0,      0,
38528    /*119 - ???     */      0,      0,      0,      0,      0,      0,
38529    /*120 - ???     */      0,      0,      0,      0,      0,      0,
38530    /*121 - ???     */      0,      0,      0,      0,      0,      0,
38531    /*122 - ???     */      0,      0,      0,      0,      0,      0,
38532    /*123 - ???     */      0,      0,      0,      0,      0,      0,
38533    /*124 - ???     */      0,      0,      0,      0,      0,      0,
38534    /*125 - ???     */      0,      0,      0,      0,      0,      0,
38535    /*126 - ???     */      0,      0,      0,      0,      0,      0,
38536    /*127 - ???     */      0,      0,      0,      0,      0,      0
38537    };


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/keymaps/uk.src
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

38600    /* Keymap for standard UK keyboard.                    Author: Darren Mason */
38601
38602    u16_t keymap[NR_SCAN_CODES * MAP_COLS] = {
38603
38604    /* scan-code            !Shift  Shift   Alt1    Alt2    Alt+Sh  Ctrl    */
38605    /* ============================================================================ */
38606    /* 00 - none    */      0,      0,      0,      0,      0,      0,
38607    /* 01 - ESC     */      C('['), C('['), CA('['),CA('['),CA('['),C('['),
38608    /* 02 - '1'     */      '1',    '!',    A('1'), A('1'), A('!'), C('A'),
38609    /* 03 - '2'     */      '2',    '"',    A('2'), A('2'), A('@'), C('@'),
38610    /* 04 - '3'     */      '3',    156,    A('3'), A('3'), A(156), C('C'),
38611    /* 05 - '4'     */      '4',    '$',    A('4'), A('4'), A('$'), C('D'),
38612    /* 06 - '5'     */      '5',    '%',    A('5'), A('5'), A('%'), C('E'),
38613    /* 07 - '6'     */      '6',    '^',    A('6'), A('6'), A('^'), C('^'),
38614    /* 08 - '7'     */      '7',    '&',    A('7'), A('7'), A('&'), C('G'),
38615    /* 09 - '8'     */      '8',    '*',    A('8'), A('8'), A('*'), C('H'),
38616    /* 10 - '9'     */      '9',    '(',    A('9'), A('9'), A('('), C('I'),
38617    /* 11 - '0'     */      '0',    ')',    A('0'), A('0'), A(')'), C('@'),
38618    /* 12 - '-'     */      '-',    '_',    A('-'), A('-'), A('_'), C('_'),
38619    /* 13 - '='     */      '=',    '+',    A('='), A('='), A('+'), C('@'),
38620    /* 14 - BS      */      C('H'), C('H'), CA('H'),CA('H'),CA('H'),0177,
38621    /* 15 - TAB     */      C('I'), C('I'), CA('I'),CA('I'),CA('I'),C('I'),
38622    /* 16 - 'q'     */      L('q'), 'Q',    A('q'), A('q'), A('Q'), C('Q'),
38623    /* 17 - 'w'     */      L('w'), 'W',    A('w'), A('w'), A('W'), C('W'),
38624    /* 18 - 'e'     */      L('e'), 'E',    A('e'), A('e'), A('E'), C('E'),
38625    /* 19 - 'r'     */      L('r'), 'R',    A('r'), A('r'), A('R'), C('R'),
38626    /* 20 - 't'     */      L('t'), 'T',    A('t'), A('t'), A('T'), C('T'),
38627    /* 21 - 'y'     */      L('y'), 'Y',    A('y'), A('y'), A('Y'), C('Y'),
38628    /* 22 - 'u'     */      L('u'), 'U',    A('u'), A('u'), A('U'), C('U'),
38629    /* 23 - 'i'     */      L('i'), 'I',    A('i'), A('i'), A('I'), C('I'),
```

```
38630   /* 24 - 'o'     */      L('o'), 'O',    A('o'), A('o'), A('O'), C('O'),
38631   /* 25 - 'p'     */      L('p'), 'P',    A('p'), A('p'), A('P'), C('P'),
38632   /* 26 - '['     */      '[',    '{',    A('['), A('['), A('{'), C('['),
38633   /* 27 - ']'     */      ']',    '}',    A(']'), A(']'), A('}'), C(']'),
38634   /* 28 - CR/LF   */      C('M'), C('M'), CA('M'),CA('M'),CA('M'),C('J'),
38635   /* 29 - Ctrl    */      CTRL,   CTRL,   CTRL,   CTRL,   CTRL,   CTRL,
38636   /* 30 - 'a'     */      L('a'), 'A',    A('a'), A('a'), A('A'), C('A'),
38637   /* 31 - 's'     */      L('s'), 'S',    A('s'), A('s'), A('S'), C('S'),
38638   /* 32 - 'd'     */      L('d'), 'D',    A('d'), A('d'), A('D'), C('D'),
38639   /* 33 - 'f'     */      L('f'), 'F',    A('f'), A('f'), A('F'), C('F'),
38640   /* 34 - 'g'     */      L('g'), 'G',    A('g'), A('g'), A('G'), C('G'),
38641   /* 35 - 'h'     */      L('h'), 'H',    A('h'), A('h'), A('H'), C('H'),
38642   /* 36 - 'j'     */      L('j'), 'J',    A('j'), A('j'), A('J'), C('J'),
38643   /* 37 - 'k'     */      L('k'), 'K',    A('k'), A('k'), A('K'), C('K'),
38644   /* 38 - 'l'     */      L('l'), 'L',    A('l'), A('l'), A('L'), C('L'),
38645   /* 39 - ';'     */      ';',    ':',    A(';'), A(';'), A(':'), C('@'),
38646   /* 40 - '\''    */      '\'',   '@',    A('\''),A('\''),A('"'), C('@'),
38647   /* 41 - '´     */      '´',    '~',    A('´'), A('´'), A('~'), C('@'),
38648   /* 42 - l. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
38649   /* 43 - '#'     */      '#',    '~',    A('#'), A('#'), A('~'), C('#'),
38650   /* 44 - 'z'     */      L('z'), 'Z',    A('z'), A('z'), A('Z'), C('Z'),
38651   /* 45 - 'x'     */      L('x'), 'X',    A('x'), A('x'), A('X'), C('X'),
38652   /* 46 - 'c'     */      L('c'), 'C',    A('c'), A('c'), A('C'), C('C'),
38653   /* 47 - 'v'     */      L('v'), 'V',    A('v'), A('v'), A('V'), C('V'),
38654   /* 48 - 'b'     */      L('b'), 'B',    A('b'), A('b'), A('B'), C('B'),
38655   /* 49 - 'n'     */      L('n'), 'N',    A('n'), A('n'), A('N'), C('N'),
38656   /* 50 - 'm'     */      L('m'), 'M',    A('m'), A('m'), A('M'), C('M'),
38657   /* 51 - ','     */      ',',    '<',    A(','), A(','), A('<'), C('@'),
38658   /* 52 - '.'     */      '.',    '>',    A('.'), A('.'), A('>'), C('@'),
38659   /* 53 - '/'     */      '/',    '?',    A('/'), A('/'), A('?'), C('@'),
38660   /* 54 - r. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
38661   /* 55 - '*'     */      '*',    '*',    A('*'), A('*'), A('*'), C('@'),
38662   /* 56 - ALT     */      ALT,    ALT,    ALT,    ALT,    ALT,    ALT,
38663   /* 57 - ' '     */      ' ',    ' ',    A(' '), A(' '), A(' '), C('@'),
38664   /* 58 - CapsLck */      CALOCK, CALOCK, CALOCK, CALOCK, CALOCK, CALOCK,
38665   /* 59 - F1      */      F1,     SF1,    AF1,    AF1,    ASF1,   CF1,
38666   /* 60 - F2      */      F2,     SF2,    AF2,    AF2,    ASF2,   CF2,
38667   /* 61 - F3      */      F3,     SF3,    AF3,    AF3,    ASF3,   CF3,
38668   /* 62 - F4      */      F4,     SF4,    AF4,    AF4,    ASF4,   CF4,
38669   /* 63 - F5      */      F5,     SF5,    AF5,    AF5,    ASF5,   CF5,
38670   /* 64 - F6      */      F6,     SF6,    AF6,    AF6,    ASF6,   CF6,
38671   /* 65 - F7      */      F7,     SF7,    AF7,    AF7,    ASF7,   CF7,
38672   /* 66 - F8      */      F8,     SF8,    AF8,    AF8,    ASF8,   CF8,
38673   /* 67 - F9      */      F9,     SF9,    AF9,    AF9,    ASF9,   CF9,
38674   /* 68 - F10     */      F10,    SF10,   AF10,   AF10,   ASF10,  CF10,
38675   /* 69 - NumLock */      NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,
38676   /* 70 - ScrLock */      SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,
38677   /* 71 - Home    */      HOME,   '7',    AHOME,  AHOME,  A('7'), CHOME,
38678   /* 72 - CurUp   */      UP,     '8',    AUP,    AUP,    A('8'), CUP,
38679   /* 73 - PgUp    */      PGUP,   '9',    APGUP,  APGUP,  A('9'), CPGUP,
38680   /* 74 - '-'     */      NMIN,   '-',    ANMIN,  ANMIN,  A('-'), CNMIN,
38681   /* 75 - Left    */      LEFT,   '4',    ALEFT,  ALEFT,  A('4'), CLEFT,
38682   /* 76 - MID     */      MID,    '5',    AMID,   AMID,   A('5'), CMID,
38683   /* 77 - Right   */      RIGHT,  '6',    ARIGHT, ARIGHT, A('6'), CRIGHT,
38684   /* 78 - '+'     */      PLUS,   '+',    APLUS,  APLUS,  A('+'), CPLUS,
38685   /* 79 - End     */      END,    '1',    AEND,   AEND,   A('1'), CEND,
38686   /* 80 - Down    */      DOWN,   '2',    ADOWN,  ADOWN,  A('2'), CDOWN,
38687   /* 81 - PgDown  */      PGDN,   '3',    APGDN,  APGDN,  A('3'), CPGDN,
38688   /* 82 - Insert  */      INSRT,  '0',    AINSRT, AINSRT, A('0'), CINSRT,
38689   /* 83 - Delete  */      0177,   '.',    A(0177),A(0177),A('.'), 0177,
```

```
38690   /* 84 - Enter   */      C('M'), C('M'), CA('M'),CA('M'),CA('M'),C('J'),
38691   /* 85 - ???     */      0,      0,      0,      0,      0,      0,
38692   /* 86 - ???     */      '\\',   '|',    A('\\'),A('|'), A('|'), C('@'),
38693   /* 87 - F11     */      F11,    SF11,   AF11,   AF11,   ASF11,  CF11,
38694   /* 88 - F12     */      F12,    SF12,   AF12,   AF12,   ASF12,  CF12,
38695   /* 89 - ???     */      0,      0,      0,      0,      0,      0,
38696   /* 90 - ???     */      0,      0,      0,      0,      0,      0,
38697   /* 91 - ???     */      0,      0,      0,      0,      0,      0,
38698   /* 92 - ???     */      0,      0,      0,      0,      0,      0,
38699   /* 93 - ???     */      0,      0,      0,      0,      0,      0,
38700   /* 94 - ???     */      0,      0,      0,      0,      0,      0,
38701   /* 95 - ???     */      0,      0,      0,      0,      0,      0,
38702   /* 96 - EXT_KEY */      EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY,
38703   /* 97 - ???     */      0,      0,      0,      0,      0,      0,
38704   /* 98 - ???     */      0,      0,      0,      0,      0,      0,
38705   /* 99 - ???     */      0,      0,      0,      0,      0,      0,
38706   /*100 - ???     */      0,      0,      0,      0,      0,      0,
38707   /*101 - ???     */      0,      0,      0,      0,      0,      0,
38708   /*102 - ???     */      0,      0,      0,      0,      0,      0,
38709   /*103 - ???     */      0,      0,      0,      0,      0,      0,
38710   /*104 - ???     */      0,      0,      0,      0,      0,      0,
38711   /*105 - ???     */      0,      0,      0,      0,      0,      0,
38712   /*106 - ???     */      0,      0,      0,      0,      0,      0,
38713   /*107 - ???     */      0,      0,      0,      0,      0,      0,
38714   /*108 - ???     */      0,      0,      0,      0,      0,      0,
38715   /*109 - ???     */      0,      0,      0,      0,      0,      0,
38716   /*110 - ???     */      0,      0,      0,      0,      0,      0,
38717   /*111 - ???     */      0,      0,      0,      0,      0,      0,
38718   /*112 - ???     */      0,      0,      0,      0,      0,      0,
38719   /*113 - ???     */      0,      0,      0,      0,      0,      0,
38720   /*114 - ???     */      0,      0,      0,      0,      0,      0,
38721   /*115 - ???     */      0,      0,      0,      0,      0,      0,
38722   /*116 - ???     */      0,      0,      0,      0,      0,      0,
38723   /*117 - ???     */      0,      0,      0,      0,      0,      0,
38724   /*118 - ???     */      0,      0,      0,      0,      0,      0,
38725   /*119 - ???     */      0,      0,      0,      0,      0,      0,
38726   /*120 - ???     */      0,      0,      0,      0,      0,      0,
38727   /*121 - ???     */      0,      0,      0,      0,      0,      0,
38728   /*122 - ???     */      0,      0,      0,      0,      0,      0,
38729   /*123 - ???     */      0,      0,      0,      0,      0,      0,
38730   /*124 - ???     */      0,      0,      0,      0,      0,      0,
38731   /*125 - ???     */      0,      0,      0,      0,      0,      0,
38732   /*126 - ???     */      0,      0,      0,      0,      0,      0,
38733   /*127 - ???     */      0,      0,      0,      0,      0,      0
38734   };
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/kernel/keymaps/us-std.src
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

38800   /* Keymap for US MF-2 keyboard. */
38801
38802   u16_t keymap[NR_SCAN_CODES * MAP_COLS] = {
38803
38804   /* scan-code            !Shift  Shift   Alt1    Alt2    Alt+Sh  Ctrl    */
38805   /* ============================================================== */
38806   /* 00 - none    */      0,      0,      0,      0,      0,      0,
38807   /* 01 - ESC     */      C('['), C('['), CA('['),CA('['),CA('['),C('['),
38808   /* 02 - '1'     */      '1',    '!',    A('1'), A('1'), A('!'), C('A'),
38809   /* 03 - '2'     */      '2',    '@',    A('2'), A('2'), A('@'), C('@'),
38810   /* 04 - '3'     */      '3',    '#',    A('3'), A('3'), A('#'), C('C'),
38811   /* 05 - '4'     */      '4',    '$',    A('4'), A('4'), A('$'), C('D'),
38812   /* 06 - '5'     */      '5',    '%',    A('5'), A('5'), A('%'), C('E'),
38813   /* 07 - '6'     */      '6',    '^',    A('6'), A('6'), A('^'), C('^'),
38814   /* 08 - '7'     */      '7',    '&',    A('7'), A('7'), A('&'), C('G'),
38815   /* 09 - '8'     */      '8',    '*',    A('8'), A('8'), A('*'), C('H'),
38816   /* 10 - '9'     */      '9',    '(',    A('9'), A('9'), A('('), C('I'),
38817   /* 11 - '0'     */      '0',    ')',    A('0'), A('0'), A(')'), C('@'),
38818   /* 12 - '-'     */      '-',    '_',    A('-'), A('-'), A('_'), C('_'),
38819   /* 13 - '='     */      '=',    '+',    A('='), A('='), A('+'), C('@'),
38820   /* 14 - BS      */      C('H'), C('H'), CA('H'),CA('H'),CA('H'),0177,
38821   /* 15 - TAB     */      C('I'), C('I'), CA('I'),CA('I'),CA('I'),C('I'),
38822   /* 16 - 'q'     */      L('q'), 'Q',    A('q'), A('q'), A('Q'), C('Q'),
38823   /* 17 - 'w'     */      L('w'), 'W',    A('w'), A('w'), A('W'), C('W'),
38824   /* 18 - 'e'     */      L('e'), 'E',    A('e'), A('e'), A('E'), C('E'),
38825   /* 19 - 'r'     */      L('r'), 'R',    A('r'), A('r'), A('R'), C('R'),
38826   /* 20 - 't'     */      L('t'), 'T',    A('t'), A('t'), A('T'), C('T'),
38827   /* 21 - 'y'     */      L('y'), 'Y',    A('y'), A('y'), A('Y'), C('Y'),
38828   /* 22 - 'u'     */      L('u'), 'U',    A('u'), A('u'), A('U'), C('U'),
38829   /* 23 - 'i'     */      L('i'), 'I',    A('i'), A('i'), A('I'), C('I'),
38830   /* 24 - 'o'     */      L('o'), 'O',    A('o'), A('o'), A('O'), C('O'),
38831   /* 25 - 'p'     */      L('p'), 'P',    A('p'), A('p'), A('P'), C('P'),
38832   /* 26 - '['     */      '[',    '{',    A('['), A('['), A('{'), C('['),
38833   /* 27 - ']'     */      ']',    '}',    A(']'), A(']'), A('}'), C(']'),
38834   /* 28 - CR/LF   */      C('M'), C('M'), CA('M'),CA('M'),CA('M'),C('J'),
38835   /* 29 - Ctrl    */      CTRL,   CTRL,   CTRL,   CTRL,   CTRL,   CTRL,
38836   /* 30 - 'a'     */      L('a'), 'A',    A('a'), A('a'), A('A'), C('A'),
38837   /* 31 - 's'     */      L('s'), 'S',    A('s'), A('s'), A('S'), C('S'),
38838   /* 32 - 'd'     */      L('d'), 'D',    A('d'), A('d'), A('D'), C('D'),
38839   /* 33 - 'f'     */      L('f'), 'F',    A('f'), A('f'), A('F'), C('F'),
38840   /* 34 - 'g'     */      L('g'), 'G',    A('g'), A('g'), A('G'), C('G'),
38841   /* 35 - 'h'     */      L('h'), 'H',    A('h'), A('h'), A('H'), C('H'),
38842   /* 36 - 'j'     */      L('j'), 'J',    A('j'), A('j'), A('J'), C('J'),
38843   /* 37 - 'k'     */      L('k'), 'K',    A('k'), A('k'), A('K'), C('K'),
38844   /* 38 - 'l'     */      L('l'), 'L',    A('l'), A('l'), A('L'), C('L'),
38845   /* 39 - ';'     */      ';',    ':',    A(';'), A(';'), A(':'), C('@'),
38846   /* 40 - '\''    */      '\'',   '"',    A('\''),A('\''),A('"'), C('@'),
38847   /* 41 - '´     */      '´',    '~',    A('´'), A('´'), A('~'), C('@'),
38848   /* 42 - l. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
38849   /* 43 - '\\'    */      '\\',   '|',    A('\\'),A('\\'),A('|'), C('\\'),
38850   /* 44 - 'z'     */      L('z'), 'Z',    A('z'), A('z'), A('Z'), C('Z'),
38851   /* 45 - 'x'     */      L('x'), 'X',    A('x'), A('x'), A('X'), C('X'),
38852   /* 46 - 'c'     */      L('c'), 'C',    A('c'), A('c'), A('C'), C('C'),
38853   /* 47 - 'v'     */      L('v'), 'V',    A('v'), A('v'), A('V'), C('V'),
38854   /* 48 - 'b'     */      L('b'), 'B',    A('b'), A('b'), A('B'), C('B'),
```

```
38855   /* 49 - 'n'     */      L('n'), 'N',    A('n'), A('n'), A('N'), C('N'),
38856   /* 50 - 'm'     */      L('m'), 'M',    A('m'), A('m'), A('M'), C('M'),
38857   /* 51 - ','     */      ',',    '<',    A(','), A(','), A('<'), C('@'),
38858   /* 52 - '.'     */      '.',    '>',    A('.'), A('.'), A('>'), C('@'),
38859   /* 53 - '/'     */      '/',    '?',    A('/'), A('/'), A('?'), C('@'),
38860   /* 54 - r. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
38861   /* 55 - '*'     */      '*',    '*',    A('*'), A('*'), A('*'), C('@'),
38862   /* 56 - ALT     */      ALT,    ALT,    ALT,    ALT,    ALT,    ALT,
38863   /* 57 - ' '     */      ' ',    ' ',    A(' '), A(' '), A(' '), C('@'),
38864   /* 58 - CapsLck */      CALOCK, CALOCK, CALOCK, CALOCK, CALOCK, CALOCK,
38865   /* 59 - F1      */      F1,     SF1,    AF1,    AF1,    ASF1,   CF1,
38866   /* 60 - F2      */      F2,     SF2,    AF2,    AF2,    ASF2,   CF2,
38867   /* 61 - F3      */      F3,     SF3,    AF3,    AF3,    ASF3,   CF3,
38868   /* 62 - F4      */      F4,     SF4,    AF4,    AF4,    ASF4,   CF4,
38869   /* 63 - F5      */      F5,     SF5,    AF5,    AF5,    ASF5,   CF5,
38870   /* 64 - F6      */      F6,     SF6,    AF6,    AF6,    ASF6,   CF6,
38871   /* 65 - F7      */      F7,     SF7,    AF7,    AF7,    ASF7,   CF7,
38872   /* 66 - F8      */      F8,     SF8,    AF8,    AF8,    ASF8,   CF8,
38873   /* 67 - F9      */      F9,     SF9,    AF9,    AF9,    ASF9,   CF9,
38874   /* 68 - F10     */      F10,    SF10,   AF10,   AF10,   ASF10,  CF10,
38875   /* 69 - NumLock */      NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,
38876   /* 70 - ScrLock */      SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,
38877   /* 71 - Home    */      HOME,   '7',    AHOME,  AHOME,  A('7'), CHOME,
38878   /* 72 - CurUp   */      UP,     '8',    AUP,    AUP,    A('8'), CUP,
38879   /* 73 - PgUp    */      PGUP,   '9',    APGUP,  APGUP,  A('9'), CPGUP,
38880   /* 74 - '-'     */      NMIN,   '-',    ANMIN,  ANMIN,  A('-'), CNMIN,
38881   /* 75 - Left    */      LEFT,   '4',    ALEFT,  ALEFT,  A('4'), CLEFT,
38882   /* 76 - MID     */      MID,    '5',    AMID,   AMID,   A('5'), CMID,
38883   /* 77 - Right   */      RIGHT,  '6',    ARIGHT, ARIGHT, A('6'), CRIGHT,
38884   /* 78 - '+'     */      PLUS,   '+',    APLUS,  APLUS,  A('+'), CPLUS,
38885   /* 79 - End     */      END,    '1',    AEND,   AEND,   A('1'), CEND,
38886   /* 80 - Down    */      DOWN,   '2',    ADOWN,  ADOWN,  A('2'), CDOWN,
38887   /* 81 - PgDown  */      PGDN,   '3',    APGDN,  APGDN,  A('3'), CPGDN,
38888   /* 82 - Insert  */      INSRT,  '0',    AINSRT, AINSRT, A('0'), CINSRT,
38889   /* 83 - Delete  */      0177,   '.',    A(0177),A(0177),A('.'), 0177,
38890   /* 84 - Enter   */      C('M'), C('M'), CA('M'),CA('M'),CA('M'),C('J'),
38891   /* 85 - ???     */      0,      0,      0,      0,      0,      0,
38892   /* 86 - ???     */      '<',    '>',    A('<'), A('|'), A('>'), C('@'),
38893   /* 87 - F11     */      F11,    SF11,   AF11,   AF11,   ASF11,  CF11,
38894   /* 88 - F12     */      F12,    SF12,   AF12,   AF12,   ASF12,  CF12,
38895   /* 89 - ???     */      0,      0,      0,      0,      0,      0,
38896   /* 90 - ???     */      0,      0,      0,      0,      0,      0,
38897   /* 91 - ???     */      0,      0,      0,      0,      0,      0,
38898   /* 92 - ???     */      0,      0,      0,      0,      0,      0,
38899   /* 93 - ???     */      0,      0,      0,      0,      0,      0,
38900   /* 94 - ???     */      0,      0,      0,      0,      0,      0,
38901   /* 95 - ???     */      0,      0,      0,      0,      0,      0,
38902   /* 96 - EXT_KEY */      EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY,
38903   /* 97 - ???     */      0,      0,      0,      0,      0,      0,
38904   /* 98 - ???     */      0,      0,      0,      0,      0,      0,
38905   /* 99 - ???     */      0,      0,      0,      0,      0,      0,
38906   /*100 - ???     */      0,      0,      0,      0,      0,      0,
38907   /*101 - ???     */      0,      0,      0,      0,      0,      0,
38908   /*102 - ???     */      0,      0,      0,      0,      0,      0,
38909   /*103 - ???     */      0,      0,      0,      0,      0,      0,
38910   /*104 - ???     */      0,      0,      0,      0,      0,      0,
38911   /*105 - ???     */      0,      0,      0,      0,      0,      0,
38912   /*106 - ???     */      0,      0,      0,      0,      0,      0,
38913   /*107 - ???     */      0,      0,      0,      0,      0,      0,
38914   /*108 - ???     */      0,      0,      0,      0,      0,      0,
```

```
38915    /*109 - ???     */      0,      0,      0,      0,      0,      0,
38916    /*110 - ???     */      0,      0,      0,      0,      0,      0,
38917    /*111 - ???     */      0,      0,      0,      0,      0,      0,
38918    /*112 - ???     */      0,      0,      0,      0,      0,      0,
38919    /*113 - ???     */      0,      0,      0,      0,      0,      0,
38920    /*114 - ???     */      0,      0,      0,      0,      0,      0,
38921    /*115 - ???     */      0,      0,      0,      0,      0,      0,
38922    /*116 - ???     */      0,      0,      0,      0,      0,      0,
38923    /*117 - ???     */      0,      0,      0,      0,      0,      0,
38924    /*118 - ???     */      0,      0,      0,      0,      0,      0,
38925    /*119 - ???     */      0,      0,      0,      0,      0,      0,
38926    /*120 - ???     */      0,      0,      0,      0,      0,      0,
38927    /*121 - ???     */      0,      0,      0,      0,      0,      0,
38928    /*122 - ???     */      0,      0,      0,      0,      0,      0,
38929    /*123 - ???     */      0,      0,      0,      0,      0,      0,
38930    /*124 - ???     */      0,      0,      0,      0,      0,      0,
38931    /*125 - ???     */      0,      0,      0,      0,      0,      0,
38932    /*126 - ???     */      0,      0,      0,      0,      0,      0,
38933    /*127 - ???     */      0,      0,      0,      0,      0,      0
38934    };
```

++

src/kernel/keymaps/us-swap.src

++

```
39000    /* Keymap for US MF-2 keyboard with the Caps Lock and Control key swapped. */
39001
39002    u16_t keymap[NR_SCAN_CODES * MAP_COLS] = {
39003
39004    /* scan-code            !Shift  Shift   Alt1    Alt2    Alt+Sh  Ctrl    */
39005    /* ============================================================== */
39006    /* 00 - none    */      0,      0,      0,      0,      0,      0,
39007    /* 01 - ESC     */      C('['), C('['), CA('['),CA('['),CA('['),C('['),
39008    /* 02 - '1'     */      '1',    '!',    A('1'), A('1'), A('!'), C('A'),
39009    /* 03 - '2'     */      '2',    '@',    A('2'), A('2'), A('@'), C('@'),
39010    /* 04 - '3'     */      '3',    '#',    A('3'), A('3'), A('#'), C('C'),
39011    /* 05 - '4'     */      '4',    '$',    A('4'), A('4'), A('$'), C('D'),
39012    /* 06 - '5'     */      '5',    '%',    A('5'), A('5'), A('%'), C('E'),
39013    /* 07 - '6'     */      '6',    '^',    A('6'), A('6'), A('^'), C('^'),
39014    /* 08 - '7'     */      '7',    '&',    A('7'), A('7'), A('&'), C('G'),
39015    /* 09 - '8'     */      '8',    '*',    A('8'), A('8'), A('*'), C('H'),
39016    /* 10 - '9'     */      '9',    '(',    A('9'), A('9'), A('('), C('I'),
39017    /* 11 - '0'     */      '0',    ')',    A('0'), A('0'), A(')'), C('@'),
39018    /* 12 - '-'     */      '-',    '_',    A('-'), A('-'), A('_'), C('_'),
39019    /* 13 - '='     */      '=',    '+',    A('='), A('='), A('+'), C('@'),
39020    /* 14 - BS      */      C('H'), C('H'), CA('H'),CA('H'),CA('H'),0177,
39021    /* 15 - TAB     */      C('I'), C('I'), CA('I'),CA('I'),CA('I'),C('I'),
39022    /* 16 - 'q'     */      L('q'), 'Q',    A('q'), A('q'), A('Q'), C('Q'),
39023    /* 17 - 'w'     */      L('w'), 'W',    A('w'), A('w'), A('W'), C('W'),
39024    /* 18 - 'e'     */      L('e'), 'E',    A('e'), A('e'), A('E'), C('E'),
39025    /* 19 - 'r'     */      L('r'), 'R',    A('r'), A('r'), A('R'), C('R'),
39026    /* 20 - 't'     */      L('t'), 'T',    A('t'), A('t'), A('T'), C('T'),
39027    /* 21 - 'y'     */      L('y'), 'Y',    A('y'), A('y'), A('Y'), C('Y'),
39028    /* 22 - 'u'     */      L('u'), 'U',    A('u'), A('u'), A('U'), C('U'),
39029    /* 23 - 'i'     */      L('i'), 'I',    A('i'), A('i'), A('I'), C('I'),
```

```
39030   /* 24 - 'o'     */      L('o'), 'O',    A('o'), A('o'), A('O'), C('O'),
39031   /* 25 - 'p'     */      L('p'), 'P',    A('p'), A('p'), A('P'), C('P'),
39032   /* 26 - '['     */      '[',    '{',    A('['), A('['), A('{'), C('['),
39033   /* 27 - ']'     */      ']',    '}',    A(']'), A(']'), A('}'), C(']'),
39034   /* 28 - CR/LF   */      C('M'), C('M'), CA('M'),CA('M'),CA('M'),C('J'),
39035   /* 29 - Ctrl    */      CALOCK, CALOCK, CALOCK, CALOCK, CALOCK, CALOCK,
39036   /* 30 - 'a'     */      L('a'), 'A',    A('a'), A('a'), A('A'), C('A'),
39037   /* 31 - 's'     */      L('s'), 'S',    A('s'), A('s'), A('S'), C('S'),
39038   /* 32 - 'd'     */      L('d'), 'D',    A('d'), A('d'), A('D'), C('D'),
39039   /* 33 - 'f'     */      L('f'), 'F',    A('f'), A('f'), A('F'), C('F'),
39040   /* 34 - 'g'     */      L('g'), 'G',    A('g'), A('g'), A('G'), C('G'),
39041   /* 35 - 'h'     */      L('h'), 'H',    A('h'), A('h'), A('H'), C('H'),
39042   /* 36 - 'j'     */      L('j'), 'J',    A('j'), A('j'), A('J'), C('J'),
39043   /* 37 - 'k'     */      L('k'), 'K',    A('k'), A('k'), A('K'), C('K'),
39044   /* 38 - 'l'     */      L('l'), 'L',    A('l'), A('l'), A('L'), C('L'),
39045   /* 39 - ';'     */      ';',    ':',    A(';'), A(';'), A(':'), C('@'),
39046   /* 40 - '\''    */      '\'',   '"',    A('\''),A('\''),A('"'), C('@'),
39047   /* 41 - '´     */      '´',    '~',    A('´'), A('´'), A('~'), C('@'),
39048   /* 42 - l. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
39049   /* 43 - '\\'    */      '\\',   '|',    A('\\'),A('\\'),A('|'), C('\\'),
39050   /* 44 - 'z'     */      L('z'), 'Z',    A('z'), A('z'), A('Z'), C('Z'),
39051   /* 45 - 'x'     */      L('x'), 'X',    A('x'), A('x'), A('X'), C('X'),
39052   /* 46 - 'c'     */      L('c'), 'C',    A('c'), A('c'), A('C'), C('C'),
39053   /* 47 - 'v'     */      L('v'), 'V',    A('v'), A('v'), A('V'), C('V'),
39054   /* 48 - 'b'     */      L('b'), 'B',    A('b'), A('b'), A('B'), C('B'),
39055   /* 49 - 'n'     */      L('n'), 'N',    A('n'), A('n'), A('N'), C('N'),
39056   /* 50 - 'm'     */      L('m'), 'M',    A('m'), A('m'), A('M'), C('M'),
39057   /* 51 - ','     */      ',',    '<',    A(','), A(','), A('<'), C('@'),
39058   /* 52 - '.'     */      '.',    '>',    A('.'), A('.'), A('>'), C('@'),
39059   /* 53 - '/'     */      '/',    '?',    A('/'), A('/'), A('?'), C('@'),
39060   /* 54 - r. SHIFT*/      SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,  SHIFT,
39061   /* 55 - '*'     */      '*',    '*',    A('*'), A('*'), A('*'), C('@'),
39062   /* 56 - ALT     */      ALT,    ALT,    ALT,    ALT,    ALT,    ALT,
39063   /* 57 - ' '     */      ' ',    ' ',    A(' '), A(' '), A(' '), C('@'),
39064   /* 58 - CapsLck */      CTRL,   CTRL,   CTRL,   CTRL,   CTRL,   CTRL,
39065   /* 59 - F1      */      F1,     SF1,    AF1,    AF1,    ASF1,   CF1,
39066   /* 60 - F2      */      F2,     SF2,    AF2,    AF2,    ASF2,   CF2,
39067   /* 61 - F3      */      F3,     SF3,    AF3,    AF3,    ASF3,   CF3,
39068   /* 62 - F4      */      F4,     SF4,    AF4,    AF4,    ASF4,   CF4,
39069   /* 63 - F5      */      F5,     SF5,    AF5,    AF5,    ASF5,   CF5,
39070   /* 64 - F6      */      F6,     SF6,    AF6,    AF6,    ASF6,   CF6,
39071   /* 65 - F7      */      F7,     SF7,    AF7,    AF7,    ASF7,   CF7,
39072   /* 66 - F8      */      F8,     SF8,    AF8,    AF8,    ASF8,   CF8,
39073   /* 67 - F9      */      F9,     SF9,    AF9,    AF9,    ASF9,   CF9,
39074   /* 68 - F10     */      F10,    SF10,   AF10,   AF10,   ASF10,  CF10,
39075   /* 69 - NumLock */      NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,  NLOCK,
39076   /* 70 - ScrLock */      SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,  SLOCK,
39077   /* 71 - Home    */      HOME,   '7',    AHOME,  AHOME,  A('7'), CHOME,
39078   /* 72 - CurUp   */      UP,     '8',    AUP,    AUP,    A('8'), CUP,
39079   /* 73 - PgUp    */      PGUP,   '9',    APGUP,  APGUP,  A('9'), CPGUP,
39080   /* 74 - '-'     */      NMIN,   '-',    ANMIN,  ANMIN,  A('-'), CNMIN,
39081   /* 75 - Left    */      LEFT,   '4',    ALEFT,  ALEFT,  A('4'), CLEFT,
39082   /* 76 - MID     */      MID,    '5',    AMID,   AMID,   A('5'), CMID,
39083   /* 77 - Right   */      RIGHT,  '6',    ARIGHT, ARIGHT, A('6'), CRIGHT,
39084   /* 78 - '+'     */      PLUS,   '+',    APLUS,  APLUS,  A('+'), CPLUS,
39085   /* 79 - End     */      END,    '1',    AEND,   AEND,   A('1'), CEND,
39086   /* 80 - Down    */      DOWN,   '2',    ADOWN,  ADOWN,  A('2'), CDOWN,
39087   /* 81 - PgDown  */      PGDN,   '3',    APGDN,  APGDN,  A('3'), CPGDN,
39088   /* 82 - Insert  */      INSRT,  '0',    AINSRT, AINSRT, A('0'), CINSRT,
39089   /* 83 - Delete  */      0177,   '.',    A(0177),A(0177),A('.'), 0177,
```

```
39090    /* 84 - Enter   */      C('M'), C('M'), CA('M'),CA('M'),CA('M'),C('J'),
39091    /* 85 - ???     */      0,      0,      0,      0,      0,      0,
39092    /* 86 - ???     */      '<',    '>',    A('<'), A('|'), A('>'), C('@'),
39093    /* 87 - F11     */      F11,    SF11,   AF11,   AF11,   ASF11,  CF11,
39094    /* 88 - F12     */      F12,    SF12,   AF12,   AF12,   ASF12,  CF12,
39095    /* 89 - ???     */      0,      0,      0,      0,      0,      0,
39096    /* 90 - ???     */      0,      0,      0,      0,      0,      0,
39097    /* 91 - ???     */      0,      0,      0,      0,      0,      0,
39098    /* 92 - ???     */      0,      0,      0,      0,      0,      0,
39099    /* 93 - ???     */      0,      0,      0,      0,      0,      0,
39100    /* 94 - ???     */      0,      0,      0,      0,      0,      0,
39101    /* 95 - ???     */      0,      0,      0,      0,      0,      0,
39102    /* 96 - EXT_KEY */      EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY, EXTKEY,
39103    /* 97 - ???     */      0,      0,      0,      0,      0,      0,
39104    /* 98 - ???     */      0,      0,      0,      0,      0,      0,
39105    /* 99 - ???     */      0,      0,      0,      0,      0,      0,
39106    /*100 - ???     */      0,      0,      0,      0,      0,      0,
39107    /*101 - ???     */      0,      0,      0,      0,      0,      0,
39108    /*102 - ???     */      0,      0,      0,      0,      0,      0,
39109    /*103 - ???     */      0,      0,      0,      0,      0,      0,
39110    /*104 - ???     */      0,      0,      0,      0,      0,      0,
39111    /*105 - ???     */      0,      0,      0,      0,      0,      0,
39112    /*106 - ???     */      0,      0,      0,      0,      0,      0,
39113    /*107 - ???     */      0,      0,      0,      0,      0,      0,
39114    /*108 - ???     */      0,      0,      0,      0,      0,      0,
39115    /*109 - ???     */      0,      0,      0,      0,      0,      0,
39116    /*110 - ???     */      0,      0,      0,      0,      0,      0,
39117    /*111 - ???     */      0,      0,      0,      0,      0,      0,
39118    /*112 - ???     */      0,      0,      0,      0,      0,      0,
39119    /*113 - ???     */      0,      0,      0,      0,      0,      0,
39120    /*114 - ???     */      0,      0,      0,      0,      0,      0,
39121    /*115 - ???     */      0,      0,      0,      0,      0,      0,
39122    /*116 - ???     */      0,      0,      0,      0,      0,      0,
39123    /*117 - ???     */      0,      0,      0,      0,      0,      0,
39124    /*118 - ???     */      0,      0,      0,      0,      0,      0,
39125    /*119 - ???     */      0,      0,      0,      0,      0,      0,
39126    /*120 - ???     */      0,      0,      0,      0,      0,      0,
39127    /*121 - ???     */      0,      0,      0,      0,      0,      0,
39128    /*122 - ???     */      0,      0,      0,      0,      0,      0,
39129    /*123 - ???     */      0,      0,      0,      0,      0,      0,
39130    /*124 - ???     */      0,      0,      0,      0,      0,      0,
39131    /*125 - ???     */      0,      0,      0,      0,      0,      0,
39132    /*126 - ???     */      0,      0,      0,      0,      0,      0,
39133    /*127 - ???     */      0,      0,      0,      0,      0,      0
39134    };
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/mm/const.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

39200    /* Constants used by the Memory Manager. */
39201
39202    #define NO_MEM ((phys_clicks) 0)  /* returned by alloc_mem() with mem is up */
39203
39204    #if (CHIP == INTEL && _WORD_SIZE == 2)
39205    /* These definitions are used in size_ok and are not needed for 386.
39206     * The 386 segment granularity is 1 for segments smaller than 1M and 4096
39207     * above that.
39208     */
39209    #define PAGE_SIZE         16    /* how many bytes in a page (s.b.HCLICK_SIZE)*/
39210    #define MAX_PAGES       4096    /* how many pages in the virtual addr space */
39211    #endif
39212
39213    #define printf        printk
39214
39215    #define INIT_PID           1    /* init's process id number */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/mm/glo.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

39300    /* EXTERN should be extern except in table.c */
39301    #ifdef _TABLE
39302    #undef EXTERN
39303    #define EXTERN
39304    #endif
39305
39306    /* Global variables. */
39307    EXTERN struct mproc *mp;        /* ptr to 'mproc' slot of current process */
39308    EXTERN int dont_reply;          /* normally 0; set to 1 to inhibit reply */
39309    EXTERN int procs_in_use;        /* how many processes are marked as IN_USE */
39310
39311    /* The parameters of the call are kept here. */
39312    EXTERN message mm_in;           /* the incoming message itself is kept here. */
39313    EXTERN message mm_out;          /* the reply message is built up here. */
39314    EXTERN int who;                 /* caller's proc number */
39315    EXTERN int mm_call;             /* system call number */
39316
39317    /* The following variables are used for returning results to the caller. */
39318    EXTERN int err_code;            /* temporary storage for error number */
39319    EXTERN int result2;             /* secondary result */
39320    EXTERN char *res_ptr;           /* result, if pointer */
39321
39322    extern _PROTOTYPE (int (*call_vec[]), (void) ); /* system call handlers */
39323    extern char core_name[];        /* file name where core images are produced */
39324    EXTERN sigset_t core_sset;      /* which signals cause core images */
```

++
src/mm/mm.h
++

```
39400    /* This is the master header for mm.  It includes some other files
39401     * and defines the principal constants.
39402     */
39403    #define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
39404    #define _MINIX             1    /* tell headers to include MINIX stuff */
39405    #define _SYSTEM            1    /* tell headers that this is the kernel */
39406
39407    /* The following are so basic, all the *.c files get them automatically. */
39408    #include <minix/config.h>       /* MUST be first */
39409    #include <ansi.h>               /* MUST be second */
39410    #include <sys/types.h>
39411    #include <minix/const.h>
39412    #include <minix/type.h>
39413
39414    #include <fcntl.h>
39415    #include <unistd.h>
39416    #include <minix/syslib.h>
39417
39418    #include <limits.h>
39419    #include <errno.h>
39420
39421    #include "const.h"
39422    #include "type.h"
39423    #include "proto.h"
39424    #include "glo.h"
```

++
src/mm/mproc.h
++

```
39500    /* This table has one slot per process.  It contains all the memory management
39501     * information for each process.  Among other things, it defines the text, data
39502     * and stack segments, uids and gids, and various flags.  The kernel and file
39503     * systems have tables that are also indexed by process, with the contents
39504     * of corresponding slots referring to the same process in all three.
39505     */
39506
39507    EXTERN struct mproc {
39508      struct mem_map mp_seg[NR_SEGS];/* points to text, data, stack */
39509      char mp_exitstatus;           /* storage for status when process exits */
39510      char mp_sigstatus;            /* storage for signal # for killed procs */
39511      pid_t mp_pid;                 /* process id */
39512      pid_t mp_procgrp;             /* pid of process group (used for signals) */
39513      pid_t mp_wpid;                /* pid this process is waiting for */
39514      int mp_parent;                /* index of parent process */
39515
39516      /* Real and effective uids and gids. */
39517      uid_t mp_realuid;             /* process' real uid */
39518      uid_t mp_effuid;              /* process' effective uid */
39519      gid_t mp_realgid;             /* process' real gid */
```

```
39520       gid_t mp_effgid;              /* process' effective gid */
39521
39522       /* File identification for sharing. */
39523       ino_t mp_ino;                 /* inode number of file */
39524       dev_t mp_dev;                 /* device number of file system */
39525       time_t mp_ctime;              /* inode changed time */
39526
39527       /* Signal handling information. */
39528       sigset_t mp_ignore;           /* 1 means ignore the signal, 0 means don't */
39529       sigset_t mp_catch;            /* 1 means catch the signal, 0 means don't */
39530       sigset_t mp_sigmask;          /* signals to be blocked */
39531       sigset_t mp_sigmask2;         /* saved copy of mp_sigmask */
39532       sigset_t mp_sigpending;       /* signals being blocked */
39533       struct sigaction mp_sigact[_NSIG + 1]; /* as in sigaction(2) */
39534       vir_bytes mp_sigreturn;       /* address of C library __sigreturn function */
39535
39536       /* Backwards compatibility for signals. */
39537       sighandler_t mp_func;         /* all sigs vectored to a single user fcn */
39538
39539       unsigned mp_flags;            /* flag bits */
39540       vir_bytes mp_procargs;        /* ptr to proc's initial stack arguments */
39541     } mproc[NR_PROCS];
39542
39543     /* Flag values */
39544     #define IN_USE           001    /* set when 'mproc' slot in use */
39545     #define WAITING          002    /* set by WAIT system call */
39546     #define HANGING          004    /* set by EXIT system call */
39547     #define PAUSED           010    /* set by PAUSE system call */
39548     #define ALARM_ON         020    /* set when SIGALRM timer started */
39549     #define SEPARATE         040    /* set if file is separate I & D space */
39550     #define TRACED          0100    /* set if process is to be traced */
39551     #define STOPPED         0200    /* set if process stopped for tracing */
39552     #define SIGSUSPENDED    0400    /* set by SIGSUSPEND system call */
39553
39554     #define NIL_MPROC ((struct mproc *) 0)
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/mm/param.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

39600     /* The following names are synonyms for the variables in the input message. */
39601     #define addr            mm_in.m1_p1
39602     #define exec_name       mm_in.m1_p1
39603     #define exec_len        mm_in.m1_i1
39604     #define func            mm_in.m6_f1
39605     #define grpid           (gid_t) mm_in.m1_i1
39606     #define namelen         mm_in.m1_i1
39607     #define pid             mm_in.m1_i1
39608     #define seconds         mm_in.m1_i1
39609     #define sig             mm_in.m6_i1
39610     #define stack_bytes     mm_in.m1_i2
39611     #define stack_ptr       mm_in.m1_p2
39612     #define status          mm_in.m1_i1
39613     #define usr_id          (uid_t) mm_in.m1_i1
39614     #define request         mm_in.m2_i2
```

```
39615    #define taddr           mm_in.m2_l1
39616    #define data            mm_in.m2_l2
39617    #define sig_nr          mm_in.m1_i2
39618    #define sig_nsa         mm_in.m1_p1
39619    #define sig_osa         mm_in.m1_p2
39620    #define sig_ret         mm_in.m1_p3
39621    #define sig_set         mm_in.m2_l1
39622    #define sig_how         mm_in.m2_i1
39623    #define sig_flags       mm_in.m2_i2
39624    #define sig_context     mm_in.m2_p1
39625    #ifdef _SIGMESSAGE
39626    #define sig_msg         mm_in.m1_i1
39627    #endif
39628    #define reboot_flag     mm_in.m1_i1
39629    #define reboot_code     mm_in.m1_p1
39630    #define reboot_size     mm_in.m1_i2
39631
39632    /* The following names are synonyms for the variables in the output message. */
39633    #define reply_type      mm_out.m_type
39634    #define reply_i1        mm_out.m2_i1
39635    #define reply_p1        mm_out.m2_p1
39636    #define ret_mask        mm_out.m2_l1
39637
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/mm/proto.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

39700    /* Function prototypes. */
39701
39702    struct mproc;           /* need types outside of parameter list --kub */
39703    struct stat;
39704
39705    /* alloc.c */
39706    _PROTOTYPE( phys_clicks alloc_mem, (phys_clicks clicks)                 );
39707    _PROTOTYPE( void free_mem, (phys_clicks base, phys_clicks clicks)       );
39708    _PROTOTYPE( phys_clicks max_hole, (void)                                );
39709    _PROTOTYPE( void mem_init, (phys_clicks *total, phys_clicks *free)      );
39710    _PROTOTYPE( phys_clicks mem_left, (void)                                );
39711    _PROTOTYPE( int do_brk3, (void)                                         );
39712
39713    /* break.c */
39714    _PROTOTYPE( int adjust, (struct mproc *rmp,
39715                            vir_clicks data_clicks, vir_bytes sp)           );
39716    _PROTOTYPE( int do_brk, (void)                                          );
39717    _PROTOTYPE( int size_ok, (int file_type, vir_clicks tc, vir_clicks dc,
39718                            vir_clicks sc, vir_clicks dvir, vir_clicks s_vir) );
39719
39720    /* exec.c */
39721    _PROTOTYPE( int do_exec, (void)                                         );
39722    _PROTOTYPE( struct mproc *find_share, (struct mproc *mp_ign, Ino_t ino,
39723                            Dev_t dev, time_t ctime)                        );
39724
39725    /* forkexit.c */
39726    _PROTOTYPE( int do_fork, (void)                                         );
39727    _PROTOTYPE( int do_mm_exit, (void)                                      );
39728    _PROTOTYPE( int do_waitpid, (void)                                      );
39729    _PROTOTYPE( void mm_exit, (struct mproc *rmp, int exit_status)          );
```

```
39730
39731    /* getset.c */
39732    _PROTOTYPE( int do_getset, (void)                                       );
39733
39734    /* main.c */
39735    _PROTOTYPE( void main, (void)                                           );
39736
39737    #if (MACHINE == MACINTOSH)
39738    _PROTOTYPE( phys_clicks start_click, (void)                             );
39739    #endif
39740
39741    _PROTOTYPE( void reply, (int proc_nr, int result, int res2, char *respt));
39742
39743    /* putk.c */
39744    _PROTOTYPE( void putk, (int c)                                          );
39745
39746    /* signal.c */
39747    _PROTOTYPE( int do_alarm, (void)                                        );
39748    _PROTOTYPE( int do_kill, (void)                                         );
39749    _PROTOTYPE( int do_ksig, (void)                                         );
39750    _PROTOTYPE( int do_pause, (void)                                        );
39751    _PROTOTYPE( int set_alarm, (int proc_nr, int sec)                       );
39752    _PROTOTYPE( int check_sig, (pid_t proc_id, int signo)                   );
39753    _PROTOTYPE( void sig_proc, (struct mproc *rmp, int sig_nr)              );
39754    _PROTOTYPE( int do_sigaction, (void)                                    );
39755    _PROTOTYPE( int do_sigpending, (void)                                   );
39756    _PROTOTYPE( int do_sigprocmask, (void)                                  );
39757    _PROTOTYPE( int do_sigreturn, (void)                                    );
39758    _PROTOTYPE( int do_sigsuspend, (void)                                   );
39759    _PROTOTYPE( int do_reboot, (void)                                       );
39760
39761    /* trace.c */
39762    _PROTOTYPE( int do_trace, (void)                                        );
39763    _PROTOTYPE( void stop_proc, (struct mproc *rmp, int sig_nr)             );
39764
39765    /* utility.c */
39766    _PROTOTYPE( int allowed, (char *name_buf, struct stat *s_buf, int mask) );
39767    _PROTOTYPE( int no_sys, (void)                                          );
39768    _PROTOTYPE( void panic, (char *format, int num)                         );
39769    _PROTOTYPE( void tell_fs, (int what, int p1, int p2, int p3)            );
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/mm/type.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

39800    /* If there were any type definitions local to the Memory Manager, they would
39801     * be here.  This file is included only for symmetry with the kernel and File
39802     * System, which do have some local type definitions.
39803     */
39804
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/mm/alloc.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

39900    /* This file is concerned with allocating and freeing arbitrary-size blocks of
39901     * physical memory on behalf of the FORK and EXEC system calls.  The key data
39902     * structure used is the hole table, which maintains a list of holes in memory.
39903     * It is kept sorted in order of increasing memory address. The addresses
39904     * it contains refer to physical memory, starting at absolute address 0
39905     * (i.e., they are not relative to the start of MM).  During system
39906     * initialization, that part of memory containing the interrupt vectors,
39907     * kernel, and MM are "allocated" to mark them as not available and to
39908     * remove them from the hole list.
39909     *
39910     * The entry points into this file are:
39911     *   alloc_mem: allocate a given sized chunk of memory
39912     *   free_mem:  release a previously allocated chunk of memory
39913     *   mem_init:  initialize the tables when MM start up
39914     *   max_hole:  returns the largest hole currently available
39915     */
39916
39917    #include "mm.h"
39918    #include <minix/com.h>
39919
39920    #define NR_HOLES         128    /* max # entries in hole table */
39921    #define NIL_HOLE (struct hole *) 0
39922
39923    PRIVATE struct hole {
39924      phys_clicks h_base;           /* where does the hole begin? */
39925      phys_clicks h_len;            /* how big is the hole? */
39926      struct hole *h_next;          /* pointer to next entry on the list */
39927    } hole[NR_HOLES];
39928
39929
39930    PRIVATE struct hole *hole_head; /* pointer to first hole */
39931    PRIVATE struct hole *free_slots;        /* ptr to list of unused table slots */
39932
39933    FORWARD _PROTOTYPE( void del_slot, (struct hole *prev_ptr, struct hole *hp) );
39934    FORWARD _PROTOTYPE( void merge, (struct hole *hp)                           );
39935
39936
39937    /*==========================================================================*
39938     *                              alloc_mem                                   *
39939     *==========================================================================*/
39940    PUBLIC phys_clicks alloc_mem(clicks)
39941    phys_clicks clicks;             /* amount of memory requested */
39942    {
39943    /* Allocate a block of memory from the free list using first fit. The block
39944     * consists of a sequence of contiguous bytes, whose length in clicks is
39945     * given by 'clicks'.  A pointer to the block is returned.  The block is
39946     * always on a click boundary.  This procedure is called when memory is
39947     * needed for FORK or EXEC.
39948     */
39949
39950      register struct hole *hp, *prev_ptr;
39951      phys_clicks old_base;
39952
39953      hp = hole_head;
39954      while (hp != NIL_HOLE) {
```

```
39955           if (hp->h_len >= clicks) {
39956                   /* We found a hole that is big enough.  Use it. */
39957                   old_base = hp->h_base;  /* remember where it started */
39958                   hp->h_base += clicks;   /* bite a piece off */
39959                   hp->h_len -= clicks;    /* ditto */
39960
39961                   /* If hole is only partly used, reduce size and return. */
39962                   if (hp->h_len != 0) return(old_base);
39963
39964                   /* The entire hole has been used up.  Manipulate free list. */
39965                   del_slot(prev_ptr, hp);
39966                   return(old_base);
39967           }
39968
39969           prev_ptr = hp;
39970           hp = hp->h_next;
39971     }
39972     return(NO_MEM);
39973   }


39976   /*==========================================================================*
39977    *                              free_mem                                    *
39978    *==========================================================================*/
39979   PUBLIC void free_mem(base, clicks)
39980   phys_clicks base;               /* base address of block to free */
39981   phys_clicks clicks;             /* number of clicks to free */
39982   {
39983   /* Return a block of free memory to the hole list.  The parameters tell where
39984    * the block starts in physical memory and how big it is.  The block is added
39985    * to the hole list.  If it is contiguous with an existing hole on either end,
39986    * it is merged with the hole or holes.
39987    */
39988
39989     register struct hole *hp, *new_ptr, *prev_ptr;
39990
39991     if (clicks == 0) return;
39992     if ( (new_ptr = free_slots) == NIL_HOLE) panic("Hole table full", NO_NUM);
39993     new_ptr->h_base = base;
39994     new_ptr->h_len = clicks;
39995     free_slots = new_ptr->h_next;
39996     hp = hole_head;
39997
39998     /* If this block's address is numerically less than the lowest hole currently
39999      * available, or if no holes are currently available, put this hole on the
40000      * front of the hole list.
40001      */
40002     if (hp == NIL_HOLE || base <= hp->h_base) {
40003           /* Block to be freed goes on front of the hole list. */
40004           new_ptr->h_next = hp;
40005           hole_head = new_ptr;
40006           merge(new_ptr);
40007           return;
40008     }
40009
40010     /* Block to be returned does not go on front of hole list. */
40011     while (hp != NIL_HOLE && base > hp->h_base) {
40012           prev_ptr = hp;
40013           hp = hp->h_next;
40014     }
```

```
40015
40016       /* We found where it goes.  Insert block after 'prev_ptr'. */
40017       new_ptr->h_next = prev_ptr->h_next;
40018       prev_ptr->h_next = new_ptr;
40019       merge(prev_ptr);              /* sequence is 'prev_ptr', 'new_ptr', 'hp' */
40020     }


40023     /*===========================================================================*
40024      *                              del_slot                                     *
40025      *===========================================================================*/
40026     PRIVATE void del_slot(prev_ptr, hp)
40027     register struct hole *prev_ptr; /* pointer to hole entry just ahead of 'hp' */
40028     register struct hole *hp;       /* pointer to hole entry to be removed */
40029     {
40030     /* Remove an entry from the hole list.  This procedure is called when a
40031      * request to allocate memory removes a hole in its entirety, thus reducing
40032      * the numbers of holes in memory, and requiring the elimination of one
40033      * entry in the hole list.
40034      */
40035
40036       if (hp == hole_head)
40037             hole_head = hp->h_next;
40038       else
40039             prev_ptr->h_next = hp->h_next;
40040
40041       hp->h_next = free_slots;
40042       free_slots = hp;
40043     }


40046     /*===========================================================================*
40047      *                              merge                                        *
40048      *===========================================================================*/
40049     PRIVATE void merge(hp)
40050     register struct hole *hp;       /* ptr to hole to merge with its successors */
40051     {
40052     /* Check for contiguous holes and merge any found.  Contiguous holes can occur
40053      * when a block of memory is freed, and it happens to abut another hole on
40054      * either or both ends.  The pointer 'hp' points to the first of a series of
40055      * three holes that can potentially all be merged together.
40056      */
40057
40058       register struct hole *next_ptr;
40059
40060       /* If 'hp' points to the last hole, no merging is possible.  If it does not,
40061        * try to absorb its successor into it and free the successor's table entry.
40062        */
40063       if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
40064       if (hp->h_base + hp->h_len == next_ptr->h_base) {
40065             hp->h_len += next_ptr->h_len;   /* first one gets second one's mem */
40066             del_slot(hp, next_ptr);
40067       } else {
40068             hp = next_ptr;
40069       }
40070
40071       /* If 'hp' now points to the last hole, return; otherwise, try to absorb its
40072        * successor into it.
40073        */
40074       if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
```

```
40075     if (hp->h_base + hp->h_len == next_ptr->h_base) {
40076           hp->h_len += next_ptr->h_len;
40077           del_slot(hp, next_ptr);
40078     }
40079   }


40082   /*==========================================================================*
40083    *                              max_hole                                    *
40084    *==========================================================================*/
40085   PUBLIC phys_clicks max_hole()
40086   {
40087   /* Scan the hole list and return the largest hole. */
40088
40089     register struct hole *hp;
40090     register phys_clicks max;
40091
40092     hp = hole_head;
40093     max = 0;
40094     while (hp != NIL_HOLE) {
40095           if (hp->h_len > max) max = hp->h_len;
40096           hp = hp->h_next;
40097     }
40098     return(max);
40099   }


40102   /*==========================================================================*
40103    *                              mem_init                                    *
40104    *==========================================================================*/
40105   PUBLIC void mem_init(total, free)
40106   phys_clicks *total, *free;              /* memory size summaries */
40107   {
40108   /* Initialize hole lists.  There are two lists: 'hole_head' points to a linked
40109    * list of all the holes (unused memory) in the system; 'free_slots' points to
40110    * a linked list of table entries that are not in use.  Initially, the former
40111    * list has one entry for each chunk of physical memory, and the second
40112    * list links together the remaining table slots.  As memory becomes more
40113    * fragmented in the course of time (i.e., the initial big holes break up into
40114    * smaller holes), new table slots are needed to represent them.  These slots
40115    * are taken from the list headed by 'free_slots'.
40116    */
40117
40118     register struct hole *hp;
40119     phys_clicks base;             /* base address of chunk */
40120     phys_clicks size;             /* size of chunk */
40121     message mess;
40122
40123     /* Put all holes on the free list. */
40124     for (hp = &hole[0]; hp < &hole[NR_HOLES]; hp++) hp->h_next = hp + 1;
40125     hole[NR_HOLES-1].h_next = NIL_HOLE;
40126     hole_head = NIL_HOLE;
40127     free_slots = &hole[0];
40128
40129     /* Ask the kernel for chunks of physical memory and allocate a hole for
40130      * each of them.  The SYS_MEM call responds with the base and size of the
40131      * next chunk and the total amount of memory.
40132      */
40133     *free = 0;
40134     for (;;) {
```

```
40135           mess.m_type = SYS_MEM;
40136           if (sendrec(SYSTASK, &mess) != OK) panic("bad SYS_MEM?", NO_NUM);
40137           base = mess.m1_i1;
40138           size = mess.m1_i2;
40139           if (size == 0) break;           /* no more? */
40140
40141           free_mem(base, size);
40142           *total = mess.m1_i3;
40143           *free += size;
40144       }
40145   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/mm/break.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
40200   /* The MINIX model of memory allocation reserves a fixed amount of memory for
40201    * the combined text, data, and stack segments.  The amount used for a child
40202    * process created by FORK is the same as the parent had.  If the child does
40203    * an EXEC later, the new size is taken from the header of the file EXEC'ed.
40204    *
40205    * The layout in memory consists of the text segment, followed by the data
40206    * segment, followed by a gap (unused memory), followed by the stack segment.
40207    * The data segment grows upward and the stack grows downward, so each can
40208    * take memory from the gap.  If they meet, the process must be killed.  The
40209    * procedures in this file deal with the growth of the data and stack segments.
40210    *
40211    * The entry points into this file are:
40212    *   do_brk:      BRK/SBRK system calls to grow or shrink the data segment
40213    *   adjust:      see if a proposed segment adjustment is allowed
40214    *   size_ok:     see if the segment sizes are feasible
40215    */
40216
40217   #include "mm.h"
40218   #include <signal.h>
40219   #include "mproc.h"
40220   #include "param.h"
40221
40222   #define DATA_CHANGED       1    /* flag value when data segment size changed */
40223   #define STACK_CHANGED      2    /* flag value when stack size changed */
40224
40225   /*===========================================================================*
40226    *                              do_brk                                       *
40227    *===========================================================================*/
40228   PUBLIC int do_brk()
40229   {
40230   /* Perform the brk(addr) system call.
40231    *
40232    * The call is complicated by the fact that on some machines (e.g., 8088),
40233    * the stack pointer can grow beyond the base of the stack segment without
40234    * anybody noticing it.
40235    * The parameter, 'addr' is the new virtual address in D space.
40236    */
40237
40238     register struct mproc *rmp;
40239     int r;
```

```
40240     vir_bytes v, new_sp;
40241     vir_clicks new_clicks;
40242
40243     rmp = mp;
40244     v = (vir_bytes) addr;
40245     new_clicks = (vir_clicks) ( ((long) v + CLICK_SIZE - 1) >> CLICK_SHIFT);
40246     if (new_clicks < rmp->mp_seg[D].mem_vir) {
40247           res_ptr = (char *) -1;
40248           return(ENOMEM);
40249     }
40250     new_clicks -= rmp->mp_seg[D].mem_vir;
40251     sys_getsp(who, &new_sp);      /* ask kernel for current sp value */
40252     r = adjust(rmp, new_clicks, new_sp);
40253     res_ptr = (r == OK ? addr : (char *) -1);
40254     return(r);                    /* return new address or -1 */
40255   }


40258   /*============================================================================*
40259    *                              adjust                                        *
40260    *============================================================================*/
40261   PUBLIC int adjust(rmp, data_clicks, sp)
40262   register struct mproc *rmp;     /* whose memory is being adjusted? */
40263   vir_clicks data_clicks;         /* how big is data segment to become? */
40264   vir_bytes sp;                   /* new value of sp */
40265   {
40266   /* See if data and stack segments can coexist, adjusting them if need be.
40267    * Memory is never allocated or freed.  Instead it is added or removed from the
40268    * gap between data segment and stack segment.  If the gap size becomes
40269    * negative, the adjustment of data or stack fails and ENOMEM is returned.
40270    */
40271
40272     register struct mem_map *mem_sp, *mem_dp;
40273     vir_clicks sp_click, gap_base, lower, old_clicks;
40274     int changed, r, ft;
40275     long base_of_stack, delta;    /* longs avoid certain problems */
40276
40277     mem_dp = &rmp->mp_seg[D];     /* pointer to data segment map */
40278     mem_sp = &rmp->mp_seg[S];     /* pointer to stack segment map */
40279     changed = 0;                  /* set when either segment changed */
40280
40281     if (mem_sp->mem_len == 0) return(OK); /* don't bother init */
40282
40283     /* See if stack size has gone negative (i.e., sp too close to 0xFFFF...) */
40284     base_of_stack = (long) mem_sp->mem_vir + (long) mem_sp->mem_len;
40285     sp_click = sp >> CLICK_SHIFT; /* click containing sp */
40286     if (sp_click >= base_of_stack) return(ENOMEM);        /* sp too high */
40287
40288     /* Compute size of gap between stack and data segments. */
40289     delta = (long) mem_sp->mem_vir - (long) sp_click;
40290     lower = (delta > 0 ? sp_click : mem_sp->mem_vir);
40291
40292     /* Add a safety margin for future stack growth. Impossible to do right. */
40293   #define SAFETY_BYTES  (384 * sizeof(char *))
40294   #define SAFETY_CLICKS ((SAFETY_BYTES + CLICK_SIZE - 1) / CLICK_SIZE)
40295     gap_base = mem_dp->mem_vir + data_clicks + SAFETY_CLICKS;
40296     if (lower < gap_base) return(ENOMEM); /* data and stack collided */
40297
40298     /* Update data length (but not data orgin) on behalf of brk() system call. */
40299     old_clicks = mem_dp->mem_len;
```

```
40300      if (data_clicks != mem_dp->mem_len) {
40301            mem_dp->mem_len = data_clicks;
40302            changed |= DATA_CHANGED;
40303      }
40304
40305      /* Update stack length and origin due to change in stack pointer. */
40306      if (delta > 0) {
40307            mem_sp->mem_vir -= delta;
40308            mem_sp->mem_phys -= delta;
40309            mem_sp->mem_len += delta;
40310            changed |= STACK_CHANGED;
40311      }
40312
40313      /* Do the new data and stack segment sizes fit in the address space? */
40314      ft = (rmp->mp_flags & SEPARATE);
40315      r = size_ok(ft, rmp->mp_seg[T].mem_len, rmp->mp_seg[D].mem_len,
40316           rmp->mp_seg[S].mem_len, rmp->mp_seg[D].mem_vir, rmp->mp_seg[S].mem_vir);
40317      if (r == OK) {
40318            if (changed) sys_newmap((int)(rmp - mproc), rmp->mp_seg);
40319            return(OK);
40320      }
40321
40322      /* New sizes don't fit or require too many page/segment registers. Restore.*/
40323      if (changed & DATA_CHANGED) mem_dp->mem_len = old_clicks;
40324      if (changed & STACK_CHANGED) {
40325            mem_sp->mem_vir += delta;
40326            mem_sp->mem_phys += delta;
40327            mem_sp->mem_len -= delta;
40328      }
40329      return(ENOMEM);
40330    }


40333    /*============================================================================*
40334     *                              size_ok                                       *
40335     *============================================================================*/
40336    PUBLIC int size_ok(file_type, tc, dc, sc, dvir, s_vir)
40337    int file_type;                  /* SEPARATE or 0 */
40338    vir_clicks tc;                  /* text size in clicks */
40339    vir_clicks dc;                  /* data size in clicks */
40340    vir_clicks sc;                  /* stack size in clicks */
40341    vir_clicks dvir;                /* virtual address for start of data seg */
40342    vir_clicks s_vir;               /* virtual address for start of stack seg */
40343    {
40344    /* Check to see if the sizes are feasible and enough segmentation registers
40345     * exist.  On a machine with eight 8K pages, text, data, stack sizes of
40346     * (32K, 16K, 16K) will fit, but (33K, 17K, 13K) will not, even though the
40347     * former is bigger (64K) than the latter (63K).  Even on the 8088 this test
40348     * is needed, since the data and stack may not exceed 4096 clicks.
40349     */
40350
40351    #if (CHIP == INTEL && _WORD_SIZE == 2)
40352      int pt, pd, ps;               /* segment sizes in pages */
40353
40354      pt = ( (tc << CLICK_SHIFT) + PAGE_SIZE - 1)/PAGE_SIZE;
40355      pd = ( (dc << CLICK_SHIFT) + PAGE_SIZE - 1)/PAGE_SIZE;
40356      ps = ( (sc << CLICK_SHIFT) + PAGE_SIZE - 1)/PAGE_SIZE;
40357
40358      if (file_type == SEPARATE) {
40359            if (pt > MAX_PAGES || pd + ps > MAX_PAGES) return(ENOMEM);
```

```
40360       } else {
40361             if (pt + pd + ps > MAX_PAGES) return(ENOMEM);
40362       }
40363     #endif
40364
40365       if (dvir + dc > s_vir) return(ENOMEM);
40366
40367       return(OK);
40368     }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/mm/exec.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
40400     /* This file handles the EXEC system call.  It performs the work as follows:
40401      *    - see if the permissions allow the file to be executed
40402      *    - read the header and extract the sizes
40403      *    - fetch the initial args and environment from the user space
40404      *    - allocate the memory for the new process
40405      *    - copy the initial stack from MM to the process
40406      *    - read in the text and data segments and copy to the process
40407      *    - take care of setuid and setgid bits
40408      *    - fix up 'mproc' table
40409      *    - tell kernel about EXEC
40410      *    - save offset to initial argc (for ps)
40411      *
40412      * The entry points into this file are:
40413      *   do_exec:    perform the EXEC system call
40414      *   find_share: find a process whose text segment can be shared
40415      */
40416
40417     #include "mm.h"
40418     #include <sys/stat.h>
40419     #include <minix/callnr.h>
40420     #include <a.out.h>
40421     #include <signal.h>
40422     #include <string.h>
40423     #include "mproc.h"
40424     #include "param.h"
40425
40426     FORWARD _PROTOTYPE( void load_seg, (int fd, int seg, vir_bytes seg_bytes) );
40427     FORWARD _PROTOTYPE( int new_mem, (struct mproc *sh_mp, vir_bytes text_bytes,
40428                     vir_bytes data_bytes, vir_bytes bss_bytes,
40429                     vir_bytes stk_bytes, phys_bytes tot_bytes)              );
40430     FORWARD _PROTOTYPE( void patch_ptr, (char stack [ARG_MAX ], vir_bytes base) );
40431     FORWARD _PROTOTYPE( int read_header, (int fd, int *ft, vir_bytes *text_bytes,
40432                     vir_bytes *data_bytes, vir_bytes *bss_bytes,
40433                     phys_bytes *tot_bytes, long *sym_bytes, vir_clicks sc,
40434                     vir_bytes *pc)                                          );
40435
40436     #if (SHADOWING == 1)
40437     FORWARD _PROTOTYPE( int relocate, (int fd, unsigned char *buf)          );
40438     #endif
40439
40440     /*===========================================================================*
40441      *                                do_exec                                    *
40442      *===========================================================================*/
40443     PUBLIC int do_exec()
40444     {
```

```
40445  /* Perform the execve(name, argv, envp) call.  The user library builds a
40446   * complete stack image, including pointers, args, environ, etc.  The stack
40447   * is copied to a buffer inside MM, and then to the new core image.
40448   */
40449
40450    register struct mproc *rmp;
40451    struct mproc *sh_mp;
40452    int m, r, fd, ft, sn;
40453    static char mbuf[ARG_MAX];    /* buffer for stack and zeroes */
40454    static char name_buf[PATH_MAX]; /* the name of the file to exec */
40455    char *new_sp, *basename;
40456    vir_bytes src, dst, text_bytes, data_bytes, bss_bytes, stk_bytes, vsp;
40457    phys_bytes tot_bytes;         /* total space for program, including gap */
40458    long sym_bytes;
40459    vir_clicks sc;
40460    struct stat s_buf;
40461    vir_bytes pc;
40462
40463    /* Do some validity checks. */
40464    rmp = mp;
40465    stk_bytes = (vir_bytes) stack_bytes;
40466    if (stk_bytes > ARG_MAX) return(ENOMEM);      /* stack too big */
40467    if (exec_len <= 0 || exec_len > PATH_MAX) return(EINVAL);
40468
40469    /* Get the exec file name and see if the file is executable. */
40470    src = (vir_bytes) exec_name;
40471    dst = (vir_bytes) name_buf;
40472    r = sys_copy(who, D, (phys_bytes) src,
40473                 MM_PROC_NR, D, (phys_bytes) dst, (phys_bytes) exec_len);
40474    if (r != OK) return(r);       /* file name not in user data segment */
40475    tell_fs(CHDIR, who, FALSE, 0);        /* switch to the user's FS environ. */
40476    fd = allowed(name_buf, &s_buf, X_BIT);        /* is file executable? */
40477    if (fd < 0) return(fd);       /* file was not executable */
40478
40479    /* Read the file header and extract the segment sizes. */
40480    sc = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
40481    m = read_header(fd, &ft, &text_bytes, &data_bytes, &bss_bytes,
40482                                          &tot_bytes, &sym_bytes, sc, &pc);
40483    if (m < 0) {
40484          close(fd);              /* something wrong with header */
40485          return(ENOEXEC);
40486    }
40487
40488    /* Fetch the stack from the user before destroying the old core image. */
40489    src = (vir_bytes) stack_ptr;
40490    dst = (vir_bytes) mbuf;
40491    r = sys_copy(who, D, (phys_bytes) src,
40492                         MM_PROC_NR, D, (phys_bytes) dst, (phys_bytes)stk_bytes);
40493    if (r != OK) {
40494          close(fd);              /* can't fetch stack (e.g. bad virtual addr) */
40495          return(EACCES);
40496    }
40497
40498    /* Can the process' text be shared with that of one already running? */
40499    sh_mp = find_share(rmp, s_buf.st_ino, s_buf.st_dev, s_buf.st_ctime);
40500
40501    /* Allocate new memory and release old memory.  Fix map and tell kernel. */
40502    r = new_mem(sh_mp, text_bytes, data_bytes, bss_bytes, stk_bytes, tot_bytes);
40503    if (r != OK) {
40504          close(fd);              /* insufficient core or program too big */
```

```
40505           return(r);
40506       }
40507
40508       /* Save file identification to allow it to be shared. */
40509       rmp->mp_ino = s_buf.st_ino;
40510       rmp->mp_dev = s_buf.st_dev;
40511       rmp->mp_ctime = s_buf.st_ctime;
40512
40513       /* Patch up stack and copy it from MM to new core image. */
40514       vsp = (vir_bytes) rmp->mp_seg[S].mem_vir << CLICK_SHIFT;
40515       vsp += (vir_bytes) rmp->mp_seg[S].mem_len << CLICK_SHIFT;
40516       vsp -= stk_bytes;
40517       patch_ptr(mbuf, vsp);
40518       src = (vir_bytes) mbuf;
40519       r = sys_copy(MM_PROC_NR, D, (phys_bytes) src,
40520                             who, D, (phys_bytes) vsp, (phys_bytes)stk_bytes);
40521       if (r != OK) panic("do_exec stack copy err", NO_NUM);
40522
40523       /* Read in text and data segments. */
40524       if (sh_mp != NULL) {
40525           lseek(fd, (off_t) text_bytes, SEEK_CUR);  /* shared: skip text */
40526       } else {
40527           load_seg(fd, T, text_bytes);
40528       }
40529       load_seg(fd, D, data_bytes);
40530
40531   #if (SHADOWING == 1)
40532       if (lseek(fd, (off_t)sym_bytes, SEEK_CUR) == (off_t) -1) ;    /* error */
40533       if (relocate(fd, (unsigned char *)mbuf) < 0)  ;               /* error */
40534       pc += (vir_bytes) rp->mp_seg[T].mem_vir << CLICK_SHIFT;
40535   #endif
40536
40537       close(fd);                    /* don't need exec file any more */
40538
40539       /* Take care of setuid/setgid bits. */
40540       if ((rmp->mp_flags & TRACED) == 0) { /* suppress if tracing */
40541           if (s_buf.st_mode & I_SET_UID_BIT) {
40542               rmp->mp_effuid = s_buf.st_uid;
40543               tell_fs(SETUID,who, (int)rmp->mp_realuid, (int)rmp->mp_effuid);
40544           }
40545           if (s_buf.st_mode & I_SET_GID_BIT) {
40546               rmp->mp_effgid = s_buf.st_gid;
40547               tell_fs(SETGID,who, (int)rmp->mp_realgid, (int)rmp->mp_effgid);
40548           }
40549       }
40550
40551       /* Save offset to initial argc (for ps) */
40552       rmp->mp_procargs = vsp;
40553
40554       /* Fix 'mproc' fields, tell kernel that exec is done,  reset caught sigs. */
40555       for (sn = 1; sn <= _NSIG; sn++) {
40556           if (sigismember(&rmp->mp_catch, sn)) {
40557               sigdelset(&rmp->mp_catch, sn);
40558               rmp->mp_sigact[sn].sa_handler = SIG_DFL;
40559               sigemptyset(&rmp->mp_sigact[sn].sa_mask);
40560           }
40561       }
40562
40563       rmp->mp_flags &= ~SEPARATE;   /* turn off SEPARATE bit */
40564       rmp->mp_flags |= ft;          /* turn it on for separate I & D files */
```

```
40565     new_sp = (char *) vsp;
40566
40567     tell_fs(EXEC, who, 0, 0);     /* allow FS to handle FD_CLOEXEC files */
40568
40569     /* System will save command line for debugging, ps(1) output, etc. */
40570     basename = strrchr(name_buf, '/');
40571     if (basename == NULL) basename = name_buf; else basename++;
40572     sys_exec(who, new_sp, rmp->mp_flags & TRACED, basename, pc);
40573     return(OK);
40574   }


40577   /*===========================================================================*
40578    *                              read_header                                  *
40579    *===========================================================================*/
40580   PRIVATE int read_header(fd, ft, text_bytes, data_bytes, bss_bytes,
40581                                                 tot_bytes, sym_bytes, sc, pc)
40582   int fd;                       /* file descriptor for reading exec file */
40583   int *ft;                      /* place to return ft number */
40584   vir_bytes *text_bytes;        /* place to return text size */
40585   vir_bytes *data_bytes;        /* place to return initialized data size */
40586   vir_bytes *bss_bytes;         /* place to return bss size */
40587   phys_bytes *tot_bytes;        /* place to return total size */
40588   long *sym_bytes;              /* place to return symbol table size */
40589   vir_clicks sc;                /* stack size in clicks */
40590   vir_bytes *pc;                /* program entry point (initial PC) */
40591   {
40592   /* Read the header and extract the text, data, bss and total sizes from it. */
40593
40594     int m, ct;
40595     vir_clicks tc, dc, s_vir, dvir;
40596     phys_clicks totc;
40597     struct exec hdr;            /* a.out header is read in here */
40598
40599     /* Read the header and check the magic number.  The standard MINIX header
40600      * is defined in <a.out.h>.  It consists of 8 chars followed by 6 longs.
40601      * Then come 4 more longs that are not used here.
40602      *    Byte 0: magic number 0x01
40603      *    Byte 1: magic number 0x03
40604      *    Byte 2: normal = 0x10 (not checked, 0 is OK), separate I/D = 0x20
40605      *    Byte 3: CPU type, Intel 16 bit = 0x04, Intel 32 bit = 0x10,
40606      *            Motorola = 0x0B, Sun SPARC = 0x17
40607      *    Byte 4: Header length = 0x20
40608      *    Bytes 5-7 are not used.
40609      *
40610      *    Now come the 6 longs
40611      *    Bytes  8-11: size of text segments in bytes
40612      *    Bytes 12-15: size of initialized data segment in bytes
40613      *    Bytes 16-19: size of bss in bytes
40614      *    Bytes 20-23: program entry point
40615      *    Bytes 24-27: total memory allocated to program (text, data + stack)
40616      *    Bytes 28-31: size of symbol table in bytes
40617      * The longs are represented in a machine dependent order,
40618      * little-endian on the 8088, big-endian on the 68000.
40619      * The header is followed directly by the text and data segments, and the
40620      * symbol table (if any). The sizes are given in the header. Only the
40621      * text and data segments are copied into memory by exec. The header is
40622      * used here only. The symbol table is for the benefit of a debugger and
40623      * is ignored here.
40624      */
```

```
40625
40626     if (read(fd, (char *) &hdr, A_MINHDR) != A_MINHDR) return(ENOEXEC);
40627
40628     /* Check magic number, cpu type, and flags. */
40629     if (BADMAG(hdr)) return(ENOEXEC);
40630   #if (CHIP == INTEL && _WORD_SIZE == 2)
40631     if (hdr.a_cpu != A_I8086) return(ENOEXEC);
40632   #endif
40633   #if (CHIP == INTEL && _WORD_SIZE == 4)
40634     if (hdr.a_cpu != A_I80386) return(ENOEXEC);
40635   #endif
40636     if ((hdr.a_flags & ~(A_NSYM | A_EXEC | A_SEP)) != 0) return(ENOEXEC);
40637
40638     *ft = ( (hdr.a_flags & A_SEP) ? SEPARATE : 0);    /* separate I & D or not */
40639
40640     /* Get text and data sizes. */
40641     *text_bytes = (vir_bytes) hdr.a_text; /* text size in bytes */
40642     *data_bytes = (vir_bytes) hdr.a_data; /* data size in bytes */
40643     *bss_bytes  = (vir_bytes) hdr.a_bss;  /* bss size in bytes */
40644     *tot_bytes  = hdr.a_total;            /* total bytes to allocate for prog */
40645     *sym_bytes  = hdr.a_syms;             /* symbol table size in bytes */
40646     if (*tot_bytes == 0) return(ENOEXEC);
40647
40648     if (*ft != SEPARATE) {
40649
40650   #if (SHADOWING == 0)
40651           /* If I & D space is not separated, it is all considered data. Text=0*/
40652           *data_bytes += *text_bytes;
40653           *text_bytes = 0;
40654   #else
40655           /*
40656            * Treating text as data increases the shadowing overhead.
40657            * Under the assumption that programs DO NOT MODIFY TEXT
40658            * we can share the text between father and child processes.
40659            * This is similar to the UNIX V7 -n option of ld(1).
40660            * However, for MINIX the linker did not provide alignment
40661            * to click boundaries, so an incomplete text click at the end
40662            * must be treated as data.
40663            * Correct tot_bytes, since it excludes the text segment.
40664            */
40665           *data_bytes += *text_bytes;
40666           *text_bytes = (*text_bytes >> CLICK_SHIFT) << CLICK_SHIFT;
40667           *data_bytes -= *text_bytes;
40668           *tot_bytes -= *text_bytes;
40669   #endif
40670
40671     }
40672     *pc = hdr.a_entry;    /* initial address to start execution */
40673
40674     /* Check to see if segment sizes are feasible. */
40675     tc = ((unsigned long) *text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
40676     dc = (*data_bytes + *bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
40677     totc = (*tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
40678     if (dc >= totc) return(ENOEXEC);      /* stack must be at least 1 click */
40679     dvir = (*ft == SEPARATE ? 0 : tc);
40680     s_vir = dvir + (totc - sc);
40681     m = size_ok(*ft, tc, dc, sc, dvir, s_vir);
40682     ct = hdr.a_hdrlen & BYTE;             /* header length */
40683     if (ct > A_MINHDR) lseek(fd, (off_t) ct, SEEK_SET); /* skip unused hdr */
40684     return(m);
```

```
40685   }


40688   /*===========================================================================*
40689    *                              new_mem                                      *
40690    *===========================================================================*/
40691   PRIVATE int new_mem(sh_mp, text_bytes, data_bytes,bss_bytes,stk_bytes,tot_bytes)
40692   struct mproc *sh_mp;            /* text can be shared with this process */
40693   vir_bytes text_bytes;           /* text segment size in bytes */
40694   vir_bytes data_bytes;           /* size of initialized data in bytes */
40695   vir_bytes bss_bytes;            /* size of bss in bytes */
40696   vir_bytes stk_bytes;            /* size of initial stack segment in bytes */
40697   phys_bytes tot_bytes;           /* total memory to allocate, including gap */
40698   {
40699   /* Allocate new memory and release the old memory.  Change the map and report
40700    * the new map to the kernel.  Zero the new core image's bss, gap and stack.
40701    */
40702
40703     register struct mproc *rmp;
40704     vir_clicks text_clicks, data_clicks, gap_clicks, stack_clicks, tot_clicks;
40705     phys_clicks new_base;
40706
40707   #if (SHADOWING == 1)
40708     phys_clicks base, size;
40709   #else
40710     static char zero[1024];              /* used to zero bss */
40711     phys_bytes bytes, base, count, bss_offset;
40712   #endif
40713
40714     /* No need to allocate text if it can be shared. */
40715     if (sh_mp != NULL) text_bytes = 0;
40716
40717     /* Acquire the new memory.  Each of the 4 parts: text, (data+bss), gap,
40718      * and stack occupies an integral number of clicks, starting at click
40719      * boundary.  The data and bss parts are run together with no space.
40720      */
40721
40722     text_clicks = ((unsigned long) text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
40723     data_clicks = (data_bytes + bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
40724     stack_clicks = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
40725     tot_clicks = (tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
40726     gap_clicks = tot_clicks - data_clicks - stack_clicks;
40727     if ( (int) gap_clicks < 0) return(ENOMEM);
40728
40729     /* Check to see if there is a hole big enough.  If so, we can risk first
40730      * releasing the old core image before allocating the new one, since we
40731      * know it will succeed.  If there is not enough, return failure.
40732      */
40733     if (text_clicks + tot_clicks > max_hole()) return(EAGAIN);
40734
40735     /* There is enough memory for the new core image.  Release the old one. */
40736     rmp = mp;
40737
40738   #if (SHADOWING == 0)
40739     if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL) {
40740           /* No other process shares the text segment, so free it. */
40741           free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
40742     }
40743     /* Free the data and stack segments. */
40744     free_mem(rmp->mp_seg[D].mem_phys,
```

```
40745         rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);
40746   #endif
40747
40748     /* We have now passed the point of no return.  The old core image has been
40749      * forever lost.  The call must go through now.  Set up and report new map.
40750      */
40751     new_base = alloc_mem(text_clicks + tot_clicks);       /* new core image */
40752     if (new_base == NO_MEM) panic("MM hole list is inconsistent", NO_NUM);
40753
40754     if (sh_mp != NULL) {
40755           /* Share the text segment. */
40756           rmp->mp_seg[T] = sh_mp->mp_seg[T];
40757     } else {
40758           rmp->mp_seg[T].mem_phys = new_base;
40759           rmp->mp_seg[T].mem_vir = 0;
40760           rmp->mp_seg[T].mem_len = text_clicks;
40761     }
40762     rmp->mp_seg[D].mem_phys = new_base + text_clicks;
40763     rmp->mp_seg[D].mem_vir = 0;
40764     rmp->mp_seg[D].mem_len = data_clicks;
40765     rmp->mp_seg[S].mem_phys = rmp->mp_seg[D].mem_phys + data_clicks + gap_clicks;
40766     rmp->mp_seg[S].mem_vir = rmp->mp_seg[D].mem_vir + data_clicks + gap_clicks;
40767     rmp->mp_seg[S].mem_len = stack_clicks;
40768
40769   #if (CHIP == M68000)
40770   #if (SHADOWING == 0)
40771     rmp->mp_seg[T].mem_vir = 0;
40772     rmp->mp_seg[D].mem_vir = rmp->mp_seg[T].mem_len;
40773     rmp->mp_seg[S].mem_vir = rmp->mp_seg[D].mem_vir + rmp->mp_seg[D].mem_len + gap_clicks;
40774   #else
40775     rmp->mp_seg[T].mem_vir = rmp->mp_seg[T].mem_phys;
40776     rmp->mp_seg[D].mem_vir = rmp->mp_seg[D].mem_phys;
40777     rmp->mp_seg[S].mem_vir = rmp->mp_seg[S].mem_phys;
40778   #endif
40779   #endif
40780
40781   #if (SHADOWING == 0)
40782     sys_newmap(who, rmp->mp_seg);   /* report new map to the kernel */
40783
40784     /* Zero the bss, gap, and stack segment. */
40785     bytes = (phys_bytes)(data_clicks + gap_clicks + stack_clicks) << CLICK_SHIFT;
40786     base = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
40787     bss_offset = (data_bytes >> CLICK_SHIFT) << CLICK_SHIFT;
40788     base += bss_offset;
40789     bytes -= bss_offset;
40790
40791     while (bytes > 0) {
40792           count = MIN(bytes, (phys_bytes) sizeof(zero));
40793           if (sys_copy(MM_PROC_NR, D, (phys_bytes) zero,
40794                                                 ABS, 0, base, count) != OK) {
40795                 panic("new_mem can't zero", NO_NUM);
40796           }
40797           base += count;
40798           bytes -= count;
40799     }
40800   #endif
40801
40802   #if (SHADOWING == 1)
40803     sys_fresh(who, rmp->mp_seg, (phys_clicks)(data_bytes >> CLICK_SHIFT),
40804                           &base, &size);
```

```
40805     free_mem(base, size);
40806   #endif
40807
40808     return(OK);
40809   }


40812   /*===========================================================================*
40813    *                              patch_ptr                                    *
40814    *===========================================================================*/
40815   PRIVATE void patch_ptr(stack, base)
40816   char stack[ARG_MAX];    /* pointer to stack image within MM */
40817   vir_bytes base;                /* virtual address of stack base inside user */
40818   {
40819   /* When doing an exec(name, argv, envp) call, the user builds up a stack
40820    * image with arg and env pointers relative to the start of the stack.  Now
40821    * these pointers must be relocated, since the stack is not positioned at
40822    * address 0 in the user's address space.
40823    */
40824
40825     char **ap, flag;
40826     vir_bytes v;
40827
40828     flag = 0;                    /* counts number of 0-pointers seen */
40829     ap = (char **) stack;        /* points initially to 'nargs' */
40830     ap++;                        /* now points to argv[0] */
40831     while (flag < 2) {
40832           if (ap >= (char **) &stack[ARG_MAX]) return;    /* too bad */
40833           if (*ap != NIL_PTR) {
40834                   v = (vir_bytes) *ap;    /* v is relative pointer */
40835                   v += base;              /* relocate it */
40836                   *ap = (char *) v;       /* put it back */
40837           } else {
40838                   flag++;
40839           }
40840           ap++;
40841     }
40842   }


40845   /*===========================================================================*
40846    *                              load_seg                                     *
40847    *===========================================================================*/
40848   PRIVATE void load_seg(fd, seg, seg_bytes)
40849   int fd;                        /* file descriptor to read from */
40850   int seg;                       /* T or D */
40851   vir_bytes seg_bytes;           /* how big is the segment */
40852   {
40853   /* Read in text or data from the exec file and copy to the new core image.
40854    * This procedure is a little bit tricky.  The logical way to load a segment
40855    * would be to read it block by block and copy each block to the user space
40856    * one at a time.  This is too slow, so we do something dirty here, namely
40857    * send the user space and virtual address to the file system in the upper
40858    * 10 bits of the file descriptor, and pass it the user virtual address
40859    * instead of a MM address.  The file system extracts these parameters when
40860    * gets a read call from the memory manager, which is the only process that
40861    * is permitted to use this trick.  The file system then copies the whole
40862    * segment directly to user space, bypassing MM completely.
40863    */
40864
```

```
40865     int new_fd, bytes;
40866     char *ubuf_ptr;
40867
40868     new_fd = (who << 8) | (seg << 6) | fd;
40869     ubuf_ptr = (char *) ((vir_bytes)mp->mp_seg[seg].mem_vir << CLICK_SHIFT);
40870     while (seg_bytes != 0) {
40871           bytes = (INT_MAX / BLOCK_SIZE) * BLOCK_SIZE;
40872           if (seg_bytes < bytes)
40873                   bytes = (int)seg_bytes;
40874           if (read(new_fd, ubuf_ptr, bytes) != bytes)
40875                   break;          /* error */
40876           ubuf_ptr += bytes;
40877           seg_bytes -= bytes;
40878     }
40879   }


40882   /*===========================================================================*
40883    *                              find_share                                   *
40884    *===========================================================================*/
40885   PUBLIC struct mproc *find_share(mp_ign, ino, dev, ctime)
40886   struct mproc *mp_ign;           /* process that should not be looked at */
40887   ino_t ino;                      /* parameters that uniquely identify a file */
40888   dev_t dev;
40889   time_t ctime;
40890   {
40891   /* Look for a process that is the file <ino, dev, ctime> in execution.  Don't
40892    * accidentally "find" mp_ign, because it is the process on whose behalf this
40893    * call is made.
40894    */
40895     struct mproc *sh_mp;
40896
40897     for (sh_mp = &mproc[INIT_PROC_NR]; sh_mp < &mproc[NR_PROCS]; sh_mp++) {
40898           if ((sh_mp->mp_flags & (IN_USE | HANGING | SEPARATE))
40899                                           != (IN_USE | SEPARATE)) continue;
40900           if (sh_mp == mp_ign) continue;
40901           if (sh_mp->mp_ino != ino) continue;
40902           if (sh_mp->mp_dev != dev) continue;
40903           if (sh_mp->mp_ctime != ctime) continue;
40904           return sh_mp;
40905     }
40906     return(NULL);
40907   }


40910   #if (SHADOWING == 1)
40911   /*===========================================================================*
40912    *                              relocate                                     *
40913    *===========================================================================*/
40914   PRIVATE int relocate(fd, buf)
40915   int fd;                         /* file descriptor to read from */
40916   unsigned char *buf;             /* borrowed from do_exec() */
40917   {
40918     register int n;
40919     register unsigned char *p, c;
40920     register phys_bytes off;
40921     register phys_bytes adr;
40922
40923     /* Read in relocation info from the exec file and relocate.
40924      * Relocation info is in GEMDOS format. Only longs can be relocated.
```

```
40925       *
40926       * The GEMDOS format starts with a long L: the offset to the
40927       * beginning of text for the first long to be relocated.
40928       * If L==0 then no relocations have to be made.
40929       *
40930       * The long is followed by zero or more bytes. Each byte B is
40931       * processed separately, in one of the following ways:
40932       *
40933       * B==0:
40934       *    end of relocation
40935       * B==1:
40936       *    no relocation, but add 254 to the current offset
40937       * B==0bWWWWWWW0:
40938       *    B is added to the current offset and the long addressed
40939       *    is relocated. Note that 00000010 means 1 word distance.
40940       * B==0bXXXXXXX1:
40941       *    illegal
40942       */
40943      off = (phys_bytes)mp->mp_seg[T].mem_phys << CLICK_SHIFT;
40944      p = buf;
40945      n = read(fd, (char *)p, ARG_MAX);
40946      if (n < sizeof(long)) return(-1);     /* error */
40947      if (*((long *)p) == 0) return(0);     /* ok */
40948      adr = off + *((long *)p);
40949      n -= sizeof(long);
40950      p += sizeof(long);
40951      *((long *)adr) += off;
40952      while (1) {                   /* once per relocation byte */
40953            if (--n < 0) {
40954                    p = buf;
40955                    n = read(fd, (char *)p, ARG_MAX);
40956                    if (--n < 0)
40957                            return(-1);     /* error */
40958            }
40959            c = *p++;
40960            if (c == 1)
40961                    adr += 254;
40962            else if (c == 0)
40963                    return(0);      /* ok */
40964            else if (c & 1)
40965                    return(-1);     /* error */
40966            else {
40967                    adr += c;
40968                    *((long *)adr) += off;
40969            }
40970      }
40971    }
40972    #endif
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/mm/forkexit.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

41000    /* This file deals with creating processes (via FORK) and deleting them (via
41001     * EXIT/WAIT).  When a process forks, a new slot in the 'mproc' table is
41002     * allocated for it, and a copy of the parent's core image is made for the
41003     * child.  Then the kernel and file system are informed.  A process is removed
41004     * from the 'mproc' table when two events have occurred: (1) it has exited or
41005     * been killed by a signal, and (2) the parent has done a WAIT.  If the process
41006     * exits first, it continues to occupy a slot until the parent does a WAIT.
41007     *
41008     * The entry points into this file are:
41009     *   do_fork:    perform the FORK system call
41010     *   do_mm_exit: perform the EXIT system call (by calling mm_exit())
41011     *   mm_exit:    actually do the exiting
41012     *   do_wait:    perform the WAITPID or WAIT system call
41013     */
41014
41015
41016    #include "mm.h"
41017    #include <sys/wait.h>
41018    #include <minix/callnr.h>
41019    #include <signal.h>
41020    #include "mproc.h"
41021    #include "param.h"
41022
41023    #define LAST_FEW            2   /* last few slots reserved for superuser */
41024
41025    PRIVATE pid_t next_pid = INIT_PID+1;    /* next pid to be assigned */
41026
41027    FORWARD _PROTOTYPE (void cleanup, (register struct mproc *child) );
41028
41029    /*===========================================================================*
41030     *                                do_fork                                    *
41031     *===========================================================================*/
41032    PUBLIC int do_fork()
41033    {
41034    /* The process pointed to by 'mp' has forked.  Create a child process. */
41035
41036      register struct mproc *rmp;   /* pointer to parent */
41037      register struct mproc *rmc;   /* pointer to child */
41038      int i, child_nr, t;
41039      phys_clicks prog_clicks, child_base = 0;
41040      phys_bytes prog_bytes, parent_abs, child_abs; /* Intel only */
41041
41042     /* If tables might fill up during FORK, don't even start since recovery half
41043      * way through is such a nuisance.
41044      */
41045      rmp = mp;
41046      if (procs_in_use == NR_PROCS) return(EAGAIN);
41047      if (procs_in_use >= NR_PROCS-LAST_FEW && rmp->mp_effuid != 0)return(EAGAIN);
41048
41049      /* Determine how much memory to allocate.  Only the data and stack need to
41050       * be copied, because the text segment is either shared or of zero length.
41051       */
41052      prog_clicks = (phys_clicks) rmp->mp_seg[S].mem_len;
41053      prog_clicks += (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
41054    #if (SHADOWING == 0)
```

```
41055     prog_bytes = (phys_bytes) prog_clicks << CLICK_SHIFT;
41056   #endif
41057     if ( (child_base = alloc_mem(prog_clicks)) == NO_MEM) return(ENOMEM);
41058
41059   #if (SHADOWING == 0)
41060     /* Create a copy of the parent's core image for the child. */
41061     child_abs = (phys_bytes) child_base << CLICK_SHIFT;
41062     parent_abs = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
41063     i = sys_copy(ABS, 0, parent_abs, ABS, 0, child_abs, prog_bytes);
41064     if (i < 0) panic("do_fork can't copy", i);
41065   #endif
41066
41067     /* Find a slot in 'mproc' for the child process.  A slot must exist. */
41068     for (rmc = &mproc[0]; rmc < &mproc[NR_PROCS]; rmc++)
41069           if ( (rmc->mp_flags & IN_USE) == 0) break;
41070
41071     /* Set up the child and its memory map; copy its 'mproc' slot from parent. */
41072     child_nr = (int)(rmc - mproc);        /* slot number of the child */
41073     procs_in_use++;
41074     *rmc = *rmp;                  /* copy parent's process slot to child's */
41075
41076     rmc->mp_parent = who;         /* record child's parent */
41077     rmc->mp_flags &= ~TRACED;     /* child does not inherit trace status */
41078   #if (SHADOWING == 0)
41079     /* A separate I&D child keeps the parents text segment.  The data and stack
41080      * segments must refer to the new copy.
41081      */
41082     if (!(rmc->mp_flags & SEPARATE)) rmc->mp_seg[T].mem_phys = child_base;
41083     rmc->mp_seg[D].mem_phys = child_base;
41084     rmc->mp_seg[S].mem_phys = rmc->mp_seg[D].mem_phys +
41085                             (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
41086   #endif
41087     rmc->mp_exitstatus = 0;
41088     rmc->mp_sigstatus = 0;
41089
41090     /* Find a free pid for the child and put it in the table. */
41091     do {
41092           t = 0;                  /* 't' = 0 means pid still free */
41093           next_pid = (next_pid < 30000 ? next_pid + 1 : INIT_PID + 1);
41094           for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++)
41095                   if (rmp->mp_pid == next_pid || rmp->mp_procgrp == next_pid) {
41096                           t = 1;
41097                           break;
41098                   }
41099           rmc->mp_pid = next_pid; /* assign pid to child */
41100     } while (t);
41101
41102     /* Tell kernel and file system about the (now successful) FORK. */
41103     sys_fork(who, child_nr, rmc->mp_pid, child_base); /* child_base is 68K only*/
41104     tell_fs(FORK, who, child_nr, rmc->mp_pid);
41105
41106   #if (SHADOWING == 0)
41107     /* Report child's memory map to kernel. */
41108     sys_newmap(child_nr, rmc->mp_seg);
41109   #endif
41110
41111     /* Reply to child to wake it up. */
41112     reply(child_nr, 0, 0, NIL_PTR);
41113     return(next_pid);             /* child's pid */
41114   }
```

```
41117   /*===========================================================================*
41118    *                              do_mm_exit                                   *
41119    *===========================================================================*/
41120   PUBLIC int do_mm_exit()
41121   {
41122   /* Perform the exit(status) system call. The real work is done by mm_exit(),
41123    * which is also called when a process is killed by a signal.
41124    */
41125
41126     mm_exit(mp, status);
41127     dont_reply = TRUE;            /* don't reply to newly terminated process */
41128     return(OK);                   /* pro forma return code */
41129   }


41132   /*===========================================================================*
41133    *                              mm_exit                                      *
41134    *===========================================================================*/
41135   PUBLIC void mm_exit(rmp, exit_status)
41136   register struct mproc *rmp;     /* pointer to the process to be terminated */
41137   int exit_status;                /* the process' exit status (for parent) */
41138   {
41139   /* A process is done.  Release most of the process' possessions.  If its
41140    * parent is waiting, release the rest, else hang.
41141    */
41142
41143     register int proc_nr;
41144     int parent_waiting, right_child;
41145     pid_t pidarg, procgrp;
41146     phys_clicks base, size, s;            /* base and size used on 68000 only */
41147
41148     proc_nr = (int) (rmp - mproc);        /* get process slot number */
41149
41150     /* Remember a session leader's process group. */
41151     procgrp = (rmp->mp_pid == mp->mp_procgrp) ? mp->mp_procgrp : 0;
41152
41153     /* If the exited process has a timer pending, kill it. */
41154     if (rmp->mp_flags & ALARM_ON) set_alarm(proc_nr, (unsigned) 0);
41155
41156     /* Tell the kernel and FS that the process is no longer runnable. */
41157     tell_fs(EXIT, proc_nr, 0, 0);  /* file system can free the proc slot */
41158     sys_xit(rmp->mp_parent, proc_nr, &base, &size);
41159   #if (SHADOWING == 1)
41160     free_mem(base, size);
41161   #endif
41162
41163   #if (SHADOWING == 0)
41164     /* Release the memory occupied by the child. */
41165     if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL) {
41166           /* No other process shares the text segment, so free it. */
41167           free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
41168     }
41169     /* Free the data and stack segments. */
41170     free_mem(rmp->mp_seg[D].mem_phys,
41171         rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);
41172   #endif
41173
41174     /* The process slot can only be freed if the parent has done a WAIT. */
```

```
41175     rmp->mp_exitstatus = (char) exit_status;
41176     pidarg = mproc[rmp->mp_parent].mp_wpid;       /* who's being waited for? */
41177     parent_waiting = mproc[rmp->mp_parent].mp_flags & WAITING;
41178     if (pidarg == -1 || pidarg == rmp->mp_pid || -pidarg == rmp->mp_procgrp)
41179           right_child = TRUE;             /* child meets one of the 3 tests */
41180     else
41181           right_child = FALSE;            /* child fails all 3 tests */
41182     if (parent_waiting && right_child)
41183           cleanup(rmp);                   /* tell parent and release child slot */
41184     else
41185           rmp->mp_flags |= HANGING;       /* parent not waiting, suspend child */
41186
41187     /* If the process has children, disinherit them.  INIT is the new parent. */
41188     for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
41189           if (rmp->mp_flags & IN_USE && rmp->mp_parent == proc_nr) {
41190                   /* 'rmp' now points to a child to be disinherited. */
41191                   rmp->mp_parent = INIT_PROC_NR;
41192                   parent_waiting = mproc[INIT_PROC_NR].mp_flags & WAITING;
41193                   if (parent_waiting && (rmp->mp_flags & HANGING)) cleanup(rmp);
41194           }
41195     }
41196
41197     /* Send a hangup to the process' process group if it was a session leader. */
41198     if (procgrp != 0) check_sig(-procgrp, SIGHUP);
41199   }


41202   /*===========================================================================*
41203    *                                do_waitpid                                 *
41204    *===========================================================================*/
41205   PUBLIC int do_waitpid()
41206   {
41207   /* A process wants to wait for a child to terminate. If one is already waiting,
41208    * go clean it up and let this WAIT call terminate.  Otherwise, really wait.
41209    * Both WAIT and WAITPID are handled by this code.
41210    */
41211
41212     register struct mproc *rp;
41213     int pidarg, options, children, res2;
41214
41215     /* A process calling WAIT never gets a reply in the usual way via the
41216      * reply() in the main loop (unless WNOHANG is set or no qualifying child
41217      * exists).  If a child has already exited, the routine cleanup() sends
41218      * the reply to awaken the caller.
41219      */
41220
41221     /* Set internal variables, depending on whether this is WAIT or WAITPID. */
41222     pidarg  = (mm_call == WAIT ? -1 : pid);       /* first param of waitpid */
41223     options = (mm_call == WAIT ?  0 : sig_nr);    /* third param of waitpid */
41224     if (pidarg == 0) pidarg = -mp->mp_procgrp;    /* pidarg < 0 ==> proc grp */
41225
41226     /* Is there a child waiting to be collected? At this point, pidarg != 0:
41227      *    pidarg  >  0 means pidarg is pid of a specific process to wait for
41228      *    pidarg == -1 means wait for any child
41229      *    pidarg  < -1 means wait for any child whose process group = -pidarg
41230      */
41231     children = 0;
41232     for (rp = &mproc[0]; rp < &mproc[NR_PROCS]; rp++) {
41233           if ( (rp->mp_flags & IN_USE) && rp->mp_parent == who) {
41234                   /* The value of pidarg determines which children qualify. */
```

```
41235                   if (pidarg  > 0 && pidarg != rp->mp_pid) continue;
41236                   if (pidarg < -1 && -pidarg != rp->mp_procgrp) continue;
41237
41238                   children++;             /* this child is acceptable */
41239                   if (rp->mp_flags & HANGING) {
41240                           /* This child meets the pid test and has exited. */
41241                           cleanup(rp);    /* this child has already exited */
41242                           dont_reply = TRUE;
41243                           return(OK);
41244                   }
41245                   if ((rp->mp_flags & STOPPED) && rp->mp_sigstatus) {
41246                           /* This child meets the pid test and is being traced.*/
41247                           res2 =  0177 | (rp->mp_sigstatus << 8);
41248                           reply(who, rp->mp_pid, res2, NIL_PTR);
41249                           dont_reply = TRUE;
41250                           rp->mp_sigstatus = 0;
41251                           return(OK);
41252                   }
41253           }
41254     }
41255
41256     /* No qualifying child has exited.  Wait for one, unless none exists. */
41257     if (children > 0) {
41258           /* At least 1 child meets the pid test exists, but has not exited. */
41259           if (options & WNOHANG) return(0);    /* parent does not want to wait */
41260           mp->mp_flags |= WAITING;             /* parent wants to wait */
41261           mp->mp_wpid = (pid_t) pidarg;        /* save pid for later */
41262           dont_reply = TRUE;                   /* do not reply now though */
41263           return(OK);                          /* yes - wait for one to exit */
41264     } else {
41265           /* No child even meets the pid test.  Return error immediately. */
41266           return(ECHILD);                      /* no - parent has no children */
41267     }
41268   }


41271   /*===========================================================================*
41272    *                                cleanup                                    *
41273    *===========================================================================*/
41274   PRIVATE void cleanup(child)
41275   register struct mproc *child;   /* tells which process is exiting */
41276   {
41277   /* Finish off the exit of a process.  The process has exited or been killed
41278    * by a signal, and its parent is waiting.
41279    */
41280
41281     int exitstatus;
41282
41283     /* Wake up the parent. */
41284     exitstatus = (child->mp_exitstatus << 8) | (child->mp_sigstatus & 0377);
41285     reply(child->mp_parent, child->mp_pid, exitstatus, NIL_PTR);
41286     mproc[child->mp_parent].mp_flags &= ~WAITING; /* parent no longer waiting */
41287
41288     /* Release the process table entry. */
41289     child->mp_flags = 0;
41290     procs_in_use--;
41291   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/mm/getset.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

41300    /* This file handles the 4 system calls that get and set uids and gids.
41301     * It also handles getpid(), setsid(), and getpgrp().  The code for each
41302     * one is so tiny that it hardly seemed worthwhile to make each a separate
41303     * function.
41304     */
41305
41306    #include "mm.h"
41307    #include <minix/callnr.h>
41308    #include <signal.h>
41309    #include "mproc.h"
41310    #include "param.h"
41311
41312    /*===========================================================================*
41313     *                                do_getset                                  *
41314     *===========================================================================*/
41315    PUBLIC int do_getset()
41316    {
41317    /* Handle GETUID, GETGID, GETPID, GETPGRP, SETUID, SETGID, SETSID.  The four
41318     * GETs and SETSID return their primary results in 'r'.  GETUID, GETGID, and
41319     * GETPID also return secondary results (the effective IDs, or the parent
41320     * process ID) in 'result2', which is returned to the user.
41321     */
41322
41323      register struct mproc *rmp = mp;
41324      register int r;
41325
41326      switch(mm_call) {
41327            case GETUID:
41328                    r = rmp->mp_realuid;
41329                    result2 = rmp->mp_effuid;
41330                    break;
41331
41332            case GETGID:
41333                    r = rmp->mp_realgid;
41334                    result2 = rmp->mp_effgid;
41335                    break;
41336
41337            case GETPID:
41338                    r = mproc[who].mp_pid;
41339                    result2 = mproc[rmp->mp_parent].mp_pid;
41340                    break;
41341
41342            case SETUID:
41343                    if (rmp->mp_realuid != usr_id && rmp->mp_effuid != SUPER_USER)
41344                            return(EPERM);
41345                    rmp->mp_realuid = usr_id;
41346                    rmp->mp_effuid = usr_id;
41347                    tell_fs(SETUID, who, usr_id, usr_id);
41348                    r = OK;
41349                    break;
41350
41351            case SETGID:
41352                    if (rmp->mp_realgid != grpid && rmp->mp_effuid != SUPER_USER)
41353                            return(EPERM);
41354                    rmp->mp_realgid = grpid;
```

```
41355                   rmp->mp_effgid = grpid;
41356                   tell_fs(SETGID, who, grpid, grpid);
41357                   r = OK;
41358                   break;
41359
41360           case SETSID:
41361                   if (rmp->mp_procgrp == rmp->mp_pid) return(EPERM);
41362                   rmp->mp_procgrp = rmp->mp_pid;
41363                   tell_fs(SETSID, who, 0, 0);
41364                   /*FALL THROUGH*/
41365
41366           case GETPGRP:
41367                   r = rmp->mp_procgrp;
41368                   break;
41369
41370           default:
41371                   r = EINVAL;
41372                   break;
41373     }
41374     return(r);
41375   }
```

++
src/mm/main.c
++

```
41400   /* This file contains the main program of the memory manager and some related
41401    * procedures.  When MINIX starts up, the kernel runs for a little while,
41402    * initializing itself and its tasks, and then it runs MM and FS.  Both MM
41403    * and FS initialize themselves as far as they can.  FS then makes a call to
41404    * MM, because MM has to wait for FS to acquire a RAM disk.  MM asks the
41405    * kernel for all free memory and starts serving requests.
41406    *
41407    * The entry points into this file are:
41408    *   main:      starts MM running
41409    *   reply:     reply to a process making an MM system call
41410    */
41411
41412   #include "mm.h"
41413   #include <minix/callnr.h>
41414   #include <minix/com.h>
41415   #include <signal.h>
41416   #include <fcntl.h>
41417   #include <sys/ioctl.h>
41418   #include "mproc.h"
41419   #include "param.h"
41420
41421   FORWARD _PROTOTYPE( void get_work, (void)                               );
41422   FORWARD _PROTOTYPE( void mm_init, (void)                                );
41423
41424   /*===========================================================================*
41425    *                              main                                         *
41426    *===========================================================================*/
41427   PUBLIC void main()
41428   {
41429   /* Main routine of the memory manager. */
```

```
41430
41431     int error;
41432
41433     mm_init();                    /* initialize memory manager tables */
41434
41435     /* This is MM's main loop-  get work and do it, forever and forever. */
41436     while (TRUE) {
41437           /* Wait for message. */
41438           get_work();             /* wait for an MM system call */
41439           mp = &mproc[who];
41440
41441           /* Set some flags. */
41442           error = OK;
41443           dont_reply = FALSE;
41444           err_code = -999;
41445
41446           /* If the call number is valid, perform the call. */
41447           if (mm_call < 0 || mm_call >= NCALLS)
41448                   error = EBADCALL;
41449           else
41450                   error = (*call_vec[mm_call])();
41451
41452           /* Send the results back to the user to indicate completion. */
41453           if (dont_reply) continue;       /* no reply for EXIT and WAIT */
41454           if (mm_call == EXEC && error == OK) continue;
41455           reply(who, error, result2, res_ptr);
41456     }
41457   }


41460   /*===========================================================================*
41461    *                                get_work                                   *
41462    *===========================================================================*/
41463   PRIVATE void get_work()
41464   {
41465   /* Wait for the next message and extract useful information from it. */
41466
41467     if (receive(ANY, &mm_in) != OK) panic("MM receive error", NO_NUM);
41468     who = mm_in.m_source;         /* who sent the message */
41469     mm_call = mm_in.m_type;       /* system call number */
41470   }


41473   /*===========================================================================*
41474    *                                reply                                      *
41475    *===========================================================================*/
41476   PUBLIC void reply(proc_nr, result, res2, respt)
41477   int proc_nr;                    /* process to reply to */
41478   int result;                     /* result of the call (usually OK or error #)*/
41479   int res2;                       /* secondary result */
41480   char *respt;                    /* result if pointer */
41481   {
41482   /* Send a reply to a user process. */
41483
41484     register struct mproc *proc_ptr;
41485
41486     proc_ptr = &mproc[proc_nr];
41487     /*
41488      * To make MM robust, check to see if destination is still alive.  This
41489      * validy check must be skipped if the caller is a task.
```

```
41490       */
41491     if ((who >=0) && ((proc_ptr->mp_flags&IN_USE) == 0 ||
41492           (proc_ptr->mp_flags&HANGING))) return;
41493
41494     reply_type = result;
41495     reply_i1 = res2;
41496     reply_p1 = respt;
41497     if (send(proc_nr, &mm_out) != OK) panic("MM can't reply", NO_NUM);
41498   }


41501   /*===========================================================================*
41502    *                              mm_init                                      *
41503    *===========================================================================*/
41504   PRIVATE void mm_init()
41505   {
41506   /* Initialize the memory manager. */
41507
41508     static char core_sigs[] = {
41509           SIGQUIT, SIGILL, SIGTRAP, SIGABRT,
41510           SIGEMT, SIGFPE, SIGUSR1, SIGSEGV,
41511           SIGUSR2, 0 };
41512     register int proc_nr;
41513     register struct mproc *rmp;
41514     register char *sig_ptr;
41515     phys_clicks ram_clicks, total_clicks, minix_clicks, free_clicks, dummy;
41516     message mess;
41517     struct mem_map kernel_map[NR_SEGS];
41518     int mem;
41519
41520     /* Build the set of signals which cause core dumps. Do it the Posix
41521      * way, so no knowledge of bit positions is needed.
41522      */
41523     sigemptyset(&core_sset);
41524     for (sig_ptr = core_sigs; *sig_ptr != 0; sig_ptr++)
41525           sigaddset(&core_sset, *sig_ptr);
41526
41527     /* Get the memory map of the kernel to see how much memory it uses,
41528      * including the gap between address 0 and the start of the kernel.
41529      */
41530     sys_getmap(SYSTASK, kernel_map);
41531     minix_clicks = kernel_map[S].mem_phys + kernel_map[S].mem_len;
41532
41533     /* Initialize MM's tables. */
41534     for (proc_nr = 0; proc_nr <= INIT_PROC_NR; proc_nr++) {
41535           rmp = &mproc[proc_nr];
41536           rmp->mp_flags |= IN_USE;
41537           sys_getmap(proc_nr, rmp->mp_seg);
41538           if (rmp->mp_seg[T].mem_len != 0) rmp->mp_flags |= SEPARATE;
41539           minix_clicks += (rmp->mp_seg[S].mem_phys + rmp->mp_seg[S].mem_len)
41540                                 - rmp->mp_seg[T].mem_phys;
41541     }
41542     mproc[INIT_PROC_NR].mp_pid = INIT_PID;
41543     sigemptyset(&mproc[INIT_PROC_NR].mp_ignore);
41544     sigemptyset(&mproc[INIT_PROC_NR].mp_catch);
41545     procs_in_use = LOW_USER + 1;
41546
41547     /* Wait for FS to send a message telling the RAM disk size then go "on-line".
41548      */
41549     if (receive(FS_PROC_NR, &mess) != OK)
```

```
41550           panic("MM can't obtain RAM disk size from FS", NO_NUM);
41551
41552     ram_clicks = mess.m1_i1;
41553
41554     /* Initialize tables to all physical mem. */
41555     mem_init(&total_clicks, &free_clicks);
41556
41557     /* Print memory information. */
41558     printf("\nMemory size =%5dK   ", click_to_round_k(total_clicks));
41559     printf("MINIX =%4dK   ", click_to_round_k(minix_clicks));
41560     printf("RAM disk =%5dK   ", click_to_round_k(ram_clicks));
41561     printf("Available =%5dK\n\n", click_to_round_k(free_clicks));
41562
41563     /* Tell FS to continue. */
41564     if (send(FS_PROC_NR, &mess) != OK)
41565           panic("MM can't sync up with FS", NO_NUM);
41566
41567     /* Tell the memory task where my process table is for the sake of ps(1). */
41568     if ((mem = open("/dev/mem", O_RDWR)) != -1) {
41569           ioctl(mem, MIOCSPSINFO, (void *) mproc);
41570           close(mem);
41571     }
41572   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/mm/putk.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
41600   /* MM must occasionally print some message.  It uses the standard library
41601    * routine printk().  (The name "printf" is really a macro defined as
41602    * "printk"). Printing is done by calling the TTY task directly, not going
41603    * through FS.
41604    */
41605
41606   #include "mm.h"
41607   #include <minix/com.h>
41608
41609   #define BUF_SIZE          100   /* print buffer size */
41610
41611   PRIVATE int buf_count;          /* # characters in the buffer */
41612   PRIVATE char print_buf[BUF_SIZE];       /* output is buffered here */
41613   PRIVATE message putch_msg;      /* used for message to TTY task */
41614
41615   _PROTOTYPE( FORWARD void flush, (void) );
41616
41617   /*===========================================================================*
41618    *                                putk                                       *
41619    *===========================================================================*/
41620   PUBLIC void putk(c)
41621   int c;
41622   {
41623   /* Accumulate another character.  If 0 or buffer full, print it. */
41624
41625     if (c == 0 || buf_count == BUF_SIZE) flush();
41626     if (c == '\n') putk('\r');
41627     if (c != 0) print_buf[buf_count++] = c;
41628   }
```

```
41631    /*===========================================================================*
41632     *                                flush                                      *
41633     *===========================================================================*/
41634    PRIVATE void flush()
41635    {
41636    /* Flush the print buffer by calling TTY task. */
41637
41638      if (buf_count == 0) return;
41639      putch_msg.m_type = DEV_WRITE;
41640      putch_msg.PROC_NR  = 0;
41641      putch_msg.TTY_LINE = 0;
41642      putch_msg.ADDRESS  = print_buf;
41643      putch_msg.COUNT = buf_count;
41644      sendrec(TTY, &putch_msg);
41645      buf_count = 0;
41646    }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                          src/mm/signal.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
41700    /* This file handles signals, which are asynchronous events and are generally
41701     * a messy and unpleasant business.  Signals can be generated by the KILL
41702     * system call, or from the keyboard (SIGINT) or from the clock (SIGALRM).
41703     * In all cases control eventually passes to check_sig() to see which processes
41704     * can be signaled.  The actual signaling is done by sig_proc().
41705     *
41706     * The entry points into this file are:
41707     *   do_sigaction:   perform the SIGACTION system call
41708     *   do_sigpending:  perform the SIGPENDING system call
41709     *   do_sigprocmask: perform the SIGPROCMASK system call
41710     *   do_sigreturn:   perform the SIGRETURN system call
41711     *   do_sigsuspend:  perform the SIGSUSPEND system call
41712     *   do_kill:   perform the KILL system call
41713     *   do_ksig:   accept a signal originating in the kernel (e.g., SIGINT)
41714     *   do_alarm:  perform the ALARM system call by calling set_alarm()
41715     *   set_alarm: tell the clock task to start or stop a timer
41716     *   do_pause:  perform the PAUSE system call
41717     *   do_reboot: kill all processes, then reboot system
41718     *   sig_proc:  interrupt or terminate a signaled process
41719     *   check_sig: check which processes to signal with sig_proc()
41720     */
41721
41722    #include "mm.h"
41723    #include <sys/stat.h>
41724    #include <minix/callnr.h>
41725    #include <minix/com.h>
41726    #include <signal.h>
41727    #include <sys/sigcontext.h>
41728    #include <string.h>
41729    #include "mproc.h"
41730    #include "param.h"
41731
41732    #define CORE_MODE       0777    /* mode to use on core image files */
41733    #define DUMPED          0200    /* bit set in status when core dumped */
41734    #define DUMP_SIZE       ((INT_MAX / BLOCK_SIZE) * BLOCK_SIZE)
```

```
41735                                 /* buffer size for core dumps */
41736
41737   FORWARD _PROTOTYPE( void check_pending, (void)                          );
41738   FORWARD _PROTOTYPE( void dump_core, (struct mproc *rmp)                 );
41739   FORWARD _PROTOTYPE( void unpause, (int pro)                             );
41740
41741
41742   /*===========================================================================*
41743    *                                do_sigaction                               *
41744    *===========================================================================*/
41745   PUBLIC int do_sigaction()
41746   {
41747     int r;
41748     struct sigaction svec;
41749     struct sigaction *svp;
41750
41751     if (sig_nr == SIGKILL) return(OK);
41752     if (sig_nr < 1 || sig_nr > _NSIG) return (EINVAL);
41753     svp = &mp->mp_sigact[sig_nr];
41754     if ((struct sigaction *) sig_osa != (struct sigaction *) NULL) {
41755           r = sys_copy(MM_PROC_NR,D, (phys_bytes) svp,
41756                   who, D, (phys_bytes) sig_osa, (phys_bytes) sizeof(svec));
41757           if (r != OK) return(r);
41758     }
41759
41760     if ((struct sigaction *) sig_nsa == (struct sigaction *) NULL) return(OK);
41761
41762     /* Read in the sigaction structure. */
41763     r = sys_copy(who, D, (phys_bytes) sig_nsa,
41764                   MM_PROC_NR, D, (phys_bytes) &svec, (phys_bytes) sizeof(svec));
41765     if (r != OK) return(r);
41766
41767     if (svec.sa_handler == SIG_IGN) {
41768           sigaddset(&mp->mp_ignore, sig_nr);
41769           sigdelset(&mp->mp_sigpending, sig_nr);
41770           sigdelset(&mp->mp_catch, sig_nr);
41771     } else {
41772           sigdelset(&mp->mp_ignore, sig_nr);
41773           if (svec.sa_handler == SIG_DFL)
41774                   sigdelset(&mp->mp_catch, sig_nr);
41775           else
41776                   sigaddset(&mp->mp_catch, sig_nr);
41777     }
41778     mp->mp_sigact[sig_nr].sa_handler = svec.sa_handler;
41779     sigdelset(&svec.sa_mask, SIGKILL);
41780     mp->mp_sigact[sig_nr].sa_mask = svec.sa_mask;
41781     mp->mp_sigact[sig_nr].sa_flags = svec.sa_flags;
41782     mp->mp_sigreturn = (vir_bytes) sig_ret;
41783     return(OK);
41784   }

41786   /*===========================================================================*
41787    *                              do_sigpending                                *
41788    *===========================================================================*/
41789   PUBLIC int do_sigpending()
41790   {
41791     ret_mask = (long) mp->mp_sigpending;
41792     return OK;
41793   }
```

```
41795   /*===========================================================================*
41796    *                          do_sigprocmask                                   *
41797    *===========================================================================*/
41798   PUBLIC int do_sigprocmask()
41799   {
41800   /* Note that the library interface passes the actual mask in sigmask_set,
41801    * not a pointer to the mask, in order to save a sys_copy.  Similarly,
41802    * the old mask is placed in the return message which the library
41803    * interface copies (if requested) to the user specified address.
41804    *
41805    * The library interface must set SIG_INQUIRE if the 'act' argument
41806    * is NULL.
41807    */
41808
41809     int i;
41810
41811     ret_mask = (long) mp->mp_sigmask;
41812
41813     switch (sig_how) {
41814         case SIG_BLOCK:
41815           sigdelset((sigset_t *)&sig_set, SIGKILL);
41816           for (i = 1; i < _NSIG; i++) {
41817                   if (sigismember((sigset_t *)&sig_set, i))
41818                           sigaddset(&mp->mp_sigmask, i);
41819           }
41820           break;
41821
41822         case SIG_UNBLOCK:
41823           for (i = 1; i < _NSIG; i++) {
41824                   if (sigismember((sigset_t *)&sig_set, i))
41825                           sigdelset(&mp->mp_sigmask, i);
41826           }
41827           check_pending();
41828           break;
41829
41830         case SIG_SETMASK:
41831           sigdelset((sigset_t *)&sig_set, SIGKILL);
41832           mp->mp_sigmask = (sigset_t)sig_set;
41833           check_pending();
41834           break;
41835
41836         case SIG_INQUIRE:
41837           break;
41838
41839         default:
41840           return(EINVAL);
41841           break;
41842     }
41843     return OK;
41844   }

41846   /*===========================================================================*
41847    *                          do_sigsuspend                                    *
41848    *===========================================================================*/
41849   PUBLIC int do_sigsuspend()
41850   {
41851     mp->mp_sigmask2 = mp->mp_sigmask;     /* save the old mask */
41852     mp->mp_sigmask = (sigset_t) sig_set;
41853     sigdelset(&mp->mp_sigmask, SIGKILL);
41854     mp->mp_flags |= SIGSUSPENDED;
```

```
41855     dont_reply = TRUE;
41856     check_pending();
41857     return OK;
41858   }


41861   /*===========================================================================*
41862    *                                do_sigreturn                               *
41863    *===========================================================================*/
41864   PUBLIC int do_sigreturn()
41865   {
41866   /* A user signal handler is done.  Restore context and check for
41867    * pending unblocked signals.
41868    */
41869
41870     int r;
41871
41872     mp->mp_sigmask = (sigset_t) sig_set;
41873     sigdelset(&mp->mp_sigmask, SIGKILL);
41874
41875     r = sys_sigreturn(who, (vir_bytes)sig_context, sig_flags);
41876     check_pending();
41877     return(r);
41878   }

41880   /*===========================================================================*
41881    *                                do_kill                                    *
41882    *===========================================================================*/
41883   PUBLIC int do_kill()
41884   {
41885   /* Perform the kill(pid, signo) system call. */
41886
41887     return check_sig(pid, sig_nr);
41888   }


41891   /*===========================================================================*
41892    *                                do_ksig                                    *
41893    *===========================================================================*/
41894   PUBLIC int do_ksig()
41895   {
41896   /* Certain signals, such as segmentation violations and DEL, originate in the
41897    * kernel.  When the kernel detects such signals, it sets bits in a bit map.
41898    * As soon as MM is awaiting new work, the kernel sends MM a message containing
41899    * the process slot and bit map.  That message comes here.  The File System
41900    * also uses this mechanism to signal writing on broken pipes (SIGPIPE).
41901    */
41902
41903     register struct mproc *rmp;
41904     int i, proc_nr;
41905     pid_t proc_id, id;
41906     sigset_t sig_map;
41907
41908     /* Only kernel may make this call. */
41909     if (who != HARDWARE) return(EPERM);
41910     dont_reply = TRUE;            /* don't reply to the kernel */
41911     proc_nr = mm_in.SIG_PROC;
41912     rmp = &mproc[proc_nr];
41913     if ( (rmp->mp_flags & IN_USE) == 0 || (rmp->mp_flags & HANGING) ) return(OK);
41914     proc_id = rmp->mp_pid;
```

```
41915     sig_map = (sigset_t) mm_in.SIG_MAP;
41916     mp = &mproc[0];               /* pretend kernel signals are from MM */
41917     mp->mp_procgrp = rmp->mp_procgrp;     /* get process group right */
41918
41919     /* Check each bit in turn to see if a signal is to be sent.  Unlike
41920      * kill(), the kernel may collect several unrelated signals for a
41921      * process and pass them to MM in one blow.  Thus loop on the bit
41922      * map. For SIGINT and SIGQUIT, use proc_id 0 to indicate a broadcast
41923      * to the recipient's process group.  For SIGKILL, use proc_id -1 to
41924      * indicate a systemwide broadcast.
41925      */
41926     for (i = 1; i <= _NSIG; i++) {
41927           if (!sigismember(&sig_map, i)) continue;
41928           switch (i) {
41929               case SIGINT:
41930               case SIGQUIT:
41931                   id = 0; break;  /* broadcast to process group */
41932               case SIGKILL:
41933                   id = -1; break; /* broadcast to all except INIT */
41934               case SIGALRM:
41935                   /* Disregard SIGALRM when the target process has not
41936                    * requested an alarm.  This only applies for a KERNEL
41937                    * generated signal.
41938                    */
41939                   if ((rmp->mp_flags & ALARM_ON) == 0) continue;
41940                   rmp->mp_flags &= ~ALARM_ON;
41941                   /* fall through */
41942               default:
41943                   id = proc_id;
41944                   break;
41945           }
41946           check_sig(id, i);
41947           sys_endsig(proc_nr);    /* tell kernel it's done */
41948     }
41949     return(OK);
41950   }


41953   /*===========================================================================*
41954    *                              do_alarm                                     *
41955    *===========================================================================*/
41956   PUBLIC int do_alarm()
41957   {
41958   /* Perform the alarm(seconds) system call. */
41959
41960     return(set_alarm(who, seconds));
41961   }


41964   /*===========================================================================*
41965    *                              set_alarm                                    *
41966    *===========================================================================*/
41967   PUBLIC int set_alarm(proc_nr, sec)
41968   int proc_nr;                  /* process that wants the alarm */
41969   int sec;                      /* how many seconds delay before the signal */
41970   {
41971   /* This routine is used by do_alarm() to set the alarm timer.  It is also used
41972    * to turn the timer off when a process exits with the timer still on.
41973    */
41974
```

```
41975     message m_sig;
41976     int remaining;
41977
41978     if (sec != 0)
41979           mproc[proc_nr].mp_flags |= ALARM_ON;
41980     else
41981           mproc[proc_nr].mp_flags &= ~ALARM_ON;
41982
41983     /* Tell the clock task to provide a signal message when the time comes.
41984      *
41985      * Large delays cause a lot of problems.  First, the alarm system call
41986      * takes an unsigned seconds count and the library has cast it to an int.
41987      * That probably works, but on return the library will convert "negative"
41988      * unsigneds to errors.  Presumably no one checks for these errors, so
41989      * force this call through.  Second, If unsigned and long have the same
41990      * size, converting from seconds to ticks can easily overflow.  Finally,
41991      * the kernel has similar overflow bugs adding ticks.
41992      *
41993      * Fixing this requires a lot of ugly casts to fit the wrong interface
41994      * types and to avoid overflow traps.  DELTA_TICKS has the right type
41995      * (clock_t) although it is declared as long.  How can variables like
41996      * this be declared properly without combinatorial explosion of message
41997      * types?
41998      */
41999     m_sig.m_type = SET_ALARM;
42000     m_sig.CLOCK_PROC_NR = proc_nr;
42001     m_sig.DELTA_TICKS = (clock_t) (HZ * (unsigned long) (unsigned) sec);
42002     if ( (unsigned long) m_sig.DELTA_TICKS / HZ != (unsigned) sec)
42003           m_sig.DELTA_TICKS = LONG_MAX;   /* eternity (really CLOCK_T_MAX) */
42004     if (sendrec(CLOCK, &m_sig) != OK) panic("alarm er", NO_NUM);
42005     remaining = (int) m_sig.SECONDS_LEFT;
42006     if (remaining != m_sig.SECONDS_LEFT || remaining < 0)
42007           remaining = INT_MAX;    /* true value is not representable */
42008     return(remaining);
42009   }


42012   /*===========================================================================*
42013    *                                do_pause                                   *
42014    *===========================================================================*/
42015   PUBLIC int do_pause()
42016   {
42017   /* Perform the pause() system call. */
42018
42019     mp->mp_flags |= PAUSED;
42020     dont_reply = TRUE;
42021     return(OK);
42022   }


42025   /*===========================================================================*
42026    *                             do_reboot                                     *
42027    *===========================================================================*/
42028   PUBLIC int do_reboot()
42029   {
42030     register struct mproc *rmp = mp;
42031     char monitor_code[64];
42032
42033     if (rmp->mp_effuid != SUPER_USER)   return EPERM;
42034
```

```
42035     switch (reboot_flag) {
42036     case RBT_HALT:
42037     case RBT_REBOOT:
42038     case RBT_PANIC:
42039     case RBT_RESET:
42040           break;
42041     case RBT_MONITOR:
42042           if (reboot_size > sizeof(monitor_code)) return EINVAL;
42043           memset(monitor_code, 0, sizeof(monitor_code));
42044           if (sys_copy(who, D, (phys_bytes) reboot_code,
42045                   MM_PROC_NR, D, (phys_bytes) monitor_code,
42046                   (phys_bytes) reboot_size) != OK) return EFAULT;
42047           if (monitor_code[sizeof(monitor_code)-1] != 0) return EINVAL;
42048           break;
42049     default:
42050           return EINVAL;
42051     }
42052
42053     /* Kill all processes except init. */
42054     check_sig(-1, SIGKILL);
42055
42056     tell_fs(EXIT, INIT_PROC_NR, 0, 0);    /* cleanup init */
42057
42058     tell_fs(SYNC,0,0,0);
42059
42060     sys_abort(reboot_flag, monitor_code);
42061     /* NOTREACHED */
42062   }


42065   /*===========================================================================*
42066    *                              sig_proc                                     *
42067    *===========================================================================*/
42068   PUBLIC void sig_proc(rmp, signo)
42069   register struct mproc *rmp;     /* pointer to the process to be signaled */
42070   int signo;                      /* signal to send to process (1 to _NSIG) */
42071   {
42072   /* Send a signal to a process.  Check to see if the signal is to be caught,
42073    * ignored, or blocked.  If the signal is to be caught, coordinate with
42074    * KERNEL to push a sigcontext structure and a sigframe structure onto
42075    * the catcher's stack.  Also, KERNEL will reset the program counter and
42076    * stack pointer, so that when the process next runs, it will be executing
42077    * the signal handler.  When the signal handler returns,  sigreturn(2)
42078    * will be called.  Then KERNEL will restore the signal context from the
42079    * sigcontext structure.
42080    *
42081    * If there is insufficient stack space, kill the process.
42082    */
42083
42084     vir_bytes new_sp;
42085     int slot;
42086     int sigflags;
42087     struct sigmsg sm;
42088
42089     slot = (int) (rmp - mproc);
42090     if (!(rmp->mp_flags & IN_USE)) {
42091           printf("MM: signal %d sent to dead process %d\n", signo, slot);
42092           panic("", NO_NUM);
42093     }
42094     if (rmp->mp_flags & HANGING) {
```

```
42095           printf("MM: signal %d sent to HANGING process %d\n", signo, slot);
42096           panic("", NO_NUM);
42097       }
42098       if (rmp->mp_flags & TRACED && signo != SIGKILL) {
42099           /* A traced process has special handling. */
42100           unpause(slot);
42101           stop_proc(rmp, signo);  /* a signal causes it to stop */
42102           return;
42103       }
42104       /* Some signals are ignored by default. */
42105       if (sigismember(&rmp->mp_ignore, signo)) return;
42106
42107       if (sigismember(&rmp->mp_sigmask, signo)) {
42108           /* Signal should be blocked. */
42109           sigaddset(&rmp->mp_sigpending, signo);
42110           return;
42111       }
42112       sigflags = rmp->mp_sigact[signo].sa_flags;
42113       if (sigismember(&rmp->mp_catch, signo)) {
42114           if (rmp->mp_flags & SIGSUSPENDED)
42115                   sm.sm_mask = rmp->mp_sigmask2;
42116           else
42117                   sm.sm_mask = rmp->mp_sigmask;
42118           sm.sm_signo = signo;
42119           sm.sm_sighandler = (vir_bytes) rmp->mp_sigact[signo].sa_handler;
42120           sm.sm_sigreturn = rmp->mp_sigreturn;
42121           sys_getsp(slot, &new_sp);
42122           sm.sm_stkptr = new_sp;
42123
42124           /* Make room for the sigcontext and sigframe struct. */
42125           new_sp -= sizeof(struct sigcontext)
42126                                   + 3 * sizeof(char *) + 2 * sizeof(int);
42127
42128           if (adjust(rmp, rmp->mp_seg[D].mem_len, new_sp) != OK)
42129                   goto doterminate;
42130
42131           rmp->mp_sigmask |= rmp->mp_sigact[signo].sa_mask;
42132           if (sigflags & SA_NODEFER)
42133                   sigdelset(&rmp->mp_sigmask, signo);
42134           else
42135                   sigaddset(&rmp->mp_sigmask, signo);
42136
42137           if (sigflags & SA_RESETHAND) {
42138                   sigdelset(&rmp->mp_catch, signo);
42139                   rmp->mp_sigact[signo].sa_handler = SIG_DFL;
42140           }
42141
42142           sys_sendsig(slot, &sm);
42143           sigdelset(&rmp->mp_sigpending, signo);
42144           /* If process is hanging on PAUSE, WAIT, SIGSUSPEND, tty, pipe, etc.,
42145            * release it.
42146            */
42147           unpause(slot);
42148           return;
42149       }
42150   doterminate:
42151       /* Signal should not or cannot be caught.  Terminate the process. */
42152       rmp->mp_sigstatus = (char) signo;
42153       if (sigismember(&core_sset, signo)) {
42154           /* Switch to the user's FS environment and dump core. */
```

```
42155           tell_fs(CHDIR, slot, FALSE, 0);
42156           dump_core(rmp);
42157     }
42158     mm_exit(rmp, 0);              /* terminate process */
42159   }


42162   /*===========================================================================*
42163    *                              check_sig                                    *
42164    *===========================================================================*/
42165   PUBLIC int check_sig(proc_id, signo)
42166   pid_t proc_id;                  /* pid of proc to sig, or 0 or -1, or -pgrp */
42167   int signo;                      /* signal to send to process (0 to _NSIG) */
42168   {
42169   /* Check to see if it is possible to send a signal.  The signal may have to be
42170    * sent to a group of processes.  This routine is invoked by the KILL system
42171    * call, and also when the kernel catches a DEL or other signal.
42172    */
42173
42174     register struct mproc *rmp;
42175     int count;                    /* count # of signals sent */
42176     int error_code;
42177
42178     if (signo < 0 || signo > _NSIG) return(EINVAL);
42179
42180     /* Return EINVAL for attempts to send SIGKILL to INIT alone. */
42181     if (proc_id == INIT_PID && signo == SIGKILL) return(EINVAL);
42182
42183     /* Search the proc table for processes to signal.  (See forkexit.c about
42184      * pid magic.)
42185      */
42186     count = 0;
42187     error_code = ESRCH;
42188     for (rmp = &mproc[INIT_PROC_NR]; rmp < &mproc[NR_PROCS]; rmp++) {
42189           if ( (rmp->mp_flags & IN_USE) == 0) continue;
42190           if (rmp->mp_flags & HANGING && signo != 0) continue;
42191
42192           /* Check for selection. */
42193           if (proc_id > 0 && proc_id != rmp->mp_pid) continue;
42194           if (proc_id == 0 && mp->mp_procgrp != rmp->mp_procgrp) continue;
42195           if (proc_id == -1 && rmp->mp_pid == INIT_PID) continue;
42196           if (proc_id < -1 && rmp->mp_procgrp != -proc_id) continue;
42197
42198           /* Check for permission. */
42199           if (mp->mp_effuid != SUPER_USER
42200               && mp->mp_realuid != rmp->mp_realuid
42201               && mp->mp_effuid != rmp->mp_realuid
42202               && mp->mp_realuid != rmp->mp_effuid
42203               && mp->mp_effuid != rmp->mp_effuid) {
42204                   error_code = EPERM;
42205                   continue;
42206           }
42207
42208           count++;
42209           if (signo == 0) continue;
42210
42211           /* 'sig_proc' will handle the disposition of the signal.  The
42212            * signal may be caught, blocked, ignored, or cause process
42213            * termination, possibly with core dump.
42214            */
```

```
42215           sig_proc(rmp, signo);
42216
42217           if (proc_id > 0) break; /* only one process being signaled */
42218       }
42219
42220       /* If the calling process has killed itself, don't reply. */
42221       if ((mp->mp_flags & IN_USE) == 0 || (mp->mp_flags & HANGING))
42222           dont_reply = TRUE;
42223       return(count > 0 ? OK : error_code);
42224   }


42227   /*===========================================================================*
42228    *                              check_pending                                *
42229    *===========================================================================*/
42230   PRIVATE void check_pending()
42231   {
42232       /* Check to see if any pending signals have been unblocked.  The
42233        * first such signal found is delivered.
42234        *
42235        * If multiple pending unmasked signals are found, they will be
42236        * delivered sequentially.
42237        *
42238        * There are several places in this file where the signal mask is
42239        * changed.  At each such place, check_pending() should be called to
42240        * check for newly unblocked signals.
42241        */
42242
42243       int i;
42244
42245       for (i = 1; i < _NSIG; i++) {
42246           if (sigismember(&mp->mp_sigpending, i) &&
42247                   !sigismember(&mp->mp_sigmask, i)) {
42248               sigdelset(&mp->mp_sigpending, i);
42249               sig_proc(mp, i);
42250               break;
42251           }
42252       }
42253   }


42256   /*===========================================================================*
42257    *                                unpause                                    *
42258    *===========================================================================*/
42259   PRIVATE void unpause(pro)
42260   int pro;                        /* which process number */
42261   {
42262   /* A signal is to be sent to a process.  If that process is hanging on a
42263    * system call, the system call must be terminated with EINTR.  Possible
42264    * calls are PAUSE, WAIT, READ and WRITE, the latter two for pipes and ttys.
42265    * First check if the process is hanging on an MM call.  If not, tell FS,
42266    * so it can check for READs and WRITEs from pipes, ttys and the like.
42267    */
42268
42269       register struct mproc *rmp;
42270
42271       rmp = &mproc[pro];
42272
42273       /* Check to see if process is hanging on a PAUSE call. */
42274       if ( (rmp->mp_flags & PAUSED) && (rmp->mp_flags & HANGING) == 0) {
```

```
42275           rmp->mp_flags &= ~PAUSED;
42276           reply(pro, EINTR, 0, NIL_PTR);
42277           return;
42278     }
42279
42280     /* Check to see if process is hanging on a WAIT call. */
42281     if ( (rmp->mp_flags & WAITING) && (rmp->mp_flags & HANGING) == 0) {
42282           rmp->mp_flags &= ~WAITING;
42283           reply(pro, EINTR, 0, NIL_PTR);
42284           return;
42285     }
42286
42287     /* Check to see if process is hanging on a SIGSUSPEND call. */
42288     if ((rmp->mp_flags & SIGSUSPENDED) && (rmp->mp_flags & HANGING) == 0) {
42289           rmp->mp_flags &= ~SIGSUSPENDED;
42290           reply(pro, EINTR, 0, NIL_PTR);
42291           return;
42292     }
42293
42294     /* Process is not hanging on an MM call.  Ask FS to take a look. */
42295           tell_fs(UNPAUSE, pro, 0, 0);
42296   }


42299   /*===========================================================================*
42300    *                              dump_core                                    *
42301    *===========================================================================*/
42302   PRIVATE void dump_core(rmp)
42303   register struct mproc *rmp;     /* whose core is to be dumped */
42304   {
42305   /* Make a core dump on the file "core", if possible. */
42306
42307     int fd, fake_fd, nr_written, seg, slot;
42308     char *buf;
42309     vir_bytes current_sp;
42310     phys_bytes left;              /* careful; 64K might overflow vir_bytes */
42311     unsigned nr_to_write;         /* unsigned for arg to write() but < INT_MAX */
42312     long trace_data, trace_off;
42313
42314     slot = (int) (rmp - mproc);
42315
42316     /* Can core file be written?  We are operating in the user's FS environment,
42317      * so no special permission checks are needed.
42318      */
42319     if (rmp->mp_realuid != rmp->mp_effuid) return;
42320     if ( (fd = creat(core_name, CORE_MODE)) < 0) return;
42321     rmp->mp_sigstatus |= DUMPED;
42322
42323     /* Make sure the stack segment is up to date.
42324      * We don't want adjust() to fail unless current_sp is preposterous,
42325      * but it might fail due to safety checking.  Also, we don't really want
42326      * the adjust() for sending a signal to fail due to safety checking.
42327      * Maybe make SAFETY_BYTES a parameter.
42328      */
42329     sys_getsp(slot, &current_sp);
42330     adjust(rmp, rmp->mp_seg[D].mem_len, current_sp);
42331
42332     /* Write the memory map of all segments to begin the core file. */
42333     if (write(fd, (char *) rmp->mp_seg, (unsigned) sizeof rmp->mp_seg)
42334         != (unsigned) sizeof rmp->mp_seg) {
```

```
42335           close(fd);
42336           return;
42337       }
42338
42339       /* Write out the whole kernel process table entry to get the regs. */
42340       trace_off = 0;
42341       while (sys_trace(3, slot, trace_off, &trace_data) == OK) {
42342           if (write(fd, (char *) &trace_data, (unsigned) sizeof (long))
42343               != (unsigned) sizeof (long)) {
42344                   close(fd);
42345                   return;
42346           }
42347           trace_off += sizeof (long);
42348       }
42349
42350       /* Loop through segments and write the segments themselves out. */
42351       for (seg = 0; seg < NR_SEGS; seg++) {
42352           buf = (char *) ((vir_bytes) rmp->mp_seg[seg].mem_vir << CLICK_SHIFT);
42353           left = (phys_bytes) rmp->mp_seg[seg].mem_len << CLICK_SHIFT;
42354           fake_fd = (slot << 8) | (seg << 6) | fd;
42355
42356           /* Loop through a segment, dumping it. */
42357           while (left != 0) {
42358                   nr_to_write = (unsigned) MIN(left, DUMP_SIZE);
42359                   if ( (nr_written = write(fake_fd, buf, nr_to_write)) < 0) {
42360                           close(fd);
42361                           return;
42362                   }
42363                   buf += nr_written;
42364                   left -= nr_written;
42365           }
42366       }
42367       close(fd);
42368   }
```

++
src/mm/table.c
++

```
42400   /* This file contains the table used to map system call numbers onto the
42401    * routines that perform them.
42402    */
42403
42404   #define _TABLE
42405
42406   #include "mm.h"
42407   #include <minix/callnr.h>
42408   #include <signal.h>
42409   #include "mproc.h"
42410   #include "param.h"
42411
42412   /* Miscellaneous */
42413   char core_name[] = "core";      /* file name where core images are produced */
42414
42415   _PROTOTYPE (int (*call_vec[NCALLS]), (void) ) = {
42416           no_sys,         /*  0 = unused  */
42417           do_mm_exit,     /*  1 = exit    */
42418           do_fork,        /*  2 = fork    */
42419           no_sys,         /*  3 = read    */
```

```
42420           no_sys,         /*  4 = write   */
42421           no_sys,         /*  5 = open    */
42422           no_sys,         /*  6 = close   */
42423           do_waitpid,     /*  7 = wait    */
42424           no_sys,         /*  8 = creat   */
42425           no_sys,         /*  9 = link    */
42426           no_sys,         /* 10 = unlink  */
42427           do_waitpid,     /* 11 = waitpid */
42428           no_sys,         /* 12 = chdir   */
42429           no_sys,         /* 13 = time    */
42430           no_sys,         /* 14 = mknod   */
42431           no_sys,         /* 15 = chmod   */
42432           no_sys,         /* 16 = chown   */
42433           do_brk,         /* 17 = break   */
42434           no_sys,         /* 18 = stat    */
42435           no_sys,         /* 19 = lseek   */
42436           do_getset,      /* 20 = getpid  */
42437           no_sys,         /* 21 = mount   */
42438           no_sys,         /* 22 = umount  */
42439           do_getset,      /* 23 = setuid  */
42440           do_getset,      /* 24 = getuid  */
42441           no_sys,         /* 25 = stime   */
42442           do_trace,       /* 26 = ptrace  */
42443           do_alarm,       /* 27 = alarm   */
42444           no_sys,         /* 28 = fstat   */
42445           do_pause,       /* 29 = pause   */
42446           no_sys,         /* 30 = utime   */
42447           no_sys,         /* 31 = (stty)  */
42448           no_sys,         /* 32 = (gtty)  */
42449           no_sys,         /* 33 = access  */
42450           no_sys,         /* 34 = (nice)  */
42451           no_sys,         /* 35 = (ftime) */
42452           no_sys,         /* 36 = sync    */
42453           do_kill,        /* 37 = kill    */
42454           no_sys,         /* 38 = rename  */
42455           no_sys,         /* 39 = mkdir   */
42456           no_sys,         /* 40 = rmdir   */
42457           no_sys,         /* 41 = dup     */
42458           no_sys,         /* 42 = pipe    */
42459           no_sys,         /* 43 = times   */
42460           no_sys,         /* 44 = (prof)  */
42461           no_sys,         /* 45 = unused  */
42462           do_getset,      /* 46 = setgid  */
42463           do_getset,      /* 47 = getgid  */
42464           no_sys,         /* 48 = (signal)*/
42465           no_sys,         /* 49 = unused  */
42466           no_sys,         /* 50 = unused  */
42467           no_sys,         /* 51 = (acct)  */
42468           no_sys,         /* 52 = (phys)  */
42469           no_sys,         /* 53 = (lock)  */
42470           no_sys,         /* 54 = ioctl   */
42471           no_sys,         /* 55 = fcntl   */
42472           no_sys,         /* 56 = (mpx)   */
42473           no_sys,         /* 57 = unused  */
42474           no_sys,         /* 58 = unused  */
42475           do_exec,        /* 59 = execve  */
42476           no_sys,         /* 60 = umask   */
42477           no_sys,         /* 61 = chroot  */
42478           do_getset,      /* 62 = setsid  */
42479           do_getset,      /* 63 = getpgrp */
```

```
42480
42481           do_ksig,        /* 64 = KSIG: signals originating in the kernel */
42482           no_sys,         /* 65 = UNPAUSE */
42483           no_sys,         /* 66 = unused  */
42484           no_sys,         /* 67 = REVIVE  */
42485           no_sys,         /* 68 = TASK_REPLY  */
42486           no_sys,         /* 69 = unused  */
42487           no_sys,         /* 70 = unused  */
42488           do_sigaction,   /* 71 = sigaction   */
42489           do_sigsuspend,  /* 72 = sigsuspend  */
42490           do_sigpending,  /* 73 = sigpending  */
42491           do_sigprocmask, /* 74 = sigprocmask */
42492           do_sigreturn,   /* 75 = sigreturn   */
42493           do_reboot,      /* 76 = reboot  */
42494   };
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/mm/trace.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
42500   /* This file handles the memory manager's part of debugging, using the
42501    * ptrace system call. Most of the commands are passed on to the system
42502    * task for completion.
42503    *
42504    * The debugging commands available are:
42505    * T_STOP       stop the process
42506    * T_OK         enable tracing by parent for this process
42507    * T_GETINS     return value from instruction space
42508    * T_GETDATA    return value from data space
42509    * T_GETUSER    return value from user process table
42510    * T_SETINS     set value in instruction space
42511    * T_SETDATA    set value in data space
42512    * T_SETUSER    set value in user process table
42513    * T_RESUME     resume execution
42514    * T_EXIT       exit
42515    * T_STEP       set trace bit
42516    *
42517    * The T_OK and T_EXIT commands are handled here, and the T_RESUME and
42518    * T_STEP commands are partially handled here and completed by the system
42519    * task. The rest are handled entirely by the system task.
42520    */
42521
42522   #include "mm.h"
42523   #include <sys/ptrace.h>
42524   #include <signal.h>
42525   #include "mproc.h"
42526   #include "param.h"
42527
42528   #define NIL_MPROC       ((struct mproc *) 0)
42529
42530   FORWARD _PROTOTYPE( struct mproc *findproc, (pid_t lpid) );
42531
42532   /*===========================================================================*
42533    *                              do_trace                                     *
42534    *===========================================================================*/
```

```
42535   PUBLIC int do_trace()
42536   {
42537     register struct mproc *child;
42538
42539     /* the T_OK call is made by the child fork of the debugger before it execs
42540      * the process to be traced
42541      */
42542     if (request == T_OK) {/* enable tracing by parent for this process */
42543           mp->mp_flags |= TRACED;
42544           mm_out.m2_l2 = 0;
42545           return(OK);
42546     }
42547     if ((child = findproc(pid)) == NIL_MPROC || !(child->mp_flags & STOPPED)) {
42548           return(ESRCH);
42549     }
42550     /* all the other calls are made by the parent fork of the debugger to
42551      * control execution of the child
42552      */
42553     switch (request) {
42554     case T_EXIT:          /* exit */
42555           mm_exit(child, (int)data);
42556           mm_out.m2_l2 = 0;
42557           return(OK);
42558     case T_RESUME:
42559     case T_STEP:          /* resume execution */
42560           if (data < 0 || data > _NSIG) return(EIO);
42561           if (data > 0) {         /* issue signal */
42562                   child->mp_flags &= ~TRACED;  /* so signal is not diverted */
42563                   sig_proc(child, (int) data);
42564                   child->mp_flags |= TRACED;
42565           }
42566           child->mp_flags &= ~STOPPED;
42567           break;
42568     }
42569     if (sys_trace(request, (int) (child - mproc), taddr, &data) != OK)
42570           return(-errno);
42571     mm_out.m2_l2 = data;
42572     return(OK);
42573   }

42575   /*===========================================================================*
42576    *                              findproc                                     *
42577    *===========================================================================*/
42578   PRIVATE struct mproc *findproc(lpid)
42579   pid_t lpid;
42580   {
42581     register struct mproc *rmp;
42582
42583     for (rmp = &mproc[INIT_PROC_NR + 1]; rmp < &mproc[NR_PROCS]; rmp++)
42584           if (rmp->mp_flags & IN_USE && rmp->mp_pid == lpid) return(rmp);
42585     return(NIL_MPROC);
42586   }

42588   /*===========================================================================*
42589    *                              stop_proc                                    *
42590    *===========================================================================*/
42591   PUBLIC void stop_proc(rmp, signo)
42592   register struct mproc *rmp;
42593   int signo;
42594   {
```

```
42595    /* A traced process got a signal so stop it. */
42596
42597      register struct mproc *rpmp = mproc + rmp->mp_parent;
42598
42599      if (sys_trace(-1, (int) (rmp - mproc), 0L, (long *) 0) != OK) return;
42600      rmp->mp_flags |= STOPPED;
42601      if (rpmp->mp_flags & WAITING) {
42602            rpmp->mp_flags &= ~WAITING;     /* parent is no longer waiting */
42603            reply(rmp->mp_parent, rmp->mp_pid, 0177 | (signo << 8), NIL_PTR);
42604      } else {
42605            rmp->mp_sigstatus = signo;
42606      }
42607      return;
42608    }


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/mm/utility.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

42700    /* This file contains some utility routines for MM.
42701     *
42702     * The entry points are:
42703     *   allowed:   see if an access is permitted
42704     *   no_sys:    this routine is called for invalid system call numbers
42705     *   panic:     MM has run aground of a fatal error and cannot continue
42706     *   tell_fs:   interface to FS
42707     */
42708
42709    #include "mm.h"
42710    #include <sys/stat.h>
42711    #include <minix/callnr.h>
42712    #include <minix/com.h>
42713    #include <fcntl.h>
42714    #include <signal.h>             /* needed only because mproc.h needs it */
42715    #include "mproc.h"
42716
42717    /*===========================================================================*
42718     *                              allowed                                      *
42719     *===========================================================================*/
42720    PUBLIC int allowed(name_buf, s_buf, mask)
42721    char *name_buf;                 /* pointer to file name to be EXECed */
42722    struct stat *s_buf;             /* buffer for doing and returning stat struct*/
42723    int mask;                       /* R_BIT, W_BIT, or X_BIT */
42724    {
42725    /* Check to see if file can be accessed.  Return EACCES or ENOENT if the access
42726     * is prohibited.  If it is legal open the file and return a file descriptor.
42727     */
42728
42729      int fd;
42730      int save_errno;
42731
42732      /* Use the fact that mask for access() is the same as the permissions mask.
42733       * E.g., X_BIT in <minix/const.h> is the same as X_OK in <unistd.h> and
42734       * S_IXOTH in <sys/stat.h>.  tell_fs(DO_CHDIR, ...) has set MM's real ids
42735       * to the user's effective ids, so access() works right for setuid programs.
42736       */
42737      if (access(name_buf, mask) < 0) return(-errno);
42738
42739      /* The file is accessible but might not be readable.  Make it readable. */
```

```
42740     tell_fs(SETUID, MM_PROC_NR, (int) SUPER_USER, (int) SUPER_USER);
42741
42742     /* Open the file and fstat it.  Restore the ids early to handle errors. */
42743     fd = open(name_buf, O_RDONLY);
42744     save_errno = errno;           /* open might fail, e.g. from ENFILE */
42745     tell_fs(SETUID, MM_PROC_NR, (int) mp->mp_effuid, (int) mp->mp_effuid);
42746     if (fd < 0) return(-save_errno);
42747     if (fstat(fd, s_buf) < 0) panic("allowed: fstat failed", NO_NUM);
42748
42749     /* Only regular files can be executed. */
42750     if (mask == X_BIT && (s_buf->st_mode & I_TYPE) != I_REGULAR) {
42751           close(fd);
42752           return(EACCES);
42753     }
42754     return(fd);
42755   }


42758   /*===========================================================================*
42759    *                              no_sys                                       *
42760    *===========================================================================*/
42761   PUBLIC int no_sys()
42762   {
42763   /* A system call number not implemented by MM has been requested. */
42764
42765     return(EINVAL);
42766   }


42769   /*===========================================================================*
42770    *                              panic                                        *
42771    *===========================================================================*/
42772   PUBLIC void panic(format, num)
42773   char *format;                 /* format string */
42774   int num;                      /* number to go with format string */
42775   {
42776   /* Something awful has happened.  Panics are caused when an internal
42777    * inconsistency is detected, e.g., a programming error or illegal value of a
42778    * defined constant.
42779    */
42780
42781     printf("Memory manager panic: %s ", format);
42782     if (num != NO_NUM) printf("%d",num);
42783     printf("\n");
42784     tell_fs(SYNC, 0, 0, 0);       /* flush the cache to the disk */
42785     sys_abort(RBT_PANIC);
42786   }


42789   /*===========================================================================*
42790    *                              tell_fs                                      *
42791    *===========================================================================*/
42792   PUBLIC void tell_fs(what, p1, p2, p3)
42793   int what, p1, p2, p3;
42794   {
42795   /* This routine is only used by MM to inform FS of certain events:
42796    *      tell_fs(CHDIR, slot, dir, 0)
42797    *      tell_fs(EXEC, proc, 0, 0)
42798    *      tell_fs(EXIT, proc, 0, 0)
42799    *      tell_fs(FORK, parent, child, pid)
```

```
42800     *      tell_fs(SETGID, proc, realgid, effgid)
42801     *      tell_fs(SETSID, proc, 0, 0)
42802     *      tell_fs(SETUID, proc, realuid, effuid)
42803     *      tell_fs(SYNC, 0, 0, 0)
42804     *      tell_fs(UNPAUSE, proc, signr, 0)
42805     */
42806
42807      message m;
42808
42809      m.m1_i1 = p1;
42810      m.m1_i2 = p2;
42811      m.m1_i3 = p3;
42812      _taskcall(FS_PROC_NR, what, &m);
42813    }


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/buf.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

42900    /* Buffer (block) cache.  To acquire a block, a routine calls get_block(),
42901     * telling which block it wants.  The block is then regarded as "in use"
42902     * and has its 'b_count' field incremented.  All the blocks that are not
42903     * in use are chained together in an LRU list, with 'front' pointing
42904     * to the least recently used block, and 'rear' to the most recently used
42905     * block.  A reverse chain, using the field b_prev is also maintained.
42906     * Usage for LRU is measured by the time the put_block() is done.  The second
42907     * parameter to put_block() can violate the LRU order and put a block on the
42908     * front of the list, if it will probably not be needed soon.  If a block
42909     * is modified, the modifying routine must set b_dirt to DIRTY, so the block
42910     * will eventually be rewritten to the disk.
42911     */
42912
42913    #include <sys/dir.h>                    /* need struct direct */
42914
42915    EXTERN struct buf {
42916      /* Data portion of the buffer. */
42917      union {
42918        char b__data[BLOCK_SIZE];               /* ordinary user data */
42919        struct direct b__dir[NR_DIR_ENTRIES];   /* directory block */
42920        zone1_t b__v1_ind[V1_INDIRECTS];        /* V1 indirect block */
42921        zone_t  b__v2_ind[V2_INDIRECTS];        /* V2 indirect block */
42922        d1_inode b__v1_ino[V1_INODES_PER_BLOCK]; /* V1 inode block */
42923        d2_inode b__v2_ino[V2_INODES_PER_BLOCK]; /* V2 inode block */
42924        bitchunk_t b__bitmap[BITMAP_CHUNKS];    /* bit map block */
42925      } b;
42926
42927      /* Header portion of the buffer. */
42928      struct buf *b_next;           /* used to link all free bufs in a chain */
42929      struct buf *b_prev;           /* used to link all free bufs the other way */
42930      struct buf *b_hash;           /* used to link bufs on hash chains */
42931      block_t b_blocknr;            /* block number of its (minor) device */
42932      dev_t b_dev;                  /* major | minor device where block resides */
42933      char b_dirt;                  /* CLEAN or DIRTY */
42934      char b_count;                 /* number of users of this buffer */
42935    } buf[NR_BUFS];
42936
42937    /* A block is free if b_dev == NO_DEV. */
42938
42939    #define NIL_BUF ((struct buf *) 0)      /* indicates absence of a buffer */
```

```
42940
42941    /* These defs make it possible to use to bp->b_data instead of bp->b.b__data */
42942    #define b_data   b.b__data
42943    #define b_dir    b.b__dir
42944    #define b_v1_ind b.b__v1_ind
42945    #define b_v2_ind b.b__v2_ind
42946    #define b_v1_ino b.b__v1_ino
42947    #define b_v2_ino b.b__v2_ino
42948    #define b_bitmap b.b__bitmap
42949
42950    EXTERN struct buf *buf_hash[NR_BUF_HASH];       /* the buffer hash table */
42951
42952    EXTERN struct buf *front;       /* points to least recently used free block */
42953    EXTERN struct buf *rear;        /* points to most recently used free block */
42954    EXTERN int bufs_in_use;         /* # bufs currently in use (not on free list)*/
42955
42956    /* When a block is released, the type of usage is passed to put_block(). */
42957    #define WRITE_IMMED        0100 /* block should be written to disk now */
42958    #define ONE_SHOT           0200 /* set if block not likely to be needed soon */
42959
42960    #define INODE_BLOCK       (0 + MAYBE_WRITE_IMMED)        /* inode block */
42961    #define DIRECTORY_BLOCK   (1 + MAYBE_WRITE_IMMED)        /* directory block */
42962    #define INDIRECT_BLOCK    (2 + MAYBE_WRITE_IMMED)        /* pointer block */
42963    #define MAP_BLOCK         (3 + MAYBE_WRITE_IMMED)        /* bit map */
42964    #define ZUPER_BLOCK       (4 + WRITE_IMMED + ONE_SHOT)   /* super block */
42965    #define FULL_DATA_BLOCK    5                             /* data, fully used */
42966    #define PARTIAL_DATA_BLOCK 6                             /* data, partly used*/
42967
42968    #define HASH_MASK (NR_BUF_HASH - 1)     /* mask for hashing block numbers */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/const.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

43000    /* Tables sizes */
43001    #define V1_NR_DZONES       7    /* # direct zone numbers in a V1 inode */
43002    #define V1_NR_TZONES       9    /* total # zone numbers in a V1 inode */
43003    #define V2_NR_DZONES       7    /* # direct zone numbers in a V2 inode */
43004    #define V2_NR_TZONES      10    /* total # zone numbers in a V2 inode */
43005
43006    #define NR_FILPS         128    /* # slots in filp table */
43007    #define NR_INODES         64    /* # slots in "in core" inode table */
43008    #define NR_SUPERS          8    /* # slots in super block table */
43009    #define NR_LOCKS           8    /* # slots in the file locking table */
43010
43011    /* The type of sizeof may be (unsigned) long.  Use the following macro for
43012     * taking the sizes of small objects so that there are no surprises like
43013     * (small) long constants being passed to routines expecting an int.
43014     */
43015    #define usizeof(t) ((unsigned) sizeof(t))
43016
43017    /* File system types. */
43018    #define SUPER_MAGIC   0x137F    /* magic number contained in super-block */
43019    #define SUPER_REV     0x7F13    /* magic # when 68000 disk read on PC or vv */
43020    #define SUPER_V2      0x2468    /* magic # for V2 file systems */
43021    #define SUPER_V2_REV  0x6824    /* V2 magic written on PC, read on 68K or vv */
43022
43023    #define V1                1    /* version number of V1 file systems */
43024    #define V2                2    /* version number of V2 file systems */
```

```
43025
43026   /* Miscellaneous constants */
43027   #define SU_UID   ((uid_t) 0)    /* super_user's uid_t */
43028   #define SYS_UID  ((uid_t) 0)    /* uid_t for processes MM and INIT */
43029   #define SYS_GID  ((gid_t) 0)    /* gid_t for processes MM and INIT */
43030   #define NORMAL             0    /* forces get_block to do disk read */
43031   #define NO_READ            1    /* prevents get_block from doing disk read */
43032   #define PREFETCH           2    /* tells get_block not to read or mark dev */
43033
43034   #define XPIPE  (-NR_TASKS-1)    /* used in fp_task when susp'd on pipe */
43035   #define XOPEN  (-NR_TASKS-2)    /* used in fp_task when susp'd on open */
43036   #define XLOCK  (-NR_TASKS-3)    /* used in fp_task when susp'd on lock */
43037   #define XPOPEN (-NR_TASKS-4)    /* used in fp_task when susp'd on pipe open */
43038
43039   #define NO_BIT   ((bit_t) 0)    /* returned by alloc_bit() to signal failure */
43040
43041   #define DUP_MASK        0100    /* mask to distinguish dup2 from dup */
43042
43043   #define LOOK_UP            0    /* tells search_dir to lookup string */
43044   #define ENTER              1    /* tells search_dir to make dir entry */
43045   #define DELETE             2    /* tells search_dir to delete entry */
43046   #define IS_EMPTY           3    /* tells search_dir to ret. OK or ENOTEMPTY */
43047
43048   #define CLEAN              0    /* disk and memory copies identical */
43049   #define DIRTY              1    /* disk and memory copies differ */
43050   #define ATIME            002    /* set if atime field needs updating */
43051   #define CTIME            004    /* set if ctime field needs updating */
43052   #define MTIME            010    /* set if mtime field needs updating */
43053
43054   #define BYTE_SWAP          0    /* tells conv2/conv4 to swap bytes */
43055   #define DONT_SWAP          1    /* tells conv2/conv4 not to swap bytes */
43056
43057   #define END_OF_FILE   (-104)    /* eof detected */
43058
43059   #define ROOT_INODE         1    /* inode number for root directory */
43060   #define BOOT_BLOCK  ((block_t) 0)       /* block number of boot block */
43061   #define SUPER_BLOCK ((block_t) 1)       /* block number of super block */
43062
43063   #define DIR_ENTRY_SIZE       usizeof (struct direct)  /* # bytes/dir entry   */
43064   #define NR_DIR_ENTRIES   (BLOCK_SIZE/DIR_ENTRY_SIZE)  /* # dir entries/blk   */
43065   #define SUPER_SIZE      usizeof (struct super_block)  /* super_block size    */
43066   #define PIPE_SIZE          (V1_NR_DZONES*BLOCK_SIZE)  /* pipe size in bytes  */
43067   #define BITMAP_CHUNKS (BLOCK_SIZE/usizeof (bitchunk_t))/* # map chunks/blk   */
43068
43069   /* Derived sizes pertaining to the V1 file system. */
43070   #define V1_ZONE_NUM_SIZE           usizeof (zone1_t)  /* # bytes in V1 zone  */
43071   #define V1_INODE_SIZE             usizeof (d1_inode)  /* bytes in V1 dsk ino */
43072   #define V1_INDIRECTS   (BLOCK_SIZE/V1_ZONE_NUM_SIZE)  /* # zones/indir block */
43073   #define V1_INODES_PER_BLOCK (BLOCK_SIZE/V1_INODE_SIZE)/* # V1 dsk inodes/blk */
43074
43075   /* Derived sizes pertaining to the V2 file system. */
43076   #define V2_ZONE_NUM_SIZE            usizeof (zone_t)  /* # bytes in V2 zone  */
43077   #define V2_INODE_SIZE             usizeof (d2_inode)  /* bytes in V2 dsk ino */
43078   #define V2_INDIRECTS   (BLOCK_SIZE/V2_ZONE_NUM_SIZE)  /* # zones/indir block */
43079   #define V2_INODES_PER_BLOCK (BLOCK_SIZE/V2_INODE_SIZE)/* # V2 dsk inodes/blk */
43080
43081   #define printf printk
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/dev.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

43100    /* Device table.  This table is indexed by major device number.  It provides
43101     * the link between major device numbers and the routines that process them.
43102     */
43103
43104    typedef _PROTOTYPE (void (*dmap_t), (int task, message *m_ptr) );
43105
43106    extern struct dmap {
43107      dmap_t dmap_open;
43108      dmap_t dmap_rw;
43109      dmap_t dmap_close;
43110      int dmap_task;
43111    } dmap[];
43112
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/file.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

43200    /* This is the filp table.  It is an intermediary between file descriptors and
43201     * inodes.  A slot is free if filp_count == 0.
43202     */
43203
43204    EXTERN struct filp {
43205      mode_t filp_mode;             /* RW bits, telling how file is opened */
43206      int filp_flags;               /* flags from open and fcntl */
43207      int filp_count;               /* how many file descriptors share this slot?*/
43208      struct inode *filp_ino;       /* pointer to the inode */
43209      off_t filp_pos;               /* file position */
43210    } filp[NR_FILPS];
43211
43212    #define FILP_CLOSED     0       /* filp_mode: associated device closed */
43213
43214    #define NIL_FILP (struct filp *) 0      /* indicates absence of a filp slot */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/fproc.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

43300    /* This is the per-process information.  A slot is reserved for each potential
43301     * process. Thus NR_PROCS must be the same as in the kernel. It is not possible
43302     * or even necessary to tell when a slot is free here.
43303     */
43304
43305
43306    EXTERN struct fproc {
43307      mode_t fp_umask;              /* mask set by umask system call */
43308      struct inode *fp_workdir;     /* pointer to working directory's inode */
43309      struct inode *fp_rootdir;     /* pointer to current root dir (see chroot) */
```

```
43310      struct filp *fp_filp[OPEN_MAX];/* the file descriptor table */
43311      uid_t fp_realuid;             /* real user id */
43312      uid_t fp_effuid;              /* effective user id */
43313      gid_t fp_realgid;             /* real group id */
43314      gid_t fp_effgid;              /* effective group id */
43315      dev_t fp_tty;                 /* major/minor of controlling tty */
43316      int fp_fd;                    /* place to save fd if rd/wr can't finish */
43317      char *fp_buffer;              /* place to save buffer if rd/wr can't finish*/
43318      int  fp_nbytes;               /* place to save bytes if rd/wr can't finish */
43319      int  fp_cum_io_partial;       /* partial byte count if rd/wr can't finish */
43320      char fp_suspended;            /* set to indicate process hanging */
43321      char fp_revived;              /* set to indicate process being revived */
43322      char fp_task;                 /* which task is proc suspended on */
43323      char fp_sesldr;               /* true if proc is a session leader */
43324      pid_t fp_pid;                 /* process id */
43325      long fp_cloexec;              /* bit map for POSIX Table 6-2 FD_CLOEXEC */
43326    } fproc[NR_PROCS];
43327
43328    /* Field values. */
43329    #define NOT_SUSPENDED      0    /* process is not suspended on pipe or task */
43330    #define SUSPENDED          1    /* process is suspended on pipe or task */
43331    #define NOT_REVIVING       0    /* process is not being revived */
43332    #define REVIVING           1    /* process is being revived from suspension */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/fs.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

43400    /* This is the master header for fs.  It includes some other files
43401     * and defines the principal constants.
43402     */
43403    #define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
43404    #define _MINIX             1    /* tell headers to include MINIX stuff */
43405    #define _SYSTEM            1    /* tell headers that this is the kernel */
43406
43407    /* The following are so basic, all the *.c files get them automatically. */
43408    #include <minix/config.h>       /* MUST be first */
43409    #include <ansi.h>               /* MUST be second */
43410    #include <sys/types.h>
43411    #include <minix/const.h>
43412    #include <minix/type.h>
43413
43414    #include <limits.h>
43415    #include <errno.h>
43416
43417    #include <minix/syslib.h>
43418
43419    #include "const.h"
43420    #include "type.h"
43421    #include "proto.h"
43422    #include "glo.h"
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/glo.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

43500    /* EXTERN should be extern except for the table file */
43501    #ifdef _TABLE
43502    #undef EXTERN
43503    #define EXTERN
43504    #endif
43505
43506    /* File System global variables */
43507    EXTERN struct fproc *fp;        /* pointer to caller's fproc struct */
43508    EXTERN int super_user;          /* 1 if caller is super_user, else 0 */
43509    EXTERN int dont_reply;          /* normally 0; set to 1 to inhibit reply */
43510    EXTERN int susp_count;          /* number of procs suspended on pipe */
43511    EXTERN int nr_locks;            /* number of locks currently in place */
43512    EXTERN int reviving;            /* number of pipe processes to be revived */
43513    EXTERN off_t rdahedpos;         /* position to read ahead */
43514    EXTERN struct inode *rdahed_inode;      /* pointer to inode to read ahead */
43515
43516    /* The parameters of the call are kept here. */
43517    EXTERN message m;               /* the input message itself */
43518    EXTERN message m1;              /* the output message used for reply */
43519    EXTERN int who;                 /* caller's proc number */
43520    EXTERN int fs_call;             /* system call number */
43521    EXTERN char user_path[PATH_MAX];/* storage for user path name */
43522
43523    /* The following variables are used for returning results to the caller. */
43524    EXTERN int err_code;            /* temporary storage for error number */
43525    EXTERN int rdwt_err;            /* status of last disk i/o request */
43526
43527    /* Data which need initialization. */
43528    extern _PROTOTYPE (int (*call_vector[]), (void) ); /* sys call table */
43529    extern int max_major;  /* maximum major device (+ 1) */
43530    extern char dot1[2];   /* dot1 (&dot1[0]) and dot2 (&dot2[0]) have a special */
43531    extern char dot2[3];   /* meaning to search_dir: no access permission check. */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/inode.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

43600    /* Inode table.  This table holds inodes that are currently in use.  In some
43601     * cases they have been opened by an open() or creat() system call, in other
43602     * cases the file system itself needs the inode for one reason or another,
43603     * such as to search a directory for a path name.
43604     * The first part of the struct holds fields that are present on the
43605     * disk; the second part holds fields not present on the disk.
43606     * The disk inode part is also declared in "type.h" as 'd1_inode' for V1
43607     * file systems and 'd2_inode' for V2 file systems.
43608     */
43609
43610    EXTERN struct inode {
43611      mode_t i_mode;                /* file type, protection, etc. */
43612      nlink_t i_nlinks;             /* how many links to this file */
43613      uid_t i_uid;                  /* user id of the file's owner */
43614      gid_t i_gid;                  /* group number */
```

```
43615       off_t i_size;                /* current file size in bytes */
43616       time_t i_atime;              /* time of last access (V2 only) */
43617       time_t i_mtime;              /* when was file data last changed */
43618       time_t i_ctime;              /* when was inode itself changed (V2 only)*/
43619       zone_t i_zone[V2_NR_TZONES];  /* zone numbers for direct, ind, and dbl ind */
43620
43621       /* The following items are not present on the disk. */
43622       dev_t i_dev;                 /* which device is the inode on */
43623       ino_t i_num;                 /* inode number on its (minor) device */
43624       int i_count;                 /* # times inode used; 0 means slot is free */
43625       int i_ndzones;               /* # direct zones (Vx_NR_DZONES) */
43626       int i_nindirs;               /* # indirect zones per indirect block */
43627       struct super_block *i_sp;    /* pointer to super block for inode's device */
43628       char i_dirt;                 /* CLEAN or DIRTY */
43629       char i_pipe;                 /* set to I_PIPE if pipe */
43630       char i_mount;                /* this bit is set if file mounted on */
43631       char i_seek;                 /* set on LSEEK, cleared on READ/WRITE */
43632       char i_update;               /* the ATIME, CTIME, and MTIME bits are here */
43633     } inode[NR_INODES];
43634
43635
43636     #define NIL_INODE (struct inode *) 0    /* indicates absence of inode slot */
43637
43638     /* Field values.  Note that CLEAN and DIRTY are defined in "const.h" */
43639     #define NO_PIPE            0    /* i_pipe is NO_PIPE if inode is not a pipe */
43640     #define I_PIPE             1    /* i_pipe is I_PIPE if inode is a pipe */
43641     #define NO_MOUNT           0    /* i_mount is NO_MOUNT if file not mounted on*/
43642     #define I_MOUNT            1    /* i_mount is I_MOUNT if file mounted on */
43643     #define NO_SEEK            0    /* i_seek = NO_SEEK if last op was not SEEK */
43644     #define ISEEK              1    /* i_seek = ISEEK if last op was SEEK */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/lock.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
43700     /* This is the file locking table.  Like the filp table, it points to the
43701      * inode table, however, in this case to achieve advisory locking.
43702      */
43703     EXTERN struct file_lock {
43704       short lock_type;             /* F_RDLOCK or F_WRLOCK; 0 means unused slot */
43705       pid_t lock_pid;              /* pid of the process holding the lock */
43706       struct inode *lock_inode;    /* pointer to the inode locked */
43707       off_t lock_first;            /* offset of first byte locked */
43708       off_t lock_last;             /* offset of last byte locked */
43709     } file_lock[NR_LOCKS];
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/param.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

43800    /* The following names are synonyms for the variables in the input message. */
43801    #define acc_time      m.m2_l1
43802    #define addr          m.m1_i3
43803    #define buffer        m.m1_p1
43804    #define child         m.m1_i2
43805    #define co_mode       m.m1_i1
43806    #define eff_grp_id    m.m1_i3
43807    #define eff_user_id   m.m1_i3
43808    #define erki          m.m1_p1
43809    #define fd            m.m1_i1
43810    #define fd2           m.m1_i2
43811    #define ioflags       m.m1_i3
43812    #define group         m.m1_i3
43813    #define real_grp_id   m.m1_i2
43814    #define ls_fd         m.m2_i1
43815    #define mk_mode       m.m1_i2
43816    #define mode          m.m3_i2
43817    #define c_mode        m.m1_i3
43818    #define c_name        m.m1_p1
43819    #define name          m.m3_p1
43820    #define name1         m.m1_p1
43821    #define name2         m.m1_p2
43822    #define name_length   m.m3_i1
43823    #define name1_length  m.m1_i1
43824    #define name2_length  m.m1_i2
43825    #define nbytes        m.m1_i2
43826    #define offset        m.m2_l1
43827    #define owner         m.m1_i2
43828    #define parent        m.m1_i1
43829    #define pathname      m.m3_ca1
43830    #define pid           m.m1_i3
43831    #define pro           m.m1_i1
43832    #define rd_only       m.m1_i3
43833    #define real_user_id  m.m1_i2
43834    #define request       m.m1_i2
43835    #define sig           m.m1_i2
43836    #define slot1         m.m1_i1
43837    #define tp            m.m2_l1
43838    #define utime_actime  m.m2_l1
43839    #define utime_modtime m.m2_l2
43840    #define utime_file    m.m2_p1
43841    #define utime_length  m.m2_i1
43842    #define whence        m.m2_i2
43843
43844    /* The following names are synonyms for the variables in the output message. */
43845    #define reply_type    m1.m_type
43846    #define reply_l1      m1.m2_l1
43847    #define reply_i1      m1.m1_i1
43848    #define reply_i2      m1.m1_i2
43849    #define reply_t1      m1.m4_l1
43850    #define reply_t2      m1.m4_l2
43851    #define reply_t3      m1.m4_l3
43852    #define reply_t4      m1.m4_l4
43853    #define reply_t5      m1.m4_l5
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/proto.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

43900    /* Function prototypes. */
43901
43902    /* Structs used in prototypes must be declared as such first. */
43903    struct buf;
43904    struct filp;
43905    struct inode;
43906    struct super_block;
43907
43908    /* cache.c */
43909    _PROTOTYPE( zone_t alloc_zone, (Dev_t dev, zone_t z)                    );
43910    _PROTOTYPE( void flushall, (Dev_t dev)                                  );
43911    _PROTOTYPE( void free_zone, (Dev_t dev, zone_t numb)                    );
43912    _PROTOTYPE( struct buf *get_block, (Dev_t dev, block_t block,int only_search));
43913    _PROTOTYPE( void invalidate, (Dev_t device)                             );
43914    _PROTOTYPE( void put_block, (struct buf *bp, int block_type)            );
43915    _PROTOTYPE( void rw_block, (struct buf *bp, int rw_flag)                );
43916    _PROTOTYPE( void rw_scattered, (Dev_t dev,
43917                            struct buf **bufq, int bufqsize, int rw_flag)   );
43918
43919    #if ENABLE_CACHE2
43920    /* cache2.c */
43921    _PROTOTYPE( void init_cache2, (unsigned long size)                      );
43922    _PROTOTYPE( int get_block2, (struct buf *bp, int only_search)           );
43923    _PROTOTYPE( void put_block2, (struct buf *bp)                           );
43924    _PROTOTYPE( void invalidate2, (Dev_t device)                            );
43925    #endif
43926
43927    /* device.c */
43928    _PROTOTYPE( void call_task, (int task_nr, message *mess_ptr)            );
43929    _PROTOTYPE( void dev_opcl, (int task_nr, message *mess_ptr)             );
43930    _PROTOTYPE( int dev_io, (int rw_flag, int nonblock, Dev_t dev,
43931                            off_t pos, int bytes, int proc, char *buff)     );
43932    _PROTOTYPE( int do_ioctl, (void)                                        );
43933    _PROTOTYPE( void no_dev, (int task_nr, message *m_ptr)                  );
43934    _PROTOTYPE( void call_ctty, (int task_nr, message *mess_ptr)            );
43935    _PROTOTYPE( void tty_open, (int task_nr, message *mess_ptr)             );
43936    _PROTOTYPE( void ctty_close, (int task_nr, message *mess_ptr)           );
43937    _PROTOTYPE( void ctty_open, (int task_nr, message *mess_ptr)            );
43938    _PROTOTYPE( int do_setsid, (void)                                       );
43939    #if ENABLE_NETWORKING
43940    _PROTOTYPE( void net_open, (int task_nr, message *mess_ptr)             );
43941    #else
43942    #define net_open  0
43943    #endif
43944
43945    /* filedes.c */
43946    _PROTOTYPE( struct filp *find_filp, (struct inode *rip, Mode_t bits)    );
43947    _PROTOTYPE( int get_fd, (int start, Mode_t bits, int *k, struct filp **fpt) );
43948    _PROTOTYPE( struct filp *get_filp, (int fild)                           );
43949
43950    /* inode.c */
43951    _PROTOTYPE( struct inode *alloc_inode, (Dev_t dev, Mode_t bits)         );
43952    _PROTOTYPE( void dup_inode, (struct inode *ip)                          );
43953    _PROTOTYPE( void free_inode, (Dev_t dev, Ino_t numb)                    );
43954    _PROTOTYPE( struct inode *get_inode, (Dev_t dev, int numb)              );
```

```
43955   _PROTOTYPE( void put_inode, (struct inode *rip)                         );
43956   _PROTOTYPE( void update_times, (struct inode *rip)                      );
43957   _PROTOTYPE( void rw_inode, (struct inode *rip, int rw_flag)             );
43958   _PROTOTYPE( void wipe_inode, (struct inode *rip)                        );
43959
43960   /* link.c */
43961   _PROTOTYPE( int do_link, (void)                                         );
43962   _PROTOTYPE( int do_unlink, (void)                                       );
43963   _PROTOTYPE( int do_rename, (void)                                       );
43964   _PROTOTYPE( void truncate, (struct inode *rip)                          );
43965
43966   /* lock.c */
43967   _PROTOTYPE( int lock_op, (struct filp *f, int req)                      );
43968   _PROTOTYPE( void lock_revive, (void)                                    );
43969
43970   /* main.c */
43971   _PROTOTYPE( void main, (void)                                           );
43972   _PROTOTYPE( void reply, (int whom, int result)                          );
43973
43974   /* misc.c */
43975   _PROTOTYPE( int do_dup, (void)                                          );
43976   _PROTOTYPE( int do_exit, (void)                                         );
43977   _PROTOTYPE( int do_fcntl, (void)                                        );
43978   _PROTOTYPE( int do_fork, (void)                                         );
43979   _PROTOTYPE( int do_exec, (void)                                         );
43980   _PROTOTYPE( int do_revive, (void)                                       );
43981   _PROTOTYPE( int do_set, (void)                                          );
43982   _PROTOTYPE( int do_sync, (void)                                         );
43983
43984   /* mount.c */
43985   _PROTOTYPE( int do_mount, (void)                                        );
43986   _PROTOTYPE( int do_umount, (void)                                       );
43987
43988   /* open.c */
43989   _PROTOTYPE( int do_close, (void)                                        );
43990   _PROTOTYPE( int do_creat, (void)                                        );
43991   _PROTOTYPE( int do_lseek, (void)                                        );
43992   _PROTOTYPE( int do_mknod, (void)                                        );
43993   _PROTOTYPE( int do_mkdir, (void)                                        );
43994   _PROTOTYPE( int do_open, (void)                                         );
43995
43996   /* path.c */
43997   _PROTOTYPE( struct inode *advance,(struct inode *dirp, char string[NAME_MAX]));
43998   _PROTOTYPE( int search_dir, (struct inode *ldir_ptr,
43999                           char string [NAME_MAX], ino_t *numb, int flag)  );
44000   _PROTOTYPE( struct inode *eat_path, (char *path)                        );
44001   _PROTOTYPE( struct inode *last_dir, (char *path, char string [NAME_MAX]));
44002
44003   /* pipe.c */
44004   _PROTOTYPE( int do_pipe, (void)                                         );
44005   _PROTOTYPE( int do_unpause, (void)                                      );
44006   _PROTOTYPE( int pipe_check, (struct inode *rip, int rw_flag,
44007                           int oflags, int bytes, off_t position, int *canwrite));
44008   _PROTOTYPE( void release, (struct inode *ip, int call_nr, int count)    );
44009   _PROTOTYPE( void revive, (int proc_nr, int bytes)                       );
44010   _PROTOTYPE( void suspend, (int task)                                    );
44011
44012   /* protect.c */
44013   _PROTOTYPE( int do_access, (void)                                       );
44014   _PROTOTYPE( int do_chmod, (void)                                        );
```

```
44015   _PROTOTYPE( int do_chown, (void)                                        );
44016   _PROTOTYPE( int do_umask, (void)                                        );
44017   _PROTOTYPE( int forbidden, (struct inode *rip, Mode_t access_desired)   );
44018   _PROTOTYPE( int read_only, (struct inode *ip)                           );
44019
44020   /* putk.c */
44021   _PROTOTYPE( void putk, (int c)                                          );
44022
44023   /* read.c */
44024   _PROTOTYPE( int do_read, (void)                                         );
44025   _PROTOTYPE( struct buf *rahead, (struct inode *rip, block_t baseblock,
44026                           off_t position, unsigned bytes_ahead)           );
44027   _PROTOTYPE( void read_ahead, (void)                                     );
44028   _PROTOTYPE( block_t read_map, (struct inode *rip, off_t position)       );
44029   _PROTOTYPE( int read_write, (int rw_flag)                               );
44030   _PROTOTYPE( zone_t rd_indir, (struct buf *bp, int index)                );
44031
44032   /* stadir.c */
44033   _PROTOTYPE( int do_chdir, (void)                                        );
44034   _PROTOTYPE( int do_chroot, (void)                                       );
44035   _PROTOTYPE( int do_fstat, (void)                                        );
44036   _PROTOTYPE( int do_stat, (void)                                         );
44037
44038   /* super.c */
44039   _PROTOTYPE( bit_t alloc_bit, (struct super_block *sp, int map, bit_t origin));
44040   _PROTOTYPE( void free_bit, (struct super_block *sp, int map,
44041                                           bit_t bit_returned)     );
44042   _PROTOTYPE( struct super_block *get_super, (Dev_t dev)                  );
44043   _PROTOTYPE( int mounted, (struct inode *rip)                            );
44044   _PROTOTYPE( int read_super, (struct super_block *sp)                    );
44045
44046   /* time.c */
44047   _PROTOTYPE( int do_stime, (void)                                        );
44048   _PROTOTYPE( int do_time, (void)                                         );
44049   _PROTOTYPE( int do_tims, (void)                                         );
44050   _PROTOTYPE( int do_utime, (void)                                        );
44051
44052   /* utility.c */
44053   _PROTOTYPE( time_t clock_time, (void)                                   );
44054   _PROTOTYPE( unsigned conv2, (int norm, int w)                           );
44055   _PROTOTYPE( long conv4, (int norm, long x)                              );
44056   _PROTOTYPE( int fetch_name, (char *path, int len, int flag)             );
44057   _PROTOTYPE( int no_sys, (void)                                          );
44058   _PROTOTYPE( void panic, (char *format, int num)                         );
44059
44060   /* write.c */
44061   _PROTOTYPE( void clear_zone, (struct inode *rip, off_t pos, int flag)   );
44062   _PROTOTYPE( int do_write, (void)                                        );
44063   _PROTOTYPE( struct buf *new_block, (struct inode *rip, off_t position)  );
44064   _PROTOTYPE( void zero_block, (struct buf *bp)                           );
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/super.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

44100    /* Super block table.  The root file system and every mounted file system
44101     * has an entry here.  The entry holds information about the sizes of the bit
44102     * maps and inodes.  The s_ninodes field gives the number of inodes available
44103     * for files and directories, including the root directory.  Inode 0 is
44104     * on the disk, but not used.  Thus s_ninodes = 4 means that 5 bits will be
44105     * used in the bit map, bit 0, which is always 1 and not used, and bits 1-4
44106     * for files and directories.  The disk layout is:
44107     *
44108     *      Item        # blocks
44109     *    boot block      1
44110     *    super block     1
44111     *    inode map     s_imap_blocks
44112     *    zone map      s_zmap_blocks
44113     *    inodes        (s_ninodes + 'inodes per block' - 1)/'inodes per block'
44114     *    unused        whatever is needed to fill out the current zone
44115     *    data zones    (s_zones - s_firstdatazone) << s_log_zone_size
44116     *
44117     * A super_block slot is free if s_dev == NO_DEV.
44118     */
44119
44120
44121    EXTERN struct super_block {
44122      ino_t s_ninodes;              /* # usable inodes on the minor device */
44123      zone1_t  s_nzones;            /* total device size, including bit maps etc */
44124      short s_imap_blocks;          /* # of blocks used by inode bit map */
44125      short s_zmap_blocks;          /* # of blocks used by zone bit map */
44126      zone1_t s_firstdatazone;      /* number of first data zone */
44127      short s_log_zone_size;        /* log2 of blocks/zone */
44128      off_t s_max_size;             /* maximum file size on this device */
44129      short s_magic;                /* magic number to recognize super-blocks */
44130      short s_pad;                  /* try to avoid compiler-dependent padding */
44131      zone_t s_zones;               /* number of zones (replaces s_nzones in V2) */
44132
44133      /* The following items are only used when the super_block is in memory. */
44134      struct inode *s_isup;         /* inode for root dir of mounted file sys */
44135      struct inode *s_imount;       /* inode mounted on */
44136      unsigned s_inodes_per_block;  /* precalculated from magic number */
44137      dev_t s_dev;                  /* whose super block is this? */
44138      int s_rd_only;                /* set to 1 iff file sys mounted read only */
44139      int s_native;                 /* set to 1 iff not byte swapped file system */
44140      int s_version;                /* file system version, zero means bad magic */
44141      int s_ndzones;                /* # direct zones in an inode */
44142      int s_nindirs;                /* # indirect zones per indirect block */
44143      bit_t s_isearch;              /* inodes below this bit number are in use */
44144      bit_t s_zsearch;              /* all zones below this bit number are in use*/
44145    } super_block[NR_SUPERS];
44146
44147    #define NIL_SUPER (struct super_block *) 0
44148    #define IMAP            0       /* operating on the inode bit map */
44149    #define ZMAP            1       /* operating on the zone bit map */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/type.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

44200    /* Declaration of the V1 inode as it is on the disk (not in core). */
44201    typedef struct {                /* V1.x disk inode */
44202      mode_t d1_mode;               /* file type, protection, etc. */
44203      uid_t d1_uid;                 /* user id of the file's owner */
44204      off_t d1_size;                /* current file size in bytes */
44205      time_t d1_mtime;              /* when was file data last changed */
44206      gid_t d1_gid;                 /* group number */
44207      nlink_t d1_nlinks;            /* how many links to this file */
44208      u16_t d1_zone[V1_NR_TZONES];  /* block nums for direct, ind, and dbl ind */
44209    } d1_inode;
44210
44211    /* Declaration of the V2 inode as it is on the disk (not in core). */
44212    typedef struct {                /* V2.x disk inode */
44213      mode_t d2_mode;               /* file type, protection, etc. */
44214      u16_t d2_nlinks;              /* how many links to this file. HACK! */
44215      uid_t d2_uid;                 /* user id of the file's owner. */
44216      u16_t d2_gid;                 /* group number HACK! */
44217      off_t d2_size;                /* current file size in bytes */
44218      time_t d2_atime;              /* when was file data last accessed */
44219      time_t d2_mtime;              /* when was file data last changed */
44220      time_t d2_ctime;              /* when was inode data last changed */
44221      zone_t d2_zone[V2_NR_TZONES]; /* block nums for direct, ind, and dbl ind */
44222    } d2_inode;



++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/cache.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

44300    /* The file system maintains a buffer cache to reduce the number of disk
44301     * accesses needed.  Whenever a read or write to the disk is done, a check is
44302     * first made to see if the block is in the cache.  This file manages the
44303     * cache.
44304     *
44305     * The entry points into this file are:
44306     *   get_block:   request to fetch a block for reading or writing from cache
44307     *   put_block:   return a block previously requested with get_block
44308     *   alloc_zone:  allocate a new zone (to increase the length of a file)
44309     *   free_zone:   release a zone (when a file is removed)
44310     *   rw_block:    read or write a block from the disk itself
44311     *   invalidate:  remove all the cache blocks on some device
44312     */
44313
44314    #include "fs.h"
44315    #include <minix/com.h>
44316    #include <minix/boot.h>
44317    #include "buf.h"
44318    #include "file.h"
44319    #include "fproc.h"
44320    #include "super.h"
44321
44322    FORWARD _PROTOTYPE( void rm_lru, (struct buf *bp) );
44323
44324    /*===========================================================================*
```

```
44325     *                             get_block                                        *
44326     *==============================================================================*/
44327    PUBLIC struct buf *get_block(dev, block, only_search)
44328    register dev_t dev;             /* on which device is the block? */
44329    register block_t block;         /* which block is wanted? */
44330    int only_search;                /* if NO_READ, don't read, else act normal */
44331    {
44332    /* Check to see if the requested block is in the block cache.  If so, return
44333     * a pointer to it.  If not, evict some other block and fetch it (unless
44334     * 'only_search' is 1).  All the blocks in the cache that are not in use
44335     * are linked together in a chain, with 'front' pointing to the least recently
44336     * used block and 'rear' to the most recently used block.  If 'only_search' is
44337     * 1, the block being requested will be overwritten in its entirety, so it is
44338     * only necessary to see if it is in the cache; if it is not, any free buffer
44339     * will do.  It is not necessary to actually read the block in from disk.
44340     * If 'only_search' is PREFETCH, the block need not be read from the disk,
44341     * and the device is not to be marked on the block, so callers can tell if
44342     * the block returned is valid.
44343     * In addition to the LRU chain, there is also a hash chain to link together
44344     * blocks whose block numbers end with the same bit strings, for fast lookup.
44345     */
44346
44347      int b;
44348      register struct buf *bp, *prev_ptr;
44349
44350      /* Search the hash chain for (dev, block). Do_read() can use
44351       * get_block(NO_DEV ...) to get an unnamed block to fill with zeros when
44352       * someone wants to read from a hole in a file, in which case this search
44353       * is skipped
44354       */
44355      if (dev != NO_DEV) {
44356            b = (int) block & HASH_MASK;
44357            bp = buf_hash[b];
44358            while (bp != NIL_BUF) {
44359                    if (bp->b_blocknr == block && bp->b_dev == dev) {
44360                            /* Block needed has been found. */
44361                            if (bp->b_count == 0) rm_lru(bp);
44362                            bp->b_count++;  /* record that block is in use */
44363                            return(bp);
44364                    } else {
44365                            /* This block is not the one sought. */
44366                            bp = bp->b_hash; /* move to next block on hash chain */
44367                    }
44368            }
44369      }
44370
44371      /* Desired block is not on available chain.  Take oldest block ('front'). */
44372      if ((bp = front) == NIL_BUF) panic("all buffers in use", NR_BUFS);
44373      rm_lru(bp);
44374
44375      /* Remove the block that was just taken from its hash chain. */
44376      b = (int) bp->b_blocknr & HASH_MASK;
44377      prev_ptr = buf_hash[b];
44378      if (prev_ptr == bp) {
44379            buf_hash[b] = bp->b_hash;
44380      } else {
44381            /* The block just taken is not on the front of its hash chain. */
44382            while (prev_ptr->b_hash != NIL_BUF)
44383                    if (prev_ptr->b_hash == bp) {
44384                            prev_ptr->b_hash = bp->b_hash;  /* found it */
```

```
44385                           break;
44386                   } else {
44387                           prev_ptr = prev_ptr->b_hash;    /* keep looking */
44388                   }
44389     }
44390
44391     /* If the block taken is dirty, make it clean by writing it to the disk.
44392      * Avoid hysteresis by flushing all other dirty blocks for the same device.
44393      */
44394     if (bp->b_dev != NO_DEV) {
44395           if (bp->b_dirt == DIRTY) flushall(bp->b_dev);
44396   #if ENABLE_CACHE2
44397           put_block2(bp);
44398   #endif
44399     }
44400
44401     /* Fill in block's parameters and add it to the hash chain where it goes. */
44402     bp->b_dev = dev;              /* fill in device number */
44403     bp->b_blocknr = block;        /* fill in block number */
44404     bp->b_count++;                /* record that block is being used */
44405     b = (int) bp->b_blocknr & HASH_MASK;
44406     bp->b_hash = buf_hash[b];
44407     buf_hash[b] = bp;             /* add to hash list */
44408
44409     /* Go get the requested block unless searching or prefetching. */
44410     if (dev != NO_DEV) {
44411   #if ENABLE_CACHE2
44412           if (get_block2(bp, only_search)) /* in 2nd level cache */;
44413           else
44414   #endif
44415           if (only_search == PREFETCH) bp->b_dev = NO_DEV;
44416           else
44417           if (only_search == NORMAL) rw_block(bp, READING);
44418     }
44419     return(bp);                   /* return the newly acquired block */
44420   }


44423   /*===========================================================================*
44424    *                               put_block                                   *
44425    *===========================================================================*/
44426   PUBLIC void put_block(bp, block_type)
44427   register struct buf *bp;        /* pointer to the buffer to be released */
44428   int block_type;                 /* INODE_BLOCK, DIRECTORY_BLOCK, or whatever */
44429   {
44430   /* Return a block to the list of available blocks.   Depending on 'block_type'
44431    * it may be put on the front or rear of the LRU chain.  Blocks that are
44432    * expected to be needed again shortly (e.g., partially full data blocks)
44433    * go on the rear; blocks that are unlikely to be needed again shortly
44434    * (e.g., full data blocks) go on the front.  Blocks whose loss can hurt
44435    * the integrity of the file system (e.g., inode blocks) are written to
44436    * disk immediately if they are dirty.
44437    */
44438
44439     if (bp == NIL_BUF) return;    /* it is easier to check here than in caller */
44440
44441     bp->b_count--;                /* there is one use fewer now */
44442     if (bp->b_count != 0) return; /* block is still in use */
44443
44444     bufs_in_use--;                /* one fewer block buffers in use */
```

```
44445
44446       /* Put this block back on the LRU chain.  If the ONE_SHOT bit is set in
44447        * 'block_type', the block is not likely to be needed again shortly, so put
44448        * it on the front of the LRU chain where it will be the first one to be
44449        * taken when a free buffer is needed later.
44450        */
44451       if (block_type & ONE_SHOT) {
44452             /* Block probably won't be needed quickly. Put it on front of chain.
44453              * It will be the next block to be evicted from the cache.
44454              */
44455             bp->b_prev = NIL_BUF;
44456             bp->b_next = front;
44457             if (front == NIL_BUF)
44458                     rear = bp;      /* LRU chain was empty */
44459             else
44460                     front->b_prev = bp;
44461             front = bp;
44462       } else {
44463             /* Block probably will be needed quickly.  Put it on rear of chain.
44464              * It will not be evicted from the cache for a long time.
44465              */
44466             bp->b_prev = rear;
44467             bp->b_next = NIL_BUF;
44468             if (rear == NIL_BUF)
44469                     front = bp;
44470             else
44471                     rear->b_next = bp;
44472             rear = bp;
44473       }
44474
44475       /* Some blocks are so important (e.g., inodes, indirect blocks) that they
44476        * should be written to the disk immediately to avoid messing up the file
44477        * system in the event of a crash.
44478        */
44479       if ((block_type & WRITE_IMMED) && bp->b_dirt==DIRTY && bp->b_dev != NO_DEV)
44480             rw_block(bp, WRITING);
44481   }


44484   /*===========================================================================*
44485    *                              alloc_zone                                   *
44486    *===========================================================================*/
44487   PUBLIC zone_t alloc_zone(dev, z)
44488   dev_t dev;                    /* device where zone wanted */
44489   zone_t z;                     /* try to allocate new zone near this one */
44490   {
44491   /* Allocate a new zone on the indicated device and return its number. */
44492
44493     int major, minor;
44494     bit_t b, bit;
44495     struct super_block *sp;
44496
44497     /* Note that the routine alloc_bit() returns 1 for the lowest possible
44498      * zone, which corresponds to sp->s_firstdatazone.  To convert a value
44499      * between the bit number, 'b', used by alloc_bit() and the zone number, 'z',
44500      * stored in the inode, use the formula:
44501      *     z = b + sp->s_firstdatazone - 1
44502      * Alloc_bit() never returns 0, since this is used for NO_BIT (failure).
44503      */
44504     sp = get_super(dev);          /* find the super_block for this device */
```

```
44505
44506     /* If z is 0, skip initial part of the map known to be fully in use. */
44507     if (z == sp->s_firstdatazone) {
44508           bit = sp->s_zsearch;
44509     } else {
44510           bit = (bit_t) z - (sp->s_firstdatazone - 1);
44511     }
44512     b = alloc_bit(sp, ZMAP, bit);
44513     if (b == NO_BIT) {
44514           err_code = ENOSPC;
44515           major = (int) (sp->s_dev >> MAJOR) & BYTE;
44516           minor = (int) (sp->s_dev >> MINOR) & BYTE;
44517           printf("No space on %sdevice %d/%d\n",
44518                   sp->s_dev == ROOT_DEV ? "root " : "", major, minor);
44519           return(NO_ZONE);
44520     }
44521     if (z == sp->s_firstdatazone) sp->s_zsearch = b;      /* for next time */
44522     return(sp->s_firstdatazone - 1 + (zone_t) b);
44523   }


44526   /*===========================================================================*
44527    *                              free_zone                                    *
44528    *===========================================================================*/
44529   PUBLIC void free_zone(dev, numb)
44530   dev_t dev;                            /* device where zone located */
44531   zone_t numb;                          /* zone to be returned */
44532   {
44533   /* Return a zone. */
44534
44535     register struct super_block *sp;
44536     bit_t bit;
44537
44538     /* Locate the appropriate super_block and return bit. */
44539     sp = get_super(dev);
44540     if (numb < sp->s_firstdatazone || numb >= sp->s_zones) return;
44541     bit = (bit_t) (numb - (sp->s_firstdatazone - 1));
44542     free_bit(sp, ZMAP, bit);
44543     if (bit < sp->s_zsearch) sp->s_zsearch = bit;
44544   }


44547   /*===========================================================================*
44548    *                              rw_block                                     *
44549    *===========================================================================*/
44550   PUBLIC void rw_block(bp, rw_flag)
44551   register struct buf *bp;        /* buffer pointer */
44552   int rw_flag;                    /* READING or WRITING */
44553   {
44554   /* Read or write a disk block. This is the only routine in which actual disk
44555    * I/O is invoked. If an error occurs, a message is printed here, but the error
44556    * is not reported to the caller.  If the error occurred while purging a block
44557    * from the cache, it is not clear what the caller could do about it anyway.
44558    */
44559
44560     int r, op;
44561     off_t pos;
44562     dev_t dev;
44563
44564     if ( (dev = bp->b_dev) != NO_DEV) {
```

```
44565           pos = (off_t) bp->b_blocknr * BLOCK_SIZE;
44566           op = (rw_flag == READING ? DEV_READ : DEV_WRITE);
44567           r = dev_io(op, FALSE, dev, pos, BLOCK_SIZE, FS_PROC_NR, bp->b_data);
44568           if (r != BLOCK_SIZE) {
44569               if (r >= 0) r = END_OF_FILE;
44570               if (r != END_OF_FILE)
44571                 printf("Unrecoverable disk error on device %d/%d, block %ld\n",
44572                           (dev>>MAJOR)&BYTE, (dev>>MINOR)&BYTE, bp->b_blocknr);
44573                 bp->b_dev = NO_DEV;     /* invalidate block */
44574
44575                 /* Report read errors to interested parties. */
44576                 if (rw_flag == READING) rdwt_err = r;
44577           }
44578     }
44579
44580     bp->b_dirt = CLEAN;
44581   }


44584   /*===========================================================================*
44585    *                              invalidate                                   *
44586    *===========================================================================*/
44587   PUBLIC void invalidate(device)
44588   dev_t device;                   /* device whose blocks are to be purged */
44589   {
44590   /* Remove all the blocks belonging to some device from the cache. */
44591
44592     register struct buf *bp;
44593
44594     for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++)
44595           if (bp->b_dev == device) bp->b_dev = NO_DEV;
44596
44597   #if ENABLE_CACHE2
44598     invalidate2(device);
44599   #endif
44600   }


44603   /*===========================================================================*
44604    *                              flushall                                     *
44605    *===========================================================================*/
44606   PUBLIC void flushall(dev)
44607   dev_t dev;                      /* device to flush */
44608   {
44609   /* Flush all dirty blocks for one device. */
44610
44611     register struct buf *bp;
44612     static struct buf *dirty[NR_BUFS];    /* static so it isn't on stack */
44613     int ndirty;
44614
44615     for (bp = &buf[0], ndirty = 0; bp < &buf[NR_BUFS]; bp++)
44616           if (bp->b_dirt == DIRTY && bp->b_dev == dev) dirty[ndirty++] = bp;
44617     rw_scattered(dev, dirty, ndirty, WRITING);
44618   }


44621   /*===========================================================================*
44622    *                              rw_scattered                                 *
44623    *===========================================================================*/
44624   PUBLIC void rw_scattered(dev, bufq, bufqsize, rw_flag)
```

```
44625   dev_t dev;                      /* major-minor device number */
44626   struct buf **bufq;              /* pointer to array of buffers */
44627   int bufqsize;                   /* number of buffers */
44628   int rw_flag;                    /* READING or WRITING */
44629   {
44630   /* Read or write scattered data from a device. */
44631
44632     register struct buf *bp;
44633     int gap;
44634     register int i;
44635     register struct iorequest_s *iop;
44636     static struct iorequest_s iovec[NR_IOREQS];  /* static so it isn't on stack */
44637     int j;
44638
44639     /* (Shell) sort buffers on b_blocknr. */
44640     gap = 1;
44641     do
44642           gap = 3 * gap + 1;
44643     while (gap <= bufqsize);
44644     while (gap != 1) {
44645           gap /= 3;
44646           for (j = gap; j < bufqsize; j++) {
44647                   for (i = j - gap;
44648                        i >= 0 && bufq[i]->b_blocknr > bufq[i + gap]->b_blocknr;
44649                        i -= gap) {
44650                           bp = bufq[i];
44651                           bufq[i] = bufq[i + gap];
44652                           bufq[i + gap] = bp;
44653                   }
44654           }
44655     }
44656
44657     /* Set up i/o vector and do i/o.  The result of dev_io is discarded because
44658      * all results are returned in the vector.  If dev_io fails completely, the
44659      * vector is unchanged and all results are taken as errors.
44660      */
44661     while (bufqsize > 0) {
44662           for (j = 0, iop = iovec; j < NR_IOREQS && j < bufqsize; j++, iop++) {
44663                   bp = bufq[j];
44664                   iop->io_position = (off_t) bp->b_blocknr * BLOCK_SIZE;
44665                   iop->io_buf = bp->b_data;
44666                   iop->io_nbytes = BLOCK_SIZE;
44667                   iop->io_request = rw_flag == WRITING ?
44668                                     DEV_WRITE : DEV_READ | OPTIONAL_IO;
44669           }
44670           (void) dev_io(SCATTERED_IO, 0, dev, (off_t) 0, j, FS_PROC_NR,
44671                                                           (char *) iovec);
44672
44673           /* Harvest the results.  Leave read errors for rw_block() to complain. */
44674           for (i = 0, iop = iovec; i < j; i++, iop++) {
44675                   bp = bufq[i];
44676                   if (rw_flag == READING) {
44677                       if (iop->io_nbytes == 0)
44678                           bp->b_dev = dev;        /* validate block */
44679                       put_block(bp, PARTIAL_DATA_BLOCK);
44680                   } else {
44681                       if (iop->io_nbytes != 0) {
44682                        printf("Unrecoverable write error on device %d/%d, block %ld\n",
44683                                   (dev>>MAJOR)&BYTE, (dev>>MINOR)&BYTE, bp->b_blocknr);
44684                           bp->b_dev = NO_DEV;     /* invalidate block */
```

```
44685                       }
44686                       bp->b_dirt = CLEAN;
44687                   }
44688           }
44689           bufq += j;
44690           bufqsize -= j;
44691     }
44692   }


44695   /*===========================================================================*
44696    *                               rm_lru                                      *
44697    *===========================================================================*/
44698   PRIVATE void rm_lru(bp)
44699   struct buf *bp;
44700   {
44701   /* Remove a block from its LRU chain. */
44702
44703     struct buf *next_ptr, *prev_ptr;
44704
44705     bufs_in_use++;
44706     next_ptr = bp->b_next;        /* successor on LRU chain */
44707     prev_ptr = bp->b_prev;        /* predecessor on LRU chain */
44708     if (prev_ptr != NIL_BUF)
44709           prev_ptr->b_next = next_ptr;
44710     else
44711           front = next_ptr;       /* this block was at front of chain */
44712
44713     if (next_ptr != NIL_BUF)
44714           next_ptr->b_prev = prev_ptr;
44715     else
44716           rear = prev_ptr;        /* this block was at rear of chain */
44717   }
```

++
src/fs/cache2.c
++

```
44800   /* Second level block cache to supplement the file system cache.  The block
44801    * cache of a 16-bit Minix system is very small, too small to prevent trashing.
44802    * A generic 32-bit system also doesn't have a very large cache to allow it
44803    * to run on systems with little memory.  On a system with lots of memory one
44804    * can use the RAM disk as a read-only second level cache.  Any blocks pushed
44805    * out of the primary cache are cached on the RAM disk.  This code manages the
44806    * second level cache.  The cache is a simple FIFO where old blocks are put
44807    * into and drop out at the other end.  Must be searched backwards.
44808    *
44809    * The entry points into this file are:
44810    *   init_cache2: initialize the second level cache
44811    *   get_block2:  get a block from the 2nd level cache
44812    *   put_block2:  store a block in the 2nd level cache
44813    *   invalidate2: remove all the cache blocks on some device
44814    */
44815
44816   #include "fs.h"
44817   #include <minix/com.h>
44818   #include <minix/boot.h>
44819   #include "buf.h"
```

```
44820
44821   #if ENABLE_CACHE2
44822
44823   #define MAX_BUF2        (256 * sizeof(char *))
44824
44825   PRIVATE struct buf2 {   /* 2nd level cache per block administration */
44826     block_t b2_blocknr;           /* block number */
44827     dev_t b2_dev;                 /* device number */
44828     u16_t b2_count;               /* count of in-cache block groups */
44829   } buf2[MAX_BUF2];
44830
44831   PRIVATE unsigned nr_buf2;               /* actual cache size */
44832   PRIVATE unsigned buf2_idx;              /* round-robin reuse index */
44833
44834   #define hash2(block)    ((unsigned) ((block) & (MAX_BUF2 - 1)))
44835
44836
44837   /*===========================================================================*
44838    *                               init_cache2                                 *
44839    *===========================================================================*/
44840   PUBLIC void init_cache2(size)
44841   unsigned long size;
44842   {
44843   /* Initialize the second level disk buffer cache of 'size' blocks. */
44844
44845     nr_buf2 = size > MAX_BUF2 ? MAX_BUF2 : (unsigned) size;
44846   }


44849   /*===========================================================================*
44850    *                               get_block2                                  *
44851    *===========================================================================*/
44852   PUBLIC int get_block2(bp, only_search)
44853   struct buf *bp;                 /* buffer to get from the 2nd level cache */
44854   int only_search;                /* if NO_READ, do nothing, else act normal */
44855   {
44856   /* Fill a buffer from the 2nd level cache.  Return true iff block acquired. */
44857     unsigned b;
44858     struct buf2 *bp2;
44859
44860     /* If the block wanted is in the RAM disk then our game is over. */
44861     if (bp->b_dev == DEV_RAM) nr_buf2 = 0;
44862
44863     /* Cache enabled?  NO_READ?  Any blocks with the same hash key? */
44864     if (nr_buf2 == 0 || only_search == NO_READ
44865                         || buf2[hash2(bp->b_blocknr)].b2_count == 0) return(0);
44866
44867     /* Search backwards (there may be older versions). */
44868     b = buf2_idx;
44869     for (;;) {
44870           if (b == 0) b = nr_buf2;
44871           bp2 = &buf2[--b];
44872           if (bp2->b2_blocknr == bp->b_blocknr && bp2->b2_dev == bp->b_dev) break;
44873           if (b == buf2_idx) return(0);
44874     }
44875
44876     /* Block is in the cache, get it. */
44877     if (dev_io(DEV_READ, 0, DEV_RAM, (off_t) b * BLOCK_SIZE, BLOCK_SIZE,
44878                                       FS_PROC_NR, bp->b_data) == BLOCK_SIZE) {
44879           return(1);
```

```
44880       }
44881       return(0);
44882     }


44885     /*===========================================================================*
44886      *                                put_block2                                 *
44887      *===========================================================================*/
44888     PUBLIC void put_block2(bp)
44889     struct buf *bp;                 /* buffer to store in the 2nd level cache */
44890     {
44891     /* Store a buffer into the 2nd level cache. */
44892       unsigned b;
44893       struct buf2 *bp2;
44894
44895       if (nr_buf2 == 0) return;     /* no 2nd level cache */
44896
44897       b = buf2_idx++;
44898       if (buf2_idx == nr_buf2) buf2_idx = 0;
44899
44900       bp2 = &buf2[b];
44901
44902       if (dev_io(DEV_WRITE, 0, DEV_RAM, (off_t) b * BLOCK_SIZE, BLOCK_SIZE,
44903                                           FS_PROC_NR, bp->b_data) == BLOCK_SIZE) {
44904             if (bp2->b2_dev != NO_DEV) buf2[hash2(bp2->b2_blocknr)].b2_count--;
44905             bp2->b2_dev = bp->b_dev;
44906             bp2->b2_blocknr = bp->b_blocknr;
44907             buf2[hash2(bp2->b2_blocknr)].b2_count++;
44908       }
44909     }


44912     /*===========================================================================*
44913      *                                invalidate2                                *
44914      *===========================================================================*/
44915     PUBLIC void invalidate2(device)
44916     dev_t device;
44917     {
44918     /* Invalidate all blocks from a given device in the 2nd level cache. */
44919       unsigned b;
44920       struct buf2 *bp2;
44921
44922       for (b = 0; b < nr_buf2; b++) {
44923             bp2 = &buf2[b];
44924             if (bp2->b2_dev == device) {
44925                     bp2->b2_dev = NO_DEV;
44926                     buf2[hash2(bp2->b2_blocknr)].b2_count--;
44927             }
44928       }
44929     }
44930     #endif /* ENABLE_CACHE2 */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/device.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

45000    /* When a needed block is not in the cache, it must be fetched from the disk.
45001     * Special character files also require I/O.  The routines for these are here.
45002     *
45003     * The entry points in this file are:
45004     *   dev_io:     perform a read or write on a block or character device
45005     *   dev_opcl:   perform generic device-specific processing for open & close
45006     *   tty_open:   perform tty-specific processing for open
45007     *   ctty_open:  perform controlling-tty-specific processing for open
45008     *   ctty_close: perform controlling-tty-specific processing for close
45009     *   do_setsid:  perform the SETSID system call (FS side)
45010     *   do_ioctl:   perform the IOCTL system call
45011     *   call_task:  procedure that actually calls the kernel tasks
45012     *   call_ctty:  procedure that actually calls task for /dev/tty
45013     */
45014
45015    #include "fs.h"
45016    #include <fcntl.h>
45017    #include <minix/callnr.h>
45018    #include <minix/com.h>
45019    #include "dev.h"
45020    #include "file.h"
45021    #include "fproc.h"
45022    #include "inode.h"
45023    #include "param.h"
45024
45025    PRIVATE message dev_mess;
45026    PRIVATE major, minor, task;
45027
45028    FORWARD _PROTOTYPE( void find_dev, (Dev_t dev)                          );
45029
45030    /*===========================================================================*
45031     *                              dev_io                                       *
45032     *===========================================================================*/
45033    PUBLIC int dev_io(op, nonblock, dev, pos, bytes, proc, buff)
45034    int op;                         /* DEV_READ, DEV_WRITE, DEV_IOCTL, etc. */
45035    int nonblock;                   /* TRUE if nonblocking op */
45036    dev_t dev;                      /* major-minor device number */
45037    off_t pos;                      /* byte position */
45038    int bytes;                      /* how many bytes to transfer */
45039    int proc;                       /* in whose address space is buff? */
45040    char *buff;                     /* virtual address of the buffer */
45041    {
45042    /* Read or write from a device.  The parameter 'dev' tells which one. */
45043
45044      find_dev(dev);                /* load the variables major, minor, and task */
45045
45046      /* Set up the message passed to task. */
45047      dev_mess.m_type   = op;
45048      dev_mess.DEVICE   = (dev >> MINOR) & BYTE;
45049      dev_mess.POSITION = pos;
45050      dev_mess.PROC_NR  = proc;
45051      dev_mess.ADDRESS  = buff;
45052      dev_mess.COUNT    = bytes;
45053      dev_mess.TTY_FLAGS = nonblock; /* temporary kludge */
45054
```

```
45055     /* Call the task. */
45056     (*dmap[major].dmap_rw)(task, &dev_mess);
45057
45058     /* Task has completed.  See if call completed. */
45059     if (dev_mess.REP_STATUS == SUSPEND) {
45060           if (op == DEV_OPEN) task = XPOPEN;
45061           suspend(task);          /* suspend user */
45062     }
45063
45064     return(dev_mess.REP_STATUS);
45065   }


45068   /*===========================================================================*
45069    *                              dev_opcl                                     *
45070    *===========================================================================*/
45071   PUBLIC void dev_opcl(task_nr, mess_ptr)
45072   int task_nr;                  /* which task */
45073   message *mess_ptr;            /* message pointer */
45074   {
45075   /* Called from the dmap struct in table.c on opens & closes of special files.*/
45076
45077     int op;
45078
45079     op = mess_ptr->m_type;        /* save DEV_OPEN or DEV_CLOSE for later */
45080     mess_ptr->DEVICE = (mess_ptr->DEVICE >> MINOR) & BYTE;
45081     mess_ptr->PROC_NR = fp - fproc;
45082
45083     call_task(task_nr, mess_ptr);
45084
45085     /* Task has completed.  See if call completed. */
45086     if (mess_ptr->REP_STATUS == SUSPEND) {
45087           if (op == DEV_OPEN) task_nr = XPOPEN;
45088           suspend(task_nr);       /* suspend user */
45089     }
45090   }

45092   /*===========================================================================*
45093    *                              tty_open                                     *
45094    *===========================================================================*/
45095   PUBLIC void tty_open(task_nr, mess_ptr)
45096   int task_nr;
45097   message *mess_ptr;
45098   {
45099   /* This procedure is called from the dmap struct in table.c on tty opens. */
45100
45101     int r;
45102     dev_t dev;
45103     int flags, proc;
45104     register struct fproc *rfp;
45105
45106     dev = (dev_t) mess_ptr->DEVICE;
45107     flags = mess_ptr->COUNT;
45108     proc = fp - fproc;
45109
45110     /* Add O_NOCTTY to the flags if this process is not a session leader, or
45111      * if it already has a controlling tty, or if it is someone elses
45112      * controlling tty.
45113      */
45114     if (!fp->fp_sesldr || fp->fp_tty != 0) {
```

```
45115           flags |= O_NOCTTY;
45116     } else {
45117           for (rfp = &fproc[LOW_USER]; rfp < &fproc[NR_PROCS]; rfp++) {
45118                   if (rfp->fp_tty == dev) flags |= O_NOCTTY;
45119           }
45120     }
45121
45122     r = dev_io(DEV_OPEN, mode, dev, (off_t) 0, flags, proc, NIL_PTR);
45123
45124     if (r == 1) {
45125           fp->fp_tty = dev;
45126           r = OK;
45127     }
45128
45129     mess_ptr->REP_STATUS = r;
45130   }


45133   /*===========================================================================*
45134    *                              ctty_open                                    *
45135    *===========================================================================*/
45136   PUBLIC void ctty_open(task_nr, mess_ptr)
45137   int task_nr;
45138   message *mess_ptr;
45139   {
45140   /* This procedure is called from the dmap struct in table.c on opening
45141    * /dev/tty, the magic device that translates to the controlling tty.
45142    */
45143
45144     mess_ptr->REP_STATUS = fp->fp_tty == 0 ? ENXIO : OK;
45145   }


45148   /*===========================================================================*
45149    *                              ctty_close                                   *
45150    *===========================================================================*/
45151   PUBLIC void ctty_close(task_nr, mess_ptr)
45152   int task_nr;
45153   message *mess_ptr;
45154   {
45155   /* Close /dev/tty. */
45156
45157     mess_ptr->REP_STATUS = OK;
45158   }


45161   /*===========================================================================*
45162    *                              do_setsid                                    *
45163    *===========================================================================*/
45164   PUBLIC int do_setsid()
45165   {
45166   /* Perform the FS side of the SETSID call, i.e. get rid of the controlling
45167    * terminal of a process, and make the process a session leader.
45168    */
45169     register struct fproc *rfp;
45170
45171     /* Only MM may do the SETSID call directly. */
45172     if (who != MM_PROC_NR) return(ENOSYS);
45173
45174     /* Make the process a session leader with no controlling tty. */
```

```
45175     rfp = &fproc[slot1];
45176     rfp->fp_sesldr = TRUE;
45177     rfp->fp_tty = 0;
45178   }


45181   /*===========================================================================*
45182    *                              do_ioctl                                     *
45183    *===========================================================================*/
45184   PUBLIC int do_ioctl()
45185   {
45186   /* Perform the ioctl(ls_fd, request, argx) system call (uses m2 fmt). */
45187
45188     struct filp *f;
45189     register struct inode *rip;
45190     dev_t dev;
45191
45192     if ( (f = get_filp(ls_fd)) == NIL_FILP) return(err_code);
45193     rip = f->filp_ino;            /* get inode pointer */
45194     if ( (rip->i_mode & I_TYPE) != I_CHAR_SPECIAL
45195           && (rip->i_mode & I_TYPE) != I_BLOCK_SPECIAL) return(ENOTTY);
45196     dev = (dev_t) rip->i_zone[0];
45197     find_dev(dev);
45198
45199     dev_mess= m;
45200
45201     dev_mess.m_type  = DEV_IOCTL;
45202     dev_mess.PROC_NR = who;
45203     dev_mess.TTY_LINE = minor;
45204
45205     /* Call the task. */
45206     (*dmap[major].dmap_rw)(task, &dev_mess);
45207
45208     /* Task has completed.  See if call completed. */
45209     if (dev_mess.REP_STATUS == SUSPEND) {
45210           if (f->filp_flags & O_NONBLOCK) {
45211                   /* Not supposed to block. */
45212                   dev_mess.m_type = CANCEL;
45213                   dev_mess.PROC_NR = who;
45214                   dev_mess.TTY_LINE = minor;
45215                   (*dmap[major].dmap_rw)(task, &dev_mess);
45216                   if (dev_mess.REP_STATUS == EINTR) dev_mess.REP_STATUS = EAGAIN;
45217           } else {
45218                   suspend(task);          /* User must be suspended. */
45219           }
45220     }
45221   #if ENABLE_BINCOMPAT
45222     m1.TTY_SPEK = dev_mess.TTY_SPEK;      /* erase and kill */
45223     m1.TTY_FLAGS = dev_mess.TTY_FLAGS;    /* flags */
45224   #endif
45225     return(dev_mess.REP_STATUS);
45226   }


45229   /*===========================================================================*
45230    *                              find_dev                                     *
45231    *===========================================================================*/
45232   PRIVATE void find_dev(dev)
45233   dev_t dev;                      /* device */
45234   {
```

```
45235   /* Extract the major and minor device number from the parameter. */
45236
45237     major = (dev >> MAJOR) & BYTE;        /* major device number */
45238     minor = (dev >> MINOR) & BYTE;        /* minor device number */
45239     if (major >= max_major) {
45240           major = minor = 0;              /* will fail with ENODEV */
45241     }
45242     task = dmap[major].dmap_task; /* which task services the device */
45243   }


45246   /*===========================================================================*
45247    *                              call_task                                    *
45248    *===========================================================================*/
45249   PUBLIC void call_task(task_nr, mess_ptr)
45250   int task_nr;                    /* which task to call */
45251   message *mess_ptr;              /* pointer to message for task */
45252   {
45253   /* All file system I/O ultimately comes down to I/O on major/minor device
45254    * pairs.  These lead to calls on the following routines via the dmap table.
45255    */
45256
45257     int r, proc_nr;
45258     message local_m;
45259
45260     proc_nr = mess_ptr->PROC_NR;
45261
45262     while ((r = sendrec(task_nr, mess_ptr)) == ELOCKED) {
45263           /* sendrec() failed to avoid deadlock. The task 'task_nr' is
45264            * trying to send a REVIVE message for an earlier request.
45265            * Handle it and go try again.
45266            */
45267           if ((r = receive(task_nr, &local_m)) != OK) break;
45268
45269           /* If we're trying to send a cancel message to a task which has just
45270            * sent a completion reply, ignore the reply and abort the cancel
45271            * request. The caller will do the revive for the process.
45272            */
45273           if (mess_ptr->m_type == CANCEL && local_m.REP_PROC_NR == proc_nr)
45274                   return;
45275
45276           /* Otherwise it should be a REVIVE. */
45277           if (local_m.m_type != REVIVE) {
45278                   printf(
45279                   "fs: strange device reply from %d, type = %d, proc = %d\n",
45280                           local_m.m_source,
45281                           local_m.m_type, local_m.REP_PROC_NR);
45282                   continue;
45283           }
45284
45285           revive(local_m.REP_PROC_NR, local_m.REP_STATUS);
45286     }
45287
45288     /* The message received may be a reply to this call, or a REVIVE for some
45289      * other process.
45290      */
45291     for (;;) {
45292           if (r != OK) panic("call_task: can't send/receive", NO_NUM);
45293
45294           /* Did the process we did the sendrec() for get a result? */
```

```
45295           if (mess_ptr->REP_PROC_NR == proc_nr) break;
45296
45297           /* Otherwise it should be a REVIVE. */
45298           if (mess_ptr->m_type != REVIVE) {
45299                   printf(
45300                   "fs: strange device reply from %d, type = %d, proc = %d\n",
45301                           mess_ptr->m_source,
45302                           mess_ptr->m_type, mess_ptr->REP_PROC_NR);
45303                   continue;
45304           }
45305           revive(mess_ptr->REP_PROC_NR, mess_ptr->REP_STATUS);
45306
45307           r = receive(task_nr, mess_ptr);
45308     }
45309   }


45312   /*===========================================================================*
45313    *                              call_ctty                                    *
45314    *===========================================================================*/
45315   PUBLIC void call_ctty(task_nr, mess_ptr)
45316   int task_nr;                    /* not used - for compatibility with dmap_t */
45317   message *mess_ptr;              /* pointer to message for task */
45318   {
45319   /* This routine is only called for one device, namely /dev/tty.  Its job
45320    * is to change the message to use the controlling terminal, instead of the
45321    * major/minor pair for /dev/tty itself.
45322    */
45323
45324     int major_device;
45325
45326     if (fp->fp_tty == 0) {
45327           /* No controlling tty present anymore, return an I/O error. */
45328           mess_ptr->REP_STATUS = EIO;
45329           return;
45330     }
45331     major_device = (fp->fp_tty >> MAJOR) & BYTE;
45332     task_nr = dmap[major_device].dmap_task;       /* task for controlling tty */
45333     mess_ptr->DEVICE = (fp->fp_tty >> MINOR) & BYTE;
45334     call_task(task_nr, mess_ptr);
45335   }


45338   /*===========================================================================*
45339    *                              no_dev                                       *
45340    *===========================================================================*/
45341   PUBLIC void no_dev(task_nr, m_ptr)
45342   int task_nr;                    /* not used - for compatibility with dmap_t */
45343   message *m_ptr;                 /* message pointer */
45344   {
45345   /* No device there. */
45346
45347     m_ptr->REP_STATUS = ENODEV;
45348   }


45351   #if ENABLE_NETWORKING
45352   /*===========================================================================*
45353    *                              net_open                                     *
45354    *===========================================================================*/
```

```
45355    PUBLIC void net_open(task_nr, mess_ptr)
45356    int task_nr;                    /* task to send message to */
45357    message *mess_ptr;              /* pointer to message to send */
45358    {
45359    /* Network files need special processing upon open.  A network device is
45360     * "cloned", i.e. on a succesful open it is replaced by a new network device
45361     * with a new unique minor device number.  This new device number identifies
45362     * the new IP connection with the network task.
45363     */
45364
45365      dev_t dev;
45366      struct inode *rip, *nrip;
45367      int result;
45368      int ncount, proc;
45369
45370      rip = fp->fp_filp[fd]->filp_ino;
45371
45372      nrip = alloc_inode(rip->i_dev, ALL_MODES | I_CHAR_SPECIAL);
45373      if (nrip == NIL_INODE) {
45374            mess_ptr->REP_STATUS = err_code;
45375            return;
45376      }
45377
45378      dev = (dev_t) mess_ptr->DEVICE;
45379      ncount = mess_ptr->COUNT;
45380      proc = fp - fproc;
45381      result = dev_io(DEV_OPEN, mode, dev, (off_t) 0, ncount, proc, NIL_PTR);
45382
45383      if (result < 0) {
45384            put_inode(nrip);
45385            mess_ptr->REP_STATUS = result;
45386            return;
45387      }
45388
45389      dev= rip->i_zone[0];
45390      dev= (dev & ~(BYTE << MINOR)) | ((result & BYTE) << MINOR);
45391
45392      nrip->i_zone[0]= dev;
45393      put_inode(rip);
45394      fp->fp_filp[fd]->filp_ino = nrip;
45395      mess_ptr->REP_STATUS = OK;
45396    }
45397    #endif /* ENABLE_NETWORKING */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/filedes.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

45400    /* This file contains the procedures that manipulate file descriptors.
45401     *
45402     * The entry points into this file are
45403     *   get_fd:    look for free file descriptor and free filp slots
45404     *   get_filp:  look up the filp entry for a given file descriptor
45405     *   find_filp: find a filp slot that points to a given inode
45406     */
45407
45408    #include "fs.h"
45409    #include "file.h"
```

```
45410   #include "fproc.h"
45411   #include "inode.h"
45412
45413   /*===========================================================================*
45414    *                              get_fd                                       *
45415    *===========================================================================*/
45416   PUBLIC int get_fd(start, bits, k, fpt)
45417   int start;                    /* start of search (used for F_DUPFD) */
45418   mode_t bits;                  /* mode of the file to be created (RWX bits) */
45419   int *k;                       /* place to return file descriptor */
45420   struct filp **fpt;            /* place to return filp slot */
45421   {
45422   /* Look for a free file descriptor and a free filp slot.  Fill in the mode word
45423    * in the latter, but don't claim either one yet, since the open() or creat()
45424    * may yet fail.
45425    */
45426
45427     register struct filp *f;
45428     register int i;
45429
45430     *k = -1;                    /* we need a way to tell if file desc found */
45431
45432     /* Search the fproc fp_filp table for a free file descriptor. */
45433     for (i = start; i < OPEN_MAX; i++) {
45434           if (fp->fp_filp[i] == NIL_FILP) {
45435                   /* A file descriptor has been located. */
45436                   *k = i;
45437                   break;
45438           }
45439     }
45440
45441     /* Check to see if a file descriptor has been found. */
45442     if (*k < 0) return(EMFILE);   /* this is why we initialized k to -1 */
45443
45444     /* Now that a file descriptor has been found, look for a free filp slot. */
45445     for (f = &filp[0]; f < &filp[NR_FILPS]; f++) {
45446           if (f->filp_count == 0) {
45447                   f->filp_mode = bits;
45448                   f->filp_pos = 0L;
45449                   f->filp_flags = 0;
45450                   *fpt = f;
45451                   return(OK);
45452           }
45453     }
45454
45455     /* If control passes here, the filp table must be full.  Report that back. */
45456     return(ENFILE);
45457   }


45460   /*===========================================================================*
45461    *                              get_filp                                     *
45462    *===========================================================================*/
45463   PUBLIC struct filp *get_filp(fild)
45464   int fild;                     /* file descriptor */
45465   {
45466   /* See if 'fild' refers to a valid file descr.  If so, return its filp ptr. */
45467
45468     err_code = EBADF;
45469     if (fild < 0 || fild >= OPEN_MAX ) return(NIL_FILP);
```

```
45470       return(fp->fp_filp[fild]);    /* may also be NIL_FILP */
45471     }


45474     /*===========================================================================*
45475      *                                find_filp                                  *
45476      *===========================================================================*/
45477     PUBLIC struct filp *find_filp(rip, bits)
45478     register struct inode *rip;     /* inode referred to by the filp to be found */
45479     Mode_t bits;                    /* mode of the filp to be found (RWX bits) */
45480     {
45481     /* Find a filp slot that refers to the inode 'rip' in a way as described
45482      * by the mode bit 'bits'. Used for determining whether somebody is still
45483      * interested in either end of a pipe.  Also used when opening a FIFO to
45484      * find partners to share a filp field with (to shared the file position).
45485      * Like 'get_fd' it performs its job by linear search through the filp table.
45486      */
45487
45488       register struct filp *f;
45489
45490       for (f = &filp[0]; f < &filp[NR_FILPS]; f++) {
45491             if (f->filp_count != 0 && f->filp_ino == rip && (f->filp_mode & bits)){
45492                   return(f);
45493             }
45494       }
45495
45496       /* If control passes here, the filp wasn't there.  Report that back. */
45497       return(NIL_FILP);
45498     }


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/inode.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

45500     /* This file manages the inode table.  There are procedures to allocate and
45501      * deallocate inodes, acquire, erase, and release them, and read and write
45502      * them from the disk.
45503      *
45504      * The entry points into this file are
45505      *   get_inode:    search inode table for a given inode; if not there, read it
45506      *   put_inode:    indicate that an inode is no longer needed in memory
45507      *   alloc_inode:  allocate a new, unused inode
45508      *   wipe_inode:   erase some fields of a newly allocated inode
45509      *   free_inode:   mark an inode as available for a new file
45510      *   update_times: update atime, ctime, and mtime
45511      *   rw_inode:     read a disk block and extract an inode, or corresp. write
45512      *   old_icopy:    copy to/from in-core inode struct and disk inode (V1.x)
45513      *   new_icopy:    copy to/from in-core inode struct and disk inode (V2.x)
45514      *   dup_inode:    indicate that someone else is using an inode table entry
45515      */
45516
45517     #include "fs.h"
45518     #include <minix/boot.h>
45519     #include "buf.h"
45520     #include "file.h"
45521     #include "fproc.h"
45522     #include "inode.h"
45523     #include "super.h"
45524
```

```
45525   FORWARD _PROTOTYPE( void old_icopy, (struct inode *rip, d1_inode *dip,
45526                                         int direction, int norm));
45527   FORWARD _PROTOTYPE( void new_icopy, (struct inode *rip, d2_inode *dip,
45528                                         int direction, int norm));
45529
45530
45531   /*===========================================================================*
45532    *                               get_inode                                   *
45533    *===========================================================================*/
45534   PUBLIC struct inode *get_inode(dev, numb)
45535   dev_t dev;                      /* device on which inode resides */
45536   int numb;                       /* inode number (ANSI: may not be unshort) */
45537   {
45538   /* Find a slot in the inode table, load the specified inode into it, and
45539    * return a pointer to the slot.  If 'dev' == NO_DEV, just return a free slot.
45540    */
45541
45542     register struct inode *rip, *xp;
45543
45544     /* Search the inode table both for (dev, numb) and a free slot. */
45545     xp = NIL_INODE;
45546     for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++) {
45547           if (rip->i_count > 0) { /* only check used slots for (dev, numb) */
45548                   if (rip->i_dev == dev && rip->i_num == numb) {
45549                           /* This is the inode that we are looking for. */
45550                           rip->i_count++;
45551                           return(rip);    /* (dev, numb) found */
45552                   }
45553           } else {
45554                   xp = rip;       /* remember this free slot for later */
45555           }
45556     }
45557
45558     /* Inode we want is not currently in use.  Did we find a free slot? */
45559     if (xp == NIL_INODE) {        /* inode table completely full */
45560           err_code = ENFILE;
45561           return(NIL_INODE);
45562     }
45563
45564     /* A free inode slot has been located.  Load the inode into it. */
45565     xp->i_dev = dev;
45566     xp->i_num = numb;
45567     xp->i_count = 1;
45568     if (dev != NO_DEV) rw_inode(xp, READING);     /* get inode from disk */
45569     xp->i_update = 0;             /* all the times are initially up-to-date */
45570
45571     return(xp);
45572   }


45575   /*===========================================================================*
45576    *                               put_inode                                   *
45577    *===========================================================================*/
45578   PUBLIC void put_inode(rip)
45579   register struct inode *rip;     /* pointer to inode to be released */
45580   {
45581   /* The caller is no longer using this inode.  If no one else is using it either
45582    * write it back to the disk immediately.  If it has no links, truncate it and
45583    * return it to the pool of available inodes.
45584    */
```

```
45585
45586     if (rip == NIL_INODE) return; /* checking here is easier than in caller */
45587     if (--rip->i_count == 0) {    /* i_count == 0 means no one is using it now */
45588           if ((rip->i_nlinks & BYTE) == 0) {
45589                   /* i_nlinks == 0 means free the inode. */
45590                   truncate(rip);  /* return all the disk blocks */
45591                   rip->i_mode = I_NOT_ALLOC;      /* clear I_TYPE field */
45592                   rip->i_dirt = DIRTY;
45593                   free_inode(rip->i_dev, rip->i_num);
45594           } else {
45595                   if (rip->i_pipe == I_PIPE) truncate(rip);
45596           }
45597           rip->i_pipe = NO_PIPE;  /* should always be cleared */
45598           if (rip->i_dirt == DIRTY) rw_inode(rip, WRITING);
45599     }
45600   }

45602   /*===========================================================================*
45603    *                              alloc_inode                                  *
45604    *===========================================================================*/
45605   PUBLIC struct inode *alloc_inode(dev, bits)
45606   dev_t dev;                    /* device on which to allocate the inode */
45607   mode_t bits;                  /* mode of the inode */
45608   {
45609   /* Allocate a free inode on 'dev', and return a pointer to it. */
45610
45611     register struct inode *rip;
45612     register struct super_block *sp;
45613     int major, minor, inumb;
45614     bit_t b;
45615
45616     sp = get_super(dev);  /* get pointer to super_block */
45617     if (sp->s_rd_only) {  /* can't allocate an inode on a read only device. */
45618           err_code = EROFS;
45619           return(NIL_INODE);
45620     }
45621
45622     /* Acquire an inode from the bit map. */
45623     b = alloc_bit(sp, IMAP, sp->s_isearch);
45624     if (b == NO_BIT) {
45625           err_code = ENFILE;
45626           major = (int) (sp->s_dev >> MAJOR) & BYTE;
45627           minor = (int) (sp->s_dev >> MINOR) & BYTE;
45628           printf("Out of i-nodes on %sdevice %d/%d\n",
45629                   sp->s_dev == ROOT_DEV ? "root " : "", major, minor);
45630           return(NIL_INODE);
45631     }
45632     sp->s_isearch = b;            /* next time start here */
45633     inumb = (int) b;              /* be careful not to pass unshort as param */
45634
45635     /* Try to acquire a slot in the inode table. */
45636     if ((rip = get_inode(NO_DEV, inumb)) == NIL_INODE) {
45637           /* No inode table slots available.  Free the inode just allocated. */
45638           free_bit(sp, IMAP, b);
45639     } else {
45640           /* An inode slot is available. Put the inode just allocated into it. */
45641           rip->i_mode = bits;             /* set up RWX bits */
45642           rip->i_nlinks = (nlink_t) 0;    /* initial no links */
45643           rip->i_uid = fp->fp_effuid;     /* file's uid is owner's */
45644           rip->i_gid = fp->fp_effgid;     /* ditto group id */
```

```
45645           rip->i_dev = dev;               /* mark which device it is on */
45646           rip->i_ndzones = sp->s_ndzones; /* number of direct zones */
45647           rip->i_nindirs = sp->s_nindirs; /* number of indirect zones per blk*/
45648           rip->i_sp = sp;                 /* pointer to super block */
45649
45650           /* Fields not cleared already are cleared in wipe_inode().  They have
45651            * been put there because truncate() needs to clear the same fields if
45652            * the file happens to be open while being truncated.  It saves space
45653            * not to repeat the code twice.
45654            */
45655           wipe_inode(rip);
45656     }
45657
45658     return(rip);
45659   }

45661   /*===========================================================================*
45662    *                              wipe_inode                                   *
45663    *===========================================================================*/
45664   PUBLIC void wipe_inode(rip)
45665   register struct inode *rip;     /* the inode to be erased */
45666   {
45667   /* Erase some fields in the inode.  This function is called from alloc_inode()
45668    * when a new inode is to be allocated, and from truncate(), when an existing
45669    * inode is to be truncated.
45670    */
45671
45672     register int i;
45673
45674     rip->i_size = 0;
45675     rip->i_update = ATIME | CTIME | MTIME;        /* update all times later */
45676     rip->i_dirt = DIRTY;
45677     for (i = 0; i < V2_NR_TZONES; i++) rip->i_zone[i] = NO_ZONE;
45678   }


45681   /*===========================================================================*
45682    *                              free_inode                                   *
45683    *===========================================================================*/
45684   PUBLIC void free_inode(dev, inumb)
45685   dev_t dev;                      /* on which device is the inode */
45686   ino_t inumb;                    /* number of inode to be freed */
45687   {
45688   /* Return an inode to the pool of unallocated inodes. */
45689
45690     register struct super_block *sp;
45691     bit_t b;
45692
45693     /* Locate the appropriate super_block. */
45694     sp = get_super(dev);
45695     if (inumb <= 0 || inumb > sp->s_ninodes) return;
45696     b = inumb;
45697     free_bit(sp, IMAP, b);
45698     if (b < sp->s_isearch) sp->s_isearch = b;
45699   }

45701   /*===========================================================================*
45702    *                              update_times                                 *
45703    *===========================================================================*/
45704   PUBLIC void update_times(rip)
```

```
45705   register struct inode *rip;     /* pointer to inode to be read/written */
45706   {
45707   /* Various system calls are required by the standard to update atime, ctime,
45708    * or mtime.  Since updating a time requires sending a message to the clock
45709    * task--an expensive business--the times are marked for update by setting
45710    * bits in i_update.  When a stat, fstat, or sync is done, or an inode is
45711    * released, update_times() may be called to actually fill in the times.
45712    */
45713
45714     time_t cur_time;
45715     struct super_block *sp;
45716
45717     sp = rip->i_sp;               /* get pointer to super block. */
45718     if (sp->s_rd_only) return;    /* no updates for read-only file systems */
45719
45720     cur_time = clock_time();
45721     if (rip->i_update & ATIME) rip->i_atime = cur_time;
45722     if (rip->i_update & CTIME) rip->i_ctime = cur_time;
45723     if (rip->i_update & MTIME) rip->i_mtime = cur_time;
45724     rip->i_update = 0;            /* they are all up-to-date now */
45725   }


45728   /*===========================================================================*
45729    *                                rw_inode                                   *
45730    *===========================================================================*/
45731   PUBLIC void rw_inode(rip, rw_flag)
45732   register struct inode *rip;     /* pointer to inode to be read/written */
45733   int rw_flag;                    /* READING or WRITING */
45734   {
45735   /* An entry in the inode table is to be copied to or from the disk. */
45736
45737     register struct buf *bp;
45738     register struct super_block *sp;
45739     d1_inode *dip;
45740     d2_inode *dip2;
45741     block_t b, offset;
45742
45743     /* Get the block where the inode resides. */
45744     sp = get_super(rip->i_dev);   /* get pointer to super block */
45745     rip->i_sp = sp;               /* inode must contain super block pointer */
45746     offset = sp->s_imap_blocks + sp->s_zmap_blocks + 2;
45747     b = (block_t) (rip->i_num - 1)/sp->s_inodes_per_block + offset;
45748     bp = get_block(rip->i_dev, b, NORMAL);
45749     dip  = bp->b_v1_ino + (rip->i_num - 1) % V1_INODES_PER_BLOCK;
45750     dip2 = bp->b_v2_ino + (rip->i_num - 1) % V2_INODES_PER_BLOCK;
45751
45752     /* Do the read or write. */
45753     if (rw_flag == WRITING) {
45754           if (rip->i_update) update_times(rip);   /* times need updating */
45755           if (sp->s_rd_only == FALSE) bp->b_dirt = DIRTY;
45756     }
45757
45758     /* Copy the inode from the disk block to the in-core table or vice versa.
45759      * If the fourth parameter below is FALSE, the bytes are swapped.
45760      */
45761     if (sp->s_version == V1)
45762           old_icopy(rip, dip,  rw_flag, sp->s_native);
45763     else
45764           new_icopy(rip, dip2, rw_flag, sp->s_native);
```

```
45765
45766       put_block(bp, INODE_BLOCK);
45767       rip->i_dirt = CLEAN;
45768     }


45771     /*===========================================================================*
45772      *                              old_icopy                                    *
45773      *===========================================================================*/
45774     PRIVATE void old_icopy(rip, dip, direction, norm)
45775     register struct inode *rip;     /* pointer to the in-core inode struct */
45776     register d1_inode *dip;         /* pointer to the d1_inode inode struct */
45777     int direction;                  /* READING (from disk) or WRITING (to disk) */
45778     int norm;                       /* TRUE = do not swap bytes; FALSE = swap */
45779
45780     {
45781     /* The V1.x IBM disk, the V1.x 68000 disk, and the V2 disk (same for IBM and
45782      * 68000) all have different inode layouts.  When an inode is read or written
45783      * this routine handles the conversions so that the information in the inode
45784      * table is independent of the disk structure from which the inode came.
45785      * The old_icopy routine copies to and from V1 disks.
45786      */
45787
45788       int i;
45789
45790       if (direction == READING) {
45791             /* Copy V1.x inode to the in-core table, swapping bytes if need be. */
45792             rip->i_mode    = conv2(norm, (int) dip->d1_mode);
45793             rip->i_uid     = conv2(norm, (int) dip->d1_uid );
45794             rip->i_size    = conv4(norm,       dip->d1_size);
45795             rip->i_mtime   = conv4(norm,       dip->d1_mtime);
45796             rip->i_atime   = rip->i_mtime;
45797             rip->i_ctime   = rip->i_mtime;
45798             rip->i_nlinks  = (nlink_t) dip->d1_nlinks;      /* 1 char */
45799             rip->i_gid     = (gid_t) dip->d1_gid;           /* 1 char */
45800             rip->i_ndzones = V1_NR_DZONES;
45801             rip->i_nindirs = V1_INDIRECTS;
45802             for (i = 0; i < V1_NR_TZONES; i++)
45803                     rip->i_zone[i] = conv2(norm, (int) dip->d1_zone[i]);
45804       } else {
45805             /* Copying V1.x inode to disk from the in-core table. */
45806             dip->d1_mode   = conv2(norm, (int) rip->i_mode);
45807             dip->d1_uid    = conv2(norm, (int) rip->i_uid );
45808             dip->d1_size   = conv4(norm,       rip->i_size);
45809             dip->d1_mtime  = conv4(norm,       rip->i_mtime);
45810             dip->d1_nlinks = (nlink_t) rip->i_nlinks;       /* 1 char */
45811             dip->d1_gid    = (gid_t) rip->i_gid;            /* 1 char */
45812             for (i = 0; i < V1_NR_TZONES; i++)
45813                     dip->d1_zone[i] = conv2(norm, (int) rip->i_zone[i]);
45814       }
45815     }


45818     /*===========================================================================*
45819      *                              new_icopy                                    *
45820      *===========================================================================*/
45821     PRIVATE void new_icopy(rip, dip, direction, norm)
45822     register struct inode *rip;     /* pointer to the in-core inode struct */
45823     register d2_inode *dip; /* pointer to the d2_inode struct */
45824     int direction;                  /* READING (from disk) or WRITING (to disk) */
```

```
45825   int norm;                       /* TRUE = do not swap bytes; FALSE = swap */
45826
45827   {
45828   /* Same as old_icopy, but to/from V2 disk layout. */
45829
45830     int i;
45831
45832     if (direction == READING) {
45833           /* Copy V2.x inode to the in-core table, swapping bytes if need be. */
45834           rip->i_mode    = conv2(norm,dip->d2_mode);
45835           rip->i_uid     = conv2(norm,dip->d2_uid );
45836           rip->i_nlinks  = conv2(norm,(int) dip->d2_nlinks);
45837           rip->i_gid     = conv2(norm,(int) dip->d2_gid );
45838           rip->i_size    = conv4(norm,dip->d2_size);
45839           rip->i_atime   = conv4(norm,dip->d2_atime);
45840           rip->i_ctime   = conv4(norm,dip->d2_ctime);
45841           rip->i_mtime   = conv4(norm,dip->d2_mtime);
45842           rip->i_ndzones = V2_NR_DZONES;
45843           rip->i_nindirs = V2_INDIRECTS;
45844           for (i = 0; i < V2_NR_TZONES; i++)
45845                   rip->i_zone[i] = conv4(norm, (long) dip->d2_zone[i]);
45846     } else {
45847           /* Copying V2.x inode to disk from the in-core table. */
45848           dip->d2_mode   = conv2(norm,rip->i_mode);
45849           dip->d2_uid    = conv2(norm,rip->i_uid );
45850           dip->d2_nlinks = conv2(norm,rip->i_nlinks);
45851           dip->d2_gid    = conv2(norm,rip->i_gid );
45852           dip->d2_size   = conv4(norm,rip->i_size);
45853           dip->d2_atime  = conv4(norm,rip->i_atime);
45854           dip->d2_ctime  = conv4(norm,rip->i_ctime);
45855           dip->d2_mtime  = conv4(norm,rip->i_mtime);
45856           for (i = 0; i < V2_NR_TZONES; i++)
45857                   dip->d2_zone[i] = conv4(norm, (long) rip->i_zone[i]);
45858     }
45859   }


45862   /*===========================================================================*
45863    *                               dup_inode                                   *
45864    *===========================================================================*/
45865   PUBLIC void dup_inode(ip)
45866   struct inode *ip;               /* The inode to be duplicated. */
45867   {
45868   /* This routine is a simplified form of get_inode() for the case where
45869    * the inode pointer is already known.
45870    */
45871
45872     ip->i_count++;
45873   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/link.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

45900    /* This file handles the LINK and UNLINK system calls.  It also deals with
45901     * deallocating the storage used by a file when the last UNLINK is done to a
45902     * file and the blocks must be returned to the free block pool.
45903     *
45904     * The entry points into this file are
45905     *   do_link:   perform the LINK system call
45906     *   do_unlink: perform the UNLINK and RMDIR system calls
45907     *   do_rename: perform the RENAME system call
45908     *   truncate:  release all the blocks associated with an inode
45909     */
45910
45911    #include "fs.h"
45912    #include <sys/stat.h>
45913    #include <string.h>
45914    #include <minix/callnr.h>
45915    #include "buf.h"
45916    #include "file.h"
45917    #include "fproc.h"
45918    #include "inode.h"
45919    #include "param.h"
45920    #include "super.h"
45921
45922    #define SAME 1000
45923
45924    FORWARD _PROTOTYPE( int remove_dir, (struct inode *rldirp, struct inode *rip,
45925                            char dir_name[NAME_MAX])                        );
45926
45927    FORWARD _PROTOTYPE( int unlink_file, (struct inode *dirp, struct inode *rip,
45928                            char file_name[NAME_MAX])                       );
45929
45930
45931    /*===========================================================================*
45932     *                                do_link                                    *
45933     *===========================================================================*/
45934    PUBLIC int do_link()
45935    {
45936    /* Perform the link(name1, name2) system call. */
45937
45938      register struct inode *ip, *rip;
45939      register int r;
45940      char string[NAME_MAX];
45941      struct inode *new_ip;
45942
45943      /* See if 'name' (file to be linked) exists. */
45944      if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
45945      if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
45946
45947      /* Check to see if the file has maximum number of links already. */
45948      r = OK;
45949      if ( (rip->i_nlinks & BYTE) >= LINK_MAX) r = EMLINK;
45950
45951      /* Only super_user may link to directories. */
45952      if (r == OK)
45953            if ( (rip->i_mode & I_TYPE) == I_DIRECTORY && !super_user) r = EPERM;
45954
```

```
45955     /* If error with 'name', return the inode. */
45956     if (r != OK) {
45957           put_inode(rip);
45958           return(r);
45959     }
45960
45961     /* Does the final directory of 'name2' exist? */
45962     if (fetch_name(name2, name2_length, M1) != OK) {
45963           put_inode(rip);
45964           return(err_code);
45965     }
45966     if ( (ip = last_dir(user_path, string)) == NIL_INODE) r = err_code;
45967
45968     /* If 'name2' exists in full (even if no space) set 'r' to error. */
45969     if (r == OK) {
45970           if ( (new_ip = advance(ip, string)) == NIL_INODE) {
45971                   r = err_code;
45972                   if (r == ENOENT) r = OK;
45973           } else {
45974                   put_inode(new_ip);
45975                   r = EEXIST;
45976           }
45977     }
45978
45979     /* Check for links across devices. */
45980     if (r == OK)
45981           if (rip->i_dev != ip->i_dev) r = EXDEV;
45982
45983     /* Try to link. */
45984     if (r == OK)
45985           r = search_dir(ip, string, &rip->i_num, ENTER);
45986
45987     /* If success, register the linking. */
45988     if (r == OK) {
45989           rip->i_nlinks++;
45990           rip->i_update |= CTIME;
45991           rip->i_dirt = DIRTY;
45992     }
45993
45994     /* Done.  Release both inodes. */
45995     put_inode(rip);
45996     put_inode(ip);
45997     return(r);
45998   }


46001   /*===========================================================================*
46002    *                                do_unlink                                  *
46003    *===========================================================================*/
46004   PUBLIC int do_unlink()
46005   {
46006   /* Perform the unlink(name) or rmdir(name) system call. The code for these two
46007    * is almost the same.  They differ only in some condition testing.  Unlink()
46008    * may be used by the superuser to do dangerous things; rmdir() may not.
46009    */
46010
46011     register struct inode *rip;
46012     struct inode *rldirp;
46013     int r;
46014     char string[NAME_MAX];
```

```
46015
46016     /* Get the last directory in the path. */
46017     if (fetch_name(name, name_length, M3) != OK) return(err_code);
46018     if ( (rldirp = last_dir(user_path, string)) == NIL_INODE)
46019           return(err_code);
46020
46021     /* The last directory exists.  Does the file also exist? */
46022     r = OK;
46023     if ( (rip = advance(rldirp, string)) == NIL_INODE) r = err_code;
46024
46025     /* If error, return inode. */
46026     if (r != OK) {
46027           put_inode(rldirp);
46028           return(r);
46029     }
46030
46031     /* Do not remove a mount point. */
46032     if (rip->i_num == ROOT_INODE) {
46033           put_inode(rldirp);
46034           put_inode(rip);
46035           return(EBUSY);
46036     }
46037
46038     /* Now test if the call is allowed, separately for unlink() and rmdir(). */
46039     if (fs_call == UNLINK) {
46040           /* Only the su may unlink directories, but the su can unlink any dir.*/
46041           if ( (rip->i_mode & I_TYPE) == I_DIRECTORY && !super_user) r = EPERM;
46042
46043           /* Don't unlink a file if it is the root of a mounted file system. */
46044           if (rip->i_num == ROOT_INODE) r = EBUSY;
46045
46046           /* Actually try to unlink the file; fails if parent is mode 0 etc. */
46047           if (r == OK) r = unlink_file(rldirp, rip, string);
46048
46049     } else {
46050           r = remove_dir(rldirp, rip, string); /* call is RMDIR */
46051     }
46052
46053     /* If unlink was possible, it has been done, otherwise it has not. */
46054     put_inode(rip);
46055     put_inode(rldirp);
46056     return(r);
46057   }


46060   /*===========================================================================*
46061    *                                do_rename                                  *
46062    *===========================================================================*/
46063   PUBLIC int do_rename()
46064   {
46065   /* Perform the rename(name1, name2) system call. */
46066
46067     struct inode *old_dirp, *old_ip;      /* ptrs to old dir, file inodes */
46068     struct inode *new_dirp, *new_ip;      /* ptrs to new dir, file inodes */
46069     struct inode *new_superdirp, *next_new_superdirp;
46070     int r = OK;                           /* error flag; initially no error */
46071     int odir, ndir;                       /* TRUE iff {old|new} file is dir */
46072     int same_pdir;                        /* TRUE iff parent dirs are the same */
46073     char old_name[NAME_MAX], new_name[NAME_MAX];
46074     ino_t numb;
```

```
46075      int r1;
46076
46077      /* See if 'name1' (existing file) exists.  Get dir and file inodes. */
46078      if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
46079      if ( (old_dirp = last_dir(user_path, old_name))==NIL_INODE) return(err_code);
46080
46081      if ( (old_ip = advance(old_dirp, old_name)) == NIL_INODE) r = err_code;
46082
46083      /* See if 'name2' (new name) exists.  Get dir and file inodes. */
46084      if (fetch_name(name2, name2_length, M1) != OK) r = err_code;
46085      if ( (new_dirp = last_dir(user_path, new_name)) == NIL_INODE) r = err_code;
46086      new_ip = advance(new_dirp, new_name); /* not required to exist */
46087
46088      if (old_ip != NIL_INODE)
46089            odir = ((old_ip->i_mode & I_TYPE) == I_DIRECTORY);  /* TRUE iff dir */
46090
46091      /* If it is ok, check for a variety of possible errors. */
46092      if (r == OK) {
46093            same_pdir = (old_dirp == new_dirp);
46094
46095            /* The old inode must not be a superdirectory of the new last dir. */
46096            if (odir && !same_pdir) {
46097                    dup_inode(new_superdirp = new_dirp);
46098                    while (TRUE) {          /* may hang in a file system loop */
46099                            if (new_superdirp == old_ip) {
46100                                    r = EINVAL;
46101                                    break;
46102                            }
46103                            next_new_superdirp = advance(new_superdirp, dot2);
46104                            put_inode(new_superdirp);
46105                            if (next_new_superdirp == new_superdirp)
46106                                    break;  /* back at system root directory */
46107                            new_superdirp = next_new_superdirp;
46108                            if (new_superdirp == NIL_INODE) {
46109                                    /* Missing ".." entry.  Assume the worst. */
46110                                    r = EINVAL;
46111                                    break;
46112                            }
46113                    }
46114                    put_inode(new_superdirp);
46115            }
46116
46117            /* The old or new name must not be . or .. */
46118            if (strcmp(old_name, ".")==0 || strcmp(old_name, "..")==0 ||
46119                strcmp(new_name, ".")==0 || strcmp(new_name, "..")==0) r = EINVAL;
46120
46121            /* Both parent directories must be on the same device. */
46122            if (old_dirp->i_dev != new_dirp->i_dev) r = EXDEV;
46123
46124            /* Parent dirs must be writable, searchable and on a writable device */
46125            if ((r1 = forbidden(old_dirp, W_BIT | X_BIT)) != OK ||
46126                (r1 = forbidden(new_dirp, W_BIT | X_BIT)) != OK) r = r1;
46127
46128            /* Some tests apply only if the new path exists. */
46129            if (new_ip == NIL_INODE) {
46130                    /* don't rename a file with a file system mounted on it. */
46131                    if (old_ip->i_dev != old_dirp->i_dev) r = EXDEV;
46132                    if (odir && (new_dirp->i_nlinks & BYTE) >= LINK_MAX &&
46133                        !same_pdir && r == OK) r = EMLINK;
46134            } else {
```

```
46135                   if (old_ip == new_ip) r = SAME; /* old=new */
46136
46137                   /* has the old file or new file a file system mounted on it? */
46138                   if (old_ip->i_dev != new_ip->i_dev) r = EXDEV;
46139
46140                   ndir = ((new_ip->i_mode & I_TYPE) == I_DIRECTORY); /* dir ? */
46141                   if (odir == TRUE && ndir == FALSE) r = ENOTDIR;
46142                   if (odir == FALSE && ndir == TRUE) r = EISDIR;
46143           }
46144       }
46145
46146       /* If a process has another root directory than the system root, we might
46147        * "accidently" be moving it's working directory to a place where it's
46148        * root directory isn't a super directory of it anymore. This can make
46149        * the function chroot useless. If chroot will be used often we should
46150        * probably check for it here.
46151        */
46152
46153       /* The rename will probably work. Only two things can go wrong now:
46154        * 1. being unable to remove the new file. (when new file already exists)
46155        * 2. being unable to make the new directory entry. (new file doesn't exists)
46156        *     [directory has to grow by one block and cannot because the disk
46157        *      is completely full].
46158        */
46159       if (r == OK) {
46160           if (new_ip != NIL_INODE) {
46161                     /* There is already an entry for 'new'. Try to remove it. */
46162                   if (odir)
46163                           r = remove_dir(new_dirp, new_ip, new_name);
46164                   else
46165                           r = unlink_file(new_dirp, new_ip, new_name);
46166           }
46167           /* if r is OK, the rename will succeed, while there is now an
46168            * unused entry in the new parent directory.
46169            */
46170       }
46171
46172       if (r == OK) {
46173           /* If the new name will be in the same parent directory as the old one,
46174            * first remove the old name to free an entry for the new name,
46175            * otherwise first try to create the new name entry to make sure
46176            * the rename will succeed.
46177            */
46178           numb = old_ip->i_num;           /* inode number of old file */
46179
46180           if (same_pdir) {
46181                   r = search_dir(old_dirp, old_name, (ino_t *) 0, DELETE);
46182                                                   /* shouldn't go wrong. */
46183                   if (r==OK) (void) search_dir(old_dirp, new_name, &numb, ENTER);
46184           } else {
46185                   r = search_dir(new_dirp, new_name, &numb, ENTER);
46186                   if (r == OK)
46187                       (void) search_dir(old_dirp, old_name, (ino_t *) 0, DELETE);
46188           }
46189       }
46190       /* If r is OK, the ctime and mtime of old_dirp and new_dirp have been marked
46191        * for update in search_dir.
46192        */
46193
46194       if (r == OK && odir && !same_pdir) {
```

```
46195           /* Update the .. entry in the directory (still points to old_dirp). */
46196           numb = new_dirp->i_num;
46197           (void) unlink_file(old_ip, NIL_INODE, dot2);
46198           if (search_dir(old_ip, dot2, &numb, ENTER) == OK) {
46199                   /* New link created. */
46200                   new_dirp->i_nlinks++;
46201                   new_dirp->i_dirt = DIRTY;
46202           }
46203     }
46204
46205     /* Release the inodes. */
46206     put_inode(old_dirp);
46207     put_inode(old_ip);
46208     put_inode(new_dirp);
46209     put_inode(new_ip);
46210     return(r == SAME ? OK : r);
46211   }


46214   /*===========================================================================*
46215    *                                truncate                                   *
46216    *===========================================================================*/
46217   PUBLIC void truncate(rip)
46218   register struct inode *rip;     /* pointer to inode to be truncated */
46219   {
46220   /* Remove all the zones from the inode 'rip' and mark it dirty. */
46221
46222     register block_t b;
46223     zone_t z, zone_size, z1;
46224     off_t position;
46225     int i, scale, file_type, waspipe, single, nr_indirects;
46226     struct buf *bp;
46227     dev_t dev;
46228
46229     file_type = rip->i_mode & I_TYPE;     /* check to see if file is special */
46230     if (file_type == I_CHAR_SPECIAL || file_type == I_BLOCK_SPECIAL) return;
46231     dev = rip->i_dev;             /* device on which inode resides */
46232     scale = rip->i_sp->s_log_zone_size;
46233     zone_size = (zone_t) BLOCK_SIZE << scale;
46234     nr_indirects = rip->i_nindirs;
46235
46236     /* Pipes can shrink, so adjust size to make sure all zones are removed. */
46237     waspipe = rip->i_pipe == I_PIPE;      /* TRUE is this was a pipe */
46238     if (waspipe) rip->i_size = PIPE_SIZE;
46239
46240     /* Step through the file a zone at a time, finding and freeing the zones. */
46241     for (position = 0; position < rip->i_size; position += zone_size) {
46242           if ( (b = read_map(rip, position)) != NO_BLOCK) {
46243                   z = (zone_t) b >> scale;
46244                   free_zone(dev, z);
46245           }
46246     }
46247
46248     /* All the data zones have been freed.  Now free the indirect zones. */
46249     rip->i_dirt = DIRTY;
46250     if (waspipe) {
46251           wipe_inode(rip);        /* clear out inode for pipes */
46252           return;                 /* indirect slots contain file positions */
46253     }
46254     single = rip->i_ndzones;
```

```
46255     free_zone(dev, rip->i_zone[single]);  /* single indirect zone */
46256     if ( (z = rip->i_zone[single+1]) != NO_ZONE) {
46257           /* Free all the single indirect zones pointed to by the double. */
46258           b = (block_t) z << scale;
46259           bp = get_block(dev, b, NORMAL); /* get double indirect zone */
46260           for (i = 0; i < nr_indirects; i++) {
46261                   z1 = rd_indir(bp, i);
46262                   free_zone(dev, z1);
46263           }
46264
46265           /* Now free the double indirect zone itself. */
46266           put_block(bp, INDIRECT_BLOCK);
46267           free_zone(dev, z);
46268     }
46269
46270     /* Leave zone numbers for de(1) to recover file after an unlink(2).  */
46271   }


46274   /*===========================================================================*
46275    *                                remove_dir                                 *
46276    *===========================================================================*/
46277   PRIVATE int remove_dir(rldirp, rip, dir_name)
46278   struct inode *rldirp;                   /* parent directory */
46279   struct inode *rip;                      /* directory to be removed */
46280   char dir_name[NAME_MAX];                /* name of directory to be removed */
46281   {
46282     /* A directory file has to be removed. Five conditions have to met:
46283      *    - The file must be a directory
46284      *    - The directory must be empty (except for . and ..)
46285      *    - The final component of the path must not be . or ..
46286      *    - The directory must not be the root of a mounted file system
46287      *    - The directory must not be anybody's root/working directory
46288      */
46289
46290     int r;
46291     register struct fproc *rfp;
46292
46293     /* search_dir checks that rip is a directory too. */
46294     if ((r = search_dir(rip, "", (ino_t *) 0, IS_EMPTY)) != OK) return r;
46295
46296     if (strcmp(dir_name, ".") == 0 || strcmp(dir_name, "..") == 0)return(EINVAL);
46297     if (rip->i_num == ROOT_INODE) return(EBUSY); /* can't remove 'root' */
46298
46299     for (rfp = &fproc[INIT_PROC_NR + 1]; rfp < &fproc[NR_PROCS]; rfp++)
46300           if (rfp->fp_workdir == rip || rfp->fp_rootdir == rip) return(EBUSY);
46301                                   /* can't remove anybody's working dir */
46302
46303     /* Actually try to unlink the file; fails if parent is mode 0 etc. */
46304     if ((r = unlink_file(rldirp, rip, dir_name)) != OK) return r;
46305
46306     /* Unlink . and .. from the dir. The super user can link and unlink any dir,
46307      * so don't make too many assumptions about them.
46308      */
46309     (void) unlink_file(rip, NIL_INODE, dot1);
46310     (void) unlink_file(rip, NIL_INODE, dot2);
46311     return(OK);
46312   }
```

```
46315    /*===========================================================================*
46316     *                                unlink_file                                *
46317     *===========================================================================*/
46318    PRIVATE int unlink_file(dirp, rip, file_name)
46319    struct inode *dirp;             /* parent directory of file */
46320    struct inode *rip;              /* inode of file, may be NIL_INODE too. */
46321    char file_name[NAME_MAX];       /* name of file to be removed */
46322    {
46323    /* Unlink 'file_name'; rip must be the inode of 'file_name' or NIL_INODE. */
46324
46325      ino_t numb;                   /* inode number */
46326      int   r;
46327
46328      /* If rip is not NIL_INODE, it is used to get faster access to the inode. */
46329      if (rip == NIL_INODE) {
46330            /* Search for file in directory and try to get its inode. */
46331            err_code = search_dir(dirp, file_name, &numb, LOOK_UP);
46332            if (err_code == OK) rip = get_inode(dirp->i_dev, (int) numb);
46333            if (err_code != OK || rip == NIL_INODE) return(err_code);
46334      } else {
46335            dup_inode(rip);         /* inode will be returned with put_inode */
46336      }
46337
46338      r = search_dir(dirp, file_name, (ino_t *) 0, DELETE);
46339
46340      if (r == OK) {
46341            rip->i_nlinks--;        /* entry deleted from parent's dir */
46342            rip->i_update |= CTIME;
46343            rip->i_dirt = DIRTY;
46344      }
46345
46346      put_inode(rip);
46347      return(r);
46348    }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                          src/fs/lock.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
46400    /* This file handles advisory file locking as required by POSIX.
46401     *
46402     * The entry points into this file are
46403     *   lock_op:   perform locking operations for FCNTL system call
46404     *   lock_revive: revive processes when a lock is released
46405     */
46406
46407    #include "fs.h"
46408    #include <fcntl.h>
46409    #include <unistd.h>
46410    #include "file.h"
46411    #include "fproc.h"
46412    #include "inode.h"
46413    #include "lock.h"
46414    #include "param.h"
46415
46416    /*===========================================================================*
46417     *                                lock_op                                    *
46418     *===========================================================================*/
46419    PUBLIC int lock_op(f, req)
```

```
46420   struct filp *f;
46421   int req;                        /* either F_SETLK or F_SETLKW */
46422   {
46423   /* Perform the advisory locking required by POSIX. */
46424
46425     int r, ltype, i, conflict = 0, unlocking = 0;
46426     mode_t mo;
46427     off_t first, last;
46428     struct flock flock;
46429     vir_bytes user_flock;
46430     struct file_lock *flp, *flp2, *empty;
46431
46432     /* Fetch the flock structure from user space. */
46433     user_flock = (vir_bytes) name1;
46434     r = sys_copy(who, D, (phys_bytes) user_flock,
46435           FS_PROC_NR, D, (phys_bytes) &flock, (phys_bytes) sizeof(flock));
46436     if (r != OK) return(EINVAL);
46437
46438     /* Make some error checks. */
46439     ltype = flock.l_type;
46440     mo = f->filp_mode;
46441     if (ltype != F_UNLCK && ltype != F_RDLCK && ltype != F_WRLCK) return(EINVAL);
46442     if (req == F_GETLK && ltype == F_UNLCK) return(EINVAL);
46443     if ( (f->filp_ino->i_mode & I_TYPE) != I_REGULAR) return(EINVAL);
46444     if (req != F_GETLK && ltype == F_RDLCK && (mo & R_BIT) == 0) return(EBADF);
46445     if (req != F_GETLK && ltype == F_WRLCK && (mo & W_BIT) == 0) return(EBADF);
46446
46447     /* Compute the first and last bytes in the lock region. */
46448     switch (flock.l_whence) {
46449           case SEEK_SET:  first = 0; break;
46450           case SEEK_CUR:  first = f->filp_pos; break;
46451           case SEEK_END:  first = f->filp_ino->i_size; break;
46452           default:        return(EINVAL);
46453     }
46454     /* Check for overflow. */
46455     if (((long)flock.l_start > 0) && ((first + flock.l_start) < first))
46456           return(EINVAL);
46457     if (((long)flock.l_start < 0) && ((first + flock.l_start) > first))
46458           return(EINVAL);
46459     first = first + flock.l_start;
46460     last = first + flock.l_len - 1;
46461     if (flock.l_len == 0) last = MAX_FILE_POS;
46462     if (last < first) return(EINVAL);
46463
46464     /* Check if this region conflicts with any existing lock. */
46465     empty = (struct file_lock *) 0;
46466     for (flp = &file_lock[0]; flp < & file_lock[NR_LOCKS]; flp++) {
46467           if (flp->lock_type == 0) {
46468                 if (empty == (struct file_lock *) 0) empty = flp;
46469                 continue;       /* 0 means unused slot */
46470           }
46471           if (flp->lock_inode != f->filp_ino) continue;   /* different file */
46472           if (last < flp->lock_first) continue;   /* new one is in front */
46473           if (first > flp->lock_last) continue;   /* new one is afterwards */
46474           if (ltype == F_RDLCK && flp->lock_type == F_RDLCK) continue;
46475           if (ltype != F_UNLCK && flp->lock_pid == fp->fp_pid) continue;
46476
46477           /* There might be a conflict.  Process it. */
46478           conflict = 1;
46479           if (req == F_GETLK) break;
```

```
46480
46481           /* If we are trying to set a lock, it just failed. */
46482           if (ltype == F_RDLCK || ltype == F_WRLCK) {
46483                   if (req == F_SETLK) {
46484                           /* For F_SETLK, just report back failure. */
46485                           return(EAGAIN);
46486                   } else {
46487                           /* For F_SETLKW, suspend the process. */
46488                           suspend(XLOCK);
46489                           return(0);
46490                   }
46491           }
46492
46493           /* We are clearing a lock and we found something that overlaps. */
46494           unlocking = 1;
46495           if (first <= flp->lock_first && last >= flp->lock_last) {
46496                   flp->lock_type = 0;     /* mark slot as unused */
46497                   nr_locks--;             /* number of locks is now 1 less */
46498                   continue;
46499           }
46500
46501           /* Part of a locked region has been unlocked. */
46502           if (first <= flp->lock_first) {
46503                   flp->lock_first = last + 1;
46504                   continue;
46505           }
46506
46507           if (last >= flp->lock_last) {
46508                   flp->lock_last = first - 1;
46509                   continue;
46510           }
46511
46512           /* Bad luck. A lock has been split in two by unlocking the middle. */
46513           if (nr_locks == NR_LOCKS) return(ENOLCK);
46514           for (i = 0; i < NR_LOCKS; i++)
46515                   if (file_lock[i].lock_type == 0) break;
46516           flp2 = &file_lock[i];
46517           flp2->lock_type = flp->lock_type;
46518           flp2->lock_pid = flp->lock_pid;
46519           flp2->lock_inode = flp->lock_inode;
46520           flp2->lock_first = last + 1;
46521           flp2->lock_last = flp->lock_last;
46522           flp->lock_last = first - 1;
46523           nr_locks++;
46524      }
46525      if (unlocking) lock_revive();
46526
46527      if (req == F_GETLK) {
46528           if (conflict) {
46529                   /* GETLK and conflict. Report on the conflicting lock. */
46530                   flock.l_type = flp->lock_type;
46531                   flock.l_whence = SEEK_SET;
46532                   flock.l_start = flp->lock_first;
46533                   flock.l_len = flp->lock_last - flp->lock_first + 1;
46534                   flock.l_pid = flp->lock_pid;
46535
46536           } else {
46537                   /* It is GETLK and there is no conflict. */
46538                   flock.l_type = F_UNLCK;
46539           }
```

```
46540
46541           /* Copy the flock structure back to the caller. */
46542           r = sys_copy(FS_PROC_NR, D, (phys_bytes) &flock,
46543                   who, D, (phys_bytes) user_flock, (phys_bytes) sizeof(flock));
46544           return(r);
46545      }
46546
46547      if (ltype == F_UNLCK) return(OK);     /* unlocked a region with no locks */
46548
46549      /* There is no conflict.  If space exists, store new lock in the table. */
46550      if (empty == (struct file_lock *) 0) return(ENOLCK);  /* table full */
46551      empty->lock_type = ltype;
46552      empty->lock_pid = fp->fp_pid;
46553      empty->lock_inode = f->filp_ino;
46554      empty->lock_first = first;
46555      empty->lock_last = last;
46556      nr_locks++;
46557      return(OK);
46558    }

46560    /*===========================================================================*
46561     *                              lock_revive                                  *
46562     *===========================================================================*/
46563    PUBLIC void lock_revive()
46564    {
46565    /* Go find all the processes that are waiting for any kind of lock and
46566     * revive them all.  The ones that are still blocked will block again when
46567     * they run.  The others will complete.  This strategy is a space-time
46568     * tradeoff.  Figuring out exactly which ones to unblock now would take
46569     * extra code, and the only thing it would win would be some performance in
46570     * extremely rare circumstances (namely, that somebody actually used
46571     * locking).
46572     */
46573
46574      int task;
46575      struct fproc *fptr;
46576
46577      for (fptr = &fproc[INIT_PROC_NR + 1]; fptr < &fproc[NR_PROCS]; fptr++){
46578           task = -fptr->fp_task;
46579           if (fptr->fp_suspended == SUSPENDED && task == XLOCK) {
46580                   revive( (int) (fptr - fproc), 0);
46581           }
46582      }
46583    }


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/main.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

46600    /* This file contains the main program of the File System.  It consists of
46601     * a loop that gets messages requesting work, carries out the work, and sends
46602     * replies.
46603     *
46604     * The entry points into this file are
46605     *   main:      main program of the File System
46606     *   reply:     send a reply to a process after the requested work is done
46607     */
46608
46609    struct super_block;             /* proto.h needs to know this */
```

```
46610
46611   #include "fs.h"
46612   #include <fcntl.h>
46613   #include <string.h>
46614   #include <sys/ioctl.h>
46615   #include <minix/callnr.h>
46616   #include <minix/com.h>
46617   #include <minix/boot.h>
46618   #include "buf.h"
46619   #include "dev.h"
46620   #include "file.h"
46621   #include "fproc.h"
46622   #include "inode.h"
46623   #include "param.h"
46624   #include "super.h"
46625
46626   FORWARD _PROTOTYPE( void buf_pool, (void)                               );
46627   FORWARD _PROTOTYPE( void fs_init, (void)                                );
46628   FORWARD _PROTOTYPE( void get_boot_parameters, (void)                    );
46629   FORWARD _PROTOTYPE( void get_work, (void)                               );
46630   FORWARD _PROTOTYPE( void load_ram, (void)                               );
46631   FORWARD _PROTOTYPE( void load_super, (Dev_t super_dev)                  );
46632
46633
46634   /*===========================================================================*
46635    *                              main                                         *
46636    *===========================================================================*/
46637   PUBLIC void main()
46638   {
46639   /* This is the main program of the file system.  The main loop consists of
46640    * three major activities: getting new work, processing the work, and sending
46641    * the reply.  This loop never terminates as long as the file system runs.
46642    */
46643     int error;
46644
46645     fs_init();
46646
46647     /* This is the main loop that gets work, processes it, and sends replies. */
46648     while (TRUE) {
46649           get_work();             /* sets who and fs_call */
46650
46651           fp = &fproc[who];       /* pointer to proc table struct */
46652           super_user = (fp->fp_effuid == SU_UID ? TRUE : FALSE);   /* su? */
46653           dont_reply = FALSE;     /* in other words, do reply is default */
46654
46655           /* Call the internal function that does the work. */
46656           if (fs_call < 0 || fs_call >= NCALLS)
46657                   error = EBADCALL;
46658           else
46659                   error = (*call_vector[fs_call])();
46660
46661           /* Copy the results back to the user and send reply. */
46662           if (dont_reply) continue;
46663           reply(who, error);
46664           if (rdahed_inode != NIL_INODE) read_ahead(); /* do block read ahead */
46665     }
46666   }


46669   /*===========================================================================*
```

```
46670     *                           get_work                                  *
46671     *=====================================================================*/
46672    PRIVATE void get_work()
46673    {
46674      /* Normally wait for new input.  However, if 'reviving' is
46675       * nonzero, a suspended process must be awakened.
46676       */
46677
46678      register struct fproc *rp;
46679
46680      if (reviving != 0) {
46681            /* Revive a suspended process. */
46682            for (rp = &fproc[0]; rp < &fproc[NR_PROCS]; rp++)
46683                    if (rp->fp_revived == REVIVING) {
46684                            who = (int)(rp - fproc);
46685                            fs_call = rp->fp_fd & BYTE;
46686                            fd = (rp->fp_fd >>8) & BYTE;
46687                            buffer = rp->fp_buffer;
46688                            nbytes = rp->fp_nbytes;
46689                            rp->fp_suspended = NOT_SUSPENDED; /*no longer hanging*/
46690                            rp->fp_revived = NOT_REVIVING;
46691                            reviving--;
46692                            return;
46693                    }
46694            panic("get_work couldn't revive anyone", NO_NUM);
46695      }
46696
46697      /* Normal case.  No one to revive. */
46698      if (receive(ANY, &m) != OK) panic("fs receive error", NO_NUM);
46699
46700      who = m.m_source;
46701      fs_call = m.m_type;
46702    }


46705    /*=====================================================================*
46706     *                           reply                                     *
46707     *=====================================================================*/
46708    PUBLIC void reply(whom, result)
46709    int whom;                       /* process to reply to */
46710    int result;                     /* result of the call (usually OK or error #) */
46711    {
46712    /* Send a reply to a user process. It may fail (if the process has just
46713     * been killed by a signal), so don't check the return code.  If the send
46714     * fails, just ignore it.
46715     */
46716
46717      reply_type = result;
46718      send(whom, &m1);
46719    }


46722    /*=====================================================================*
46723     *                           fs_init                                   *
46724     *=====================================================================*/
46725    PRIVATE void fs_init()
46726    {
46727    /* Initialize global variables, tables, etc. */
46728
46729      register struct inode *rip;
```

```
46730     int i;
46731     message mess;
46732
46733     /* The following initializations are needed to let dev_opcl succeed .*/
46734     fp = (struct fproc *) NULL;
46735     who = FS_PROC_NR;
46736
46737     buf_pool();                   /* initialize buffer pool */
46738     get_boot_parameters();        /* get the parameters from the menu */
46739     load_ram();                   /* init RAM disk, load if it is root */
46740     load_super(ROOT_DEV);         /* load super block for root device */
46741
46742     /* Initialize the 'fproc' fields for process 0 .. INIT. */
46743     for (i = 0; i <= LOW_USER; i+= 1) {
46744           if (i == FS_PROC_NR) continue;  /* do not initialize FS */
46745           fp = &fproc[i];
46746           rip = get_inode(ROOT_DEV, ROOT_INODE);
46747           fp->fp_rootdir = rip;
46748           dup_inode(rip);
46749           fp->fp_workdir = rip;
46750           fp->fp_realuid = (uid_t) SYS_UID;
46751           fp->fp_effuid = (uid_t) SYS_UID;
46752           fp->fp_realgid = (gid_t) SYS_GID;
46753           fp->fp_effgid = (gid_t) SYS_GID;
46754           fp->fp_umask = ~0;
46755     }
46756
46757     /* Certain relations must hold for the file system to work at all. */
46758     if (SUPER_SIZE > BLOCK_SIZE) panic("SUPER_SIZE > BLOCK_SIZE", NO_NUM);
46759     if (BLOCK_SIZE % V2_INODE_SIZE != 0)  /* this checks V1_INODE_SIZE too */
46760           panic("BLOCK_SIZE % V2_INODE_SIZE != 0", NO_NUM);
46761     if (OPEN_MAX > 127) panic("OPEN_MAX > 127", NO_NUM);
46762     if (NR_BUFS < 6) panic("NR_BUFS < 6", NO_NUM);
46763     if (V1_INODE_SIZE != 32) panic("V1 inode size != 32", NO_NUM);
46764     if (V2_INODE_SIZE != 64) panic("V2 inode size != 64", NO_NUM);
46765     if (OPEN_MAX > 8 * sizeof(long)) panic("Too few bits in fp_cloexec", NO_NUM);
46766
46767     /* Tell the memory task where my process table is for the sake of ps(1). */
46768     mess.m_type = DEV_IOCTL;
46769     mess.PROC_NR = FS_PROC_NR;
46770     mess.REQUEST = MIOCSPSINFO;
46771     mess.ADDRESS = (void *) fproc;
46772     (void) sendrec(MEM, &mess);
46773   }


46776   /*===========================================================================*
46777    *                              buf_pool                                     *
46778    *===========================================================================*/
46779   PRIVATE void buf_pool()
46780   {
46781   /* Initialize the buffer pool. */
46782
46783     register struct buf *bp;
46784
46785     bufs_in_use = 0;
46786     front = &buf[0];
46787     rear = &buf[NR_BUFS - 1];
46788
46789     for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++) {
```

```
46790           bp->b_blocknr = NO_BLOCK;
46791           bp->b_dev = NO_DEV;
46792           bp->b_next = bp + 1;
46793           bp->b_prev = bp - 1;
46794     }
46795     buf[0].b_prev = NIL_BUF;
46796     buf[NR_BUFS - 1].b_next = NIL_BUF;
46797
46798     for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++) bp->b_hash = bp->b_next;
46799     buf_hash[0] = front;
46800   }


46803   /*===========================================================================*
46804    *                              get_boot_parameters                          *
46805    *===========================================================================*/
46806   PUBLIC struct bparam_s boot_parameters;
46807
46808   PRIVATE void get_boot_parameters()
46809   {
46810   /* Ask kernel for boot parameters. */
46811
46812     m1.m_type = SYS_GBOOT;
46813     m1.PROC1 = FS_PROC_NR;
46814     m1.MEM_PTR = (char *) &boot_parameters;
46815     (void) sendrec(SYSTASK, &m1);
46816   }


46819   /*===========================================================================*
46820    *                              load_ram                                     *
46821    *===========================================================================*/
46822   PRIVATE void load_ram()
46823   {
46824   /* If the root device is the RAM disk, copy the entire root image device
46825    * block-by-block to a RAM disk with the same size as the image.
46826    * Otherwise, just allocate a RAM disk with size given in the boot parameters.
46827    */
46828
46829     register struct buf *bp, *bp1;
46830     long k_loaded, lcount;
46831     u32_t ram_size, fsmax;
46832     zone_t zones;
46833     struct super_block *sp, *dsp;
46834     block_t b;
46835     int major, task;
46836     message dev_mess;
46837
46838     ram_size = boot_parameters.bp_ramsize;
46839
46840     /* Open the root device. */
46841     major = (ROOT_DEV >> MAJOR) & BYTE;   /* major device nr */
46842     task = dmap[major].dmap_task;         /* device task nr */
46843     dev_mess.m_type = DEV_OPEN;           /* distinguish from close */
46844     dev_mess.DEVICE = ROOT_DEV;
46845     dev_mess.COUNT = R_BIT|W_BIT;
46846     (*dmap[major].dmap_open)(task, &dev_mess);
46847     if (dev_mess.REP_STATUS != OK) panic("Cannot open root device",NO_NUM);
46848
46849     /* If the root device is the ram disk then fill it from the image device. */
```

```
46850      if (ROOT_DEV == DEV_RAM) {
46851            major = (IMAGE_DEV >> MAJOR) & BYTE;    /* major device nr */
46852            task = dmap[major].dmap_task;           /* device task nr */
46853            dev_mess.m_type = DEV_OPEN;             /* distinguish from close */
46854            dev_mess.DEVICE = IMAGE_DEV;
46855            dev_mess.COUNT = R_BIT;
46856            (*dmap[major].dmap_open)(task, &dev_mess);
46857            if (dev_mess.REP_STATUS != OK) panic("Cannot open root device", NO_NUM);
46858
46859            /* Get size of RAM disk by reading root file system's super block. */
46860            sp = &super_block[0];
46861            sp->s_dev = IMAGE_DEV;
46862            if (read_super(sp) != OK) panic("Bad root file system", NO_NUM);
46863
46864            lcount = sp->s_zones << sp->s_log_zone_size;    /* # blks on root dev*/
46865
46866            /* Stretch the RAM disk file system to the boot parameters size, but
46867             * no further than the last zone bit map block allows.
46868             */
46869            if (ram_size < lcount) ram_size = lcount;
46870            fsmax = (u32_t) sp->s_zmap_blocks * CHAR_BIT * BLOCK_SIZE;
46871            fsmax = (fsmax + (sp->s_firstdatazone-1)) << sp->s_log_zone_size;
46872            if (ram_size > fsmax) ram_size = fsmax;
46873      }
46874
46875      /* Tell RAM driver how big the RAM disk must be. */
46876      m1.m_type = DEV_IOCTL;
46877      m1.PROC_NR = FS_PROC_NR;
46878      m1.REQUEST = MIOCRAMSIZE;
46879      m1.POSITION = ram_size;
46880      if (sendrec(MEM, &m1) != OK || m1.REP_STATUS != OK)
46881            panic("Can't set RAM disk size", NO_NUM);
46882
46883      /* Tell MM the RAM disk size, and wait for it to come "on-line". */
46884      m1.m1_i1 = ((long) ram_size * BLOCK_SIZE) >> CLICK_SHIFT;
46885      if (sendrec(MM_PROC_NR, &m1) != OK)
46886            panic("FS can't sync up with MM", NO_NUM);
46887
46888    #if ENABLE_CACHE2
46889      /* The RAM disk is a second level block cache while not otherwise used. */
46890      init_cache2(ram_size);
46891    #endif
46892
46893      /* If the root device is not the RAM disk, it doesn't need loading. */
46894      if (ROOT_DEV != DEV_RAM) return;
46895
46896      /* Copy the blocks one at a time from the image to the RAM disk. */
46897      printf("Loading RAM disk.\33[23CLoaded:    0K ");
46898
46899      inode[0].i_mode = I_BLOCK_SPECIAL;    /* temp inode for rahead() */
46900      inode[0].i_size = LONG_MAX;
46901      inode[0].i_dev = IMAGE_DEV;
46902      inode[0].i_zone[0] = IMAGE_DEV;
46903
46904      for (b = 0; b < (block_t) lcount; b++) {
46905            bp = rahead(&inode[0], b, (off_t)BLOCK_SIZE * b, BLOCK_SIZE);
46906            bp1 = get_block(ROOT_DEV, b, NO_READ);
46907            memcpy(bp1->b_data, bp->b_data, (size_t) BLOCK_SIZE);
46908            bp1->b_dirt = DIRTY;
46909            put_block(bp, FULL_DATA_BLOCK);
```

```
46910           put_block(bp1, FULL_DATA_BLOCK);
46911           k_loaded = ( (long) b * BLOCK_SIZE)/1024L;      /* K loaded so far */
46912           if (k_loaded % 5 == 0) printf("\b\b\b\b\b\b%5ldK ", k_loaded);
46913     }
46914
46915     printf("\rRAM disk loaded.\33[K\n\n");
46916
46917     /* Close and invalidate image device. */
46918     dev_mess.m_type = DEV_CLOSE;
46919     dev_mess.DEVICE = IMAGE_DEV;
46920     (*dmap[major].dmap_close)(task, &dev_mess);
46921     invalidate(IMAGE_DEV);
46922
46923     /* Resize the RAM disk root file system. */
46924     bp = get_block(ROOT_DEV, SUPER_BLOCK, NORMAL);
46925     dsp = (struct super_block *) bp->b_data;
46926     zones = ram_size >> sp->s_log_zone_size;
46927     dsp->s_nzones = conv2(sp->s_native, (u16_t) zones);
46928     dsp->s_zones = conv4(sp->s_native, zones);
46929     bp->b_dirt = DIRTY;
46930     put_block(bp, ZUPER_BLOCK);
46931   }


46934   /*===========================================================================*
46935    *                              load_super                                   *
46936    *===========================================================================*/
46937   PRIVATE void load_super(super_dev)
46938   dev_t super_dev;                        /* place to get superblock from */
46939   {
46940     int bad;
46941     register struct super_block *sp;
46942     register struct inode *rip;
46943
46944     /* Initialize the super_block table. */
46945     for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
46946           sp->s_dev = NO_DEV;
46947
46948     /* Read in super_block for the root file system. */
46949     sp = &super_block[0];
46950     sp->s_dev = super_dev;
46951
46952     /* Check super_block for consistency (is it the right diskette?). */
46953     bad = (read_super(sp) != OK);
46954     if (!bad) {
46955           rip = get_inode(super_dev, ROOT_INODE); /* inode for root dir */
46956           if ( (rip->i_mode & I_TYPE) != I_DIRECTORY || rip->i_nlinks < 3) bad++;
46957     }
46958     if (bad)panic("Invalid root file system.  Possibly wrong diskette.",NO_NUM);
46959
46960     sp->s_imount = rip;
46961     dup_inode(rip);
46962     sp->s_isup = rip;
46963     sp->s_rd_only = 0;
46964     return;
46965   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/misc.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

47000    /* This file contains a collection of miscellaneous procedures.  Some of them
47001     * perform simple system calls.  Some others do a little part of system calls
47002     * that are mostly performed by the Memory Manager.
47003     *
47004     * The entry points into this file are
47005     *   do_dup:      perform the DUP system call
47006     *   do_fcntl:    perform the FCNTL system call
47007     *   do_sync:     perform the SYNC system call
47008     *   do_fork:     adjust the tables after MM has performed a FORK system call
47009     *   do_exec:     handle files with FD_CLOEXEC on after MM has done an EXEC
47010     *   do_exit:     a process has exited; note that in the tables
47011     *   do_set:      set uid or gid for some process
47012     *   do_revive:   revive a process that was waiting for something (e.g. TTY)
47013     */
47014
47015    #include "fs.h"
47016    #include <fcntl.h>
47017    #include <unistd.h>
47018    #include <minix/callnr.h>
47019    #include <minix/com.h>
47020    #include <minix/boot.h>
47021    #include "buf.h"
47022    #include "file.h"
47023    #include "fproc.h"
47024    #include "inode.h"
47025    #include "dev.h"
47026    #include "param.h"
47027
47028
47029    /*===========================================================================*
47030     *                                do_dup                                     *
47031     *===========================================================================*/
47032    PUBLIC int do_dup()
47033    {
47034    /* Perform the dup(fd) or dup(fd,fd2) system call. These system calls are
47035     * obsolete.  In fact, it is not even possible to invoke them using the
47036     * current library because the library routines call fcntl().  They are
47037     * provided to permit old binary programs to continue to run.
47038     */
47039
47040      register int rfd;
47041      register struct filp *f;
47042      struct filp *dummy;
47043      int r;
47044
47045      /* Is the file descriptor valid? */
47046      rfd = fd & ~DUP_MASK;         /* kill off dup2 bit, if on */
47047      if ((f = get_filp(rfd)) == NIL_FILP) return(err_code);
47048
47049      /* Distinguish between dup and dup2. */
47050      if (fd == rfd) {                      /* bit not on */
47051            /* dup(fd) */
47052            if ( (r = get_fd(0, 0, &fd2, &dummy)) != OK) return(r);
47053      } else {
47054            /* dup2(fd, fd2) */
```

```
47055           if (fd2 < 0 || fd2 >= OPEN_MAX) return(EBADF);
47056           if (rfd == fd2) return(fd2);    /* ignore the call: dup2(x, x) */
47057           fd = fd2;               /* prepare to close fd2 */
47058           (void) do_close();      /* cannot fail */
47059       }
47060
47061       /* Success. Set up new file descriptors. */
47062       f->filp_count++;
47063       fp->fp_filp[fd2] = f;
47064       return(fd2);
47065   }

47067   /*===========================================================================*
47068    *                              do_fcntl                                     *
47069    *===========================================================================*/
47070   PUBLIC int do_fcntl()
47071   {
47072   /* Perform the fcntl(fd, request, ...) system call. */
47073
47074       register struct filp *f;
47075       int new_fd, r, fl;
47076       long cloexec_mask;            /* bit map for the FD_CLOEXEC flag */
47077       long clo_value;               /* FD_CLOEXEC flag in proper position */
47078       struct filp *dummy;
47079
47080       /* Is the file descriptor valid? */
47081       if ((f = get_filp(fd)) == NIL_FILP) return(err_code);
47082
47083       switch (request) {
47084           case F_DUPFD:
47085               /* This replaces the old dup() system call. */
47086               if (addr < 0 || addr >= OPEN_MAX) return(EINVAL);
47087               if ((r = get_fd(addr, 0, &new_fd, &dummy)) != OK) return(r);
47088               f->filp_count++;
47089               fp->fp_filp[new_fd] = f;
47090               return(new_fd);
47091
47092           case F_GETFD:
47093               /* Get close-on-exec flag (FD_CLOEXEC in POSIX Table 6-2). */
47094               return( ((fp->fp_cloexec >> fd) & 01) ? FD_CLOEXEC : 0);
47095
47096           case F_SETFD:
47097               /* Set close-on-exec flag (FD_CLOEXEC in POSIX Table 6-2). */
47098               cloexec_mask = 1L << fd;        /* singleton set position ok */
47099               clo_value = (addr & FD_CLOEXEC ? cloexec_mask : 0L);
47100               fp->fp_cloexec = (fp->fp_cloexec & ~cloexec_mask) | clo_value;
47101               return(OK);
47102
47103           case F_GETFL:
47104               /* Get file status flags (O_NONBLOCK and O_APPEND). */
47105               fl = f->filp_flags & (O_NONBLOCK | O_APPEND | O_ACCMODE);
47106               return(fl);
47107
47108           case F_SETFL:
47109               /* Set file status flags (O_NONBLOCK and O_APPEND). */
47110               fl = O_NONBLOCK | O_APPEND;
47111               f->filp_flags = (f->filp_flags & ~fl) | (addr & fl);
47112               return(OK);
47113
47114           case F_GETLK:
```

```
47115         case F_SETLK:
47116         case F_SETLKW:
47117            /* Set or clear a file lock. */
47118            r = lock_op(f, request);
47119            return(r);
47120
47121         default:
47122            return(EINVAL);
47123       }
47124     }


47127     /*===========================================================================*
47128      *                                do_sync                                    *
47129      *===========================================================================*/
47130     PUBLIC int do_sync()
47131     {
47132     /* Perform the sync() system call.  Flush all the tables. */
47133
47134       register struct inode *rip;
47135       register struct buf *bp;
47136
47137       /* The order in which the various tables are flushed is critical.  The
47138        * blocks must be flushed last, since rw_inode() leaves its results in
47139        * the block cache.
47140        */
47141
47142       /* Write all the dirty inodes to the disk. */
47143       for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++)
47144             if (rip->i_count > 0 && rip->i_dirt == DIRTY) rw_inode(rip, WRITING);
47145
47146       /* Write all the dirty blocks to the disk, one drive at a time. */
47147       for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++)
47148             if (bp->b_dev != NO_DEV && bp->b_dirt == DIRTY) flushall(bp->b_dev);
47149
47150       return(OK);           /* sync() can't fail */
47151     }


47154     /*===========================================================================*
47155      *                                do_fork                                    *
47156      *===========================================================================*/
47157     PUBLIC int do_fork()
47158     {
47159     /* Perform those aspects of the fork() system call that relate to files.
47160      * In particular, let the child inherit its parent's file descriptors.
47161      * The parent and child parameters tell who forked off whom. The file
47162      * system uses the same slot numbers as the kernel.  Only MM makes this call.
47163      */
47164
47165       register struct fproc *cp;
47166       int i;
47167
47168       /* Only MM may make this call directly. */
47169       if (who != MM_PROC_NR) return(EGENERIC);
47170
47171       /* Copy the parent's fproc struct to the child. */
47172       fproc[child] = fproc[parent];
47173
47174       /* Increase the counters in the 'filp' table. */
```

```
47175     cp = &fproc[child];
47176     for (i = 0; i < OPEN_MAX; i++)
47177           if (cp->fp_filp[i] != NIL_FILP) cp->fp_filp[i]->filp_count++;
47178
47179     /* Fill in new process id. */
47180     cp->fp_pid = pid;
47181
47182     /* A child is not a process leader. */
47183     cp->fp_sesldr = 0;
47184
47185     /* Record the fact that both root and working dir have another user. */
47186     dup_inode(cp->fp_rootdir);
47187     dup_inode(cp->fp_workdir);
47188     return(OK);
47189   }


47192   /*===========================================================================*
47193    *                                do_exec                                    *
47194    *===========================================================================*/
47195   PUBLIC int do_exec()
47196   {
47197   /* Files can be marked with the FD_CLOEXEC bit (in fp->fp_cloexec).  When
47198    * MM does an EXEC, it calls FS to allow FS to find these files and close them.
47199    */
47200
47201     register int i;
47202     long bitmap;
47203
47204     /* Only MM may make this call directly. */
47205     if (who != MM_PROC_NR) return(EGENERIC);
47206
47207     /* The array of FD_CLOEXEC bits is in the fp_cloexec bit map. */
47208     fp = &fproc[slot1];           /* get_filp() needs 'fp' */
47209     bitmap = fp->fp_cloexec;
47210     if (bitmap == 0) return(OK);  /* normal case, no FD_CLOEXECs */
47211
47212     /* Check the file desriptors one by one for presence of FD_CLOEXEC. */
47213     for (i = 0; i < OPEN_MAX; i++) {
47214           fd = i;
47215           if ( (bitmap >> i) & 01) (void) do_close();
47216     }
47217
47218     return(OK);
47219   }


47222   /*===========================================================================*
47223    *                                do_exit                                    *
47224    *===========================================================================*/
47225   PUBLIC int do_exit()
47226   {
47227   /* Perform the file system portion of the exit(status) system call. */
47228
47229     register int i, exitee, task;
47230     register struct fproc *rfp;
47231     register struct filp *rfilp;
47232     register struct inode *rip;
47233     int major;
47234     dev_t dev;
```

```
47235     message dev_mess;
47236
47237     /* Only MM may do the EXIT call directly. */
47238     if (who != MM_PROC_NR) return(EGENERIC);
47239
47240     /* Nevertheless, pretend that the call came from the user. */
47241     fp = &fproc[slot1];           /* get_filp() needs 'fp' */
47242     exitee = slot1;
47243
47244     if (fp->fp_suspended == SUSPENDED) {
47245           task = -fp->fp_task;
47246           if (task == XPIPE || task == XPOPEN) susp_count--;
47247           pro = exitee;
47248           (void) do_unpause();    /* this always succeeds for MM */
47249           fp->fp_suspended = NOT_SUSPENDED;
47250     }
47251
47252     /* Loop on file descriptors, closing any that are open. */
47253     for (i = 0; i < OPEN_MAX; i++) {
47254           fd = i;
47255           (void) do_close();
47256     }
47257
47258     /* Release root and working directories. */
47259     put_inode(fp->fp_rootdir);
47260     put_inode(fp->fp_workdir);
47261     fp->fp_rootdir = NIL_INODE;
47262     fp->fp_workdir = NIL_INODE;
47263
47264     /* If a session leader exits then revoke access to its controlling tty from
47265      * all other processes using it.
47266      */
47267     if (!fp->fp_sesldr) return(OK);               /* not a session leader */
47268     fp->fp_sesldr = FALSE;
47269     if (fp->fp_tty == 0) return(OK);              /* no controlling tty */
47270     dev = fp->fp_tty;
47271
47272     for (rfp = &fproc[LOW_USER]; rfp < &fproc[NR_PROCS]; rfp++) {
47273           if (rfp->fp_tty == dev) rfp->fp_tty = 0;
47274
47275           for (i = 0; i < OPEN_MAX; i++) {
47276                 if ((rfilp = rfp->fp_filp[i]) == NIL_FILP) continue;
47277                 if (rfilp->filp_mode == FILP_CLOSED) continue;
47278                 rip = rfilp->filp_ino;
47279                 if ((rip->i_mode & I_TYPE) != I_CHAR_SPECIAL) continue;
47280                 if ((dev_t) rip->i_zone[0] != dev) continue;
47281                 dev_mess.m_type = DEV_CLOSE;
47282                 dev_mess.DEVICE = dev;
47283                 major = (dev >> MAJOR) & BYTE;  /* major device nr */
47284                 task = dmap[major].dmap_task;   /* device task nr */
47285                 (*dmap[major].dmap_close)(task, &dev_mess);
47286                 rfilp->filp_mode = FILP_CLOSED;
47287           }
47288     }
47289     return(OK);
47290   }


47293   /*===========================================================================*
47294    *                                do_set                                     *
```

```
47295    *===========================================================================*/
47296   PUBLIC int do_set()
47297   {
47298   /* Set uid_t or gid_t field. */
47299
47300     register struct fproc *tfp;
47301
47302     /* Only MM may make this call directly. */
47303     if (who != MM_PROC_NR) return(EGENERIC);
47304
47305     tfp = &fproc[slot1];
47306     if (fs_call == SETUID) {
47307           tfp->fp_realuid = (uid_t) real_user_id;
47308           tfp->fp_effuid =  (uid_t) eff_user_id;
47309     }
47310     if (fs_call == SETGID) {
47311           tfp->fp_effgid =  (gid_t) eff_grp_id;
47312           tfp->fp_realgid = (gid_t) real_grp_id;
47313     }
47314     return(OK);
47315   }


47318   /*===========================================================================*
47319    *                                do_revive                                  *
47320    *===========================================================================*/
47321   PUBLIC int do_revive()
47322   {
47323   /* A task, typically TTY, has now gotten the characters that were needed for a
47324    * previous read.  The process did not get a reply when it made the call.
47325    * Instead it was suspended.  Now we can send the reply to wake it up.  This
47326    * business has to be done carefully, since the incoming message is from
47327    * a task (to which no reply can be sent), and the reply must go to a process
47328    * that blocked earlier.  The reply to the caller is inhibited by setting the
47329    * 'dont_reply' flag, and the reply to the blocked process is done explicitly
47330    * in revive().
47331    */
47332
47333   #if !ALLOW_USER_SEND
47334     if (who >= LOW_USER) return(EPERM);
47335   #endif
47336
47337     revive(m.REP_PROC_NR, m.REP_STATUS);
47338     dont_reply = TRUE;            /* don't reply to the TTY task */
47339     return(OK);
47340   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/mount.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

47400    /* This file performs the MOUNT and UMOUNT system calls.
47401     *
47402     * The entry points into this file are
47403     *   do_mount:  perform the MOUNT system call
47404     *   do_umount: perform the UMOUNT system call
47405     */
47406
47407    #include "fs.h"
47408    #include <fcntl.h>
47409    #include <minix/com.h>
47410    #include <sys/stat.h>
47411    #include "buf.h"
47412    #include "dev.h"
47413    #include "file.h"
47414    #include "fproc.h"
47415    #include "inode.h"
47416    #include "param.h"
47417    #include "super.h"
47418
47419    PRIVATE message dev_mess;
47420
47421    FORWARD _PROTOTYPE( dev_t name_to_dev, (char *path)                    );
47422
47423    /*===========================================================================*
47424     *                                do_mount                                   *
47425     *===========================================================================*/
47426    PUBLIC int do_mount()
47427    {
47428    /* Perform the mount(name, mfile, rd_only) system call. */
47429
47430      register struct inode *rip, *root_ip;
47431      struct super_block *xp, *sp;
47432      dev_t dev;
47433      mode_t bits;
47434      int rdir, mdir;               /* TRUE iff {root|mount} file is dir */
47435      int r, found, major, task;
47436
47437      /* Only the super-user may do MOUNT. */
47438      if (!super_user) return(EPERM);
47439
47440      /* If 'name' is not for a block special file, return error. */
47441      if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
47442      if ( (dev = name_to_dev(user_path)) == NO_DEV) return(err_code);
47443
47444      /* Scan super block table to see if dev already mounted & find a free slot.*/
47445      sp = NIL_SUPER;
47446      found = FALSE;
47447      for (xp = &super_block[0]; xp < &super_block[NR_SUPERS]; xp++) {
47448            if (xp->s_dev == dev) found = TRUE;     /* is it mounted already? */
47449            if (xp->s_dev == NO_DEV) sp = xp;       /* record free slot */
47450      }
47451      if (found) return(EBUSY);     /* already mounted */
47452      if (sp == NIL_SUPER) return(ENFILE);  /* no super block available */
47453
47454      dev_mess.m_type = DEV_OPEN;           /* distinguish from close */
```

```
47455     dev_mess.DEVICE = dev;                /* Touch the device. */
47456     if (rd_only) dev_mess.COUNT = R_BIT;
47457     else  dev_mess.COUNT = R_BIT|W_BIT;
47458
47459     major = (dev >> MAJOR) & BYTE;
47460     if (major <= 0 || major >= max_major) return(ENODEV);
47461     task = dmap[major].dmap_task;         /* device task nr */
47462     (*dmap[major].dmap_open)(task, &dev_mess);
47463     if (dev_mess.REP_STATUS != OK) return(EINVAL);
47464
47465     /* Fill in the super block. */
47466     sp->s_dev = dev;              /* read_super() needs to know which dev */
47467     r = read_super(sp);
47468
47469     /* Is it recognized as a Minix filesystem? */
47470     if (r != OK) {
47471           dev_mess.m_type = DEV_CLOSE;
47472           dev_mess.DEVICE = dev;
47473           (*dmap[major].dmap_close)(task, &dev_mess);
47474           return(r);
47475     }
47476
47477     /* Now get the inode of the file to be mounted on. */
47478     if (fetch_name(name2, name2_length, M1) != OK) {
47479           sp->s_dev = NO_DEV;
47480           dev_mess.m_type = DEV_CLOSE;
47481           dev_mess.DEVICE = dev;
47482           (*dmap[major].dmap_close)(task, &dev_mess);
47483           return(err_code);
47484     }
47485     if ( (rip = eat_path(user_path)) == NIL_INODE) {
47486           sp->s_dev = NO_DEV;
47487           dev_mess.m_type = DEV_CLOSE;
47488           dev_mess.DEVICE = dev;
47489           (*dmap[major].dmap_close)(task, &dev_mess);
47490           return(err_code);
47491     }
47492
47493     /* It may not be busy. */
47494     r = OK;
47495     if (rip->i_count > 1) r = EBUSY;
47496
47497     /* It may not be special. */
47498     bits = rip->i_mode & I_TYPE;
47499     if (bits == I_BLOCK_SPECIAL || bits == I_CHAR_SPECIAL) r = ENOTDIR;
47500
47501     /* Get the root inode of the mounted file system. */
47502     root_ip = NIL_INODE;          /* if 'r' not OK, make sure this is defined */
47503     if (r == OK) {
47504           if ( (root_ip = get_inode(dev, ROOT_INODE)) == NIL_INODE) r = err_code;
47505     }
47506     if (root_ip != NIL_INODE && root_ip->i_mode == 0) r = EINVAL;
47507
47508     /* File types of 'rip' and 'root_ip' may not conflict. */
47509     if (r == OK) {
47510           mdir = ((rip->i_mode & I_TYPE) == I_DIRECTORY);  /* TRUE iff dir */
47511           rdir = ((root_ip->i_mode & I_TYPE) == I_DIRECTORY);
47512           if (!mdir && rdir) r = EISDIR;
47513     }
47514
```

```
47515     /* If error, return the super block and both inodes; release the maps. */
47516     if (r != OK) {
47517           put_inode(rip);
47518           put_inode(root_ip);
47519           (void) do_sync();
47520           invalidate(dev);
47521
47522           sp->s_dev = NO_DEV;
47523           dev_mess.m_type = DEV_CLOSE;
47524           dev_mess.DEVICE = dev;
47525           (*dmap[major].dmap_close)(task, &dev_mess);
47526           return(r);
47527     }
47528
47529     /* Nothing else can go wrong.  Perform the mount. */
47530     rip->i_mount = I_MOUNT;       /* this bit says the inode is mounted on */
47531     sp->s_imount = rip;
47532     sp->s_isup = root_ip;
47533     sp->s_rd_only = rd_only;
47534     return(OK);
47535   }


47538   /*===========================================================================*
47539    *                                do_umount                                  *
47540    *===========================================================================*/
47541   PUBLIC int do_umount()
47542   {
47543   /* Perform the umount(name) system call. */
47544
47545     register struct inode *rip;
47546     struct super_block *sp, *sp1;
47547     dev_t dev;
47548     int count;
47549     int major, task;
47550
47551     /* Only the super-user may do UMOUNT. */
47552     if (!super_user) return(EPERM);
47553
47554     /* If 'name' is not for a block special file, return error. */
47555     if (fetch_name(name, name_length, M3) != OK) return(err_code);
47556     if ( (dev = name_to_dev(user_path)) == NO_DEV) return(err_code);
47557
47558     /* See if the mounted device is busy.  Only 1 inode using it should be
47559      * open -- the root inode -- and that inode only 1 time.
47560      */
47561     count = 0;
47562     for (rip = &inode[0]; rip< &inode[NR_INODES]; rip++)
47563           if (rip->i_count > 0 && rip->i_dev == dev) count += rip->i_count;
47564     if (count > 1) return(EBUSY); /* can't umount a busy file system */
47565
47566     /* Find the super block. */
47567     sp = NIL_SUPER;
47568     for (sp1 = &super_block[0]; sp1 < &super_block[NR_SUPERS]; sp1++) {
47569           if (sp1->s_dev == dev) {
47570                 sp = sp1;
47571                 break;
47572           }
47573     }
47574
```

```
47575     /* Sync the disk, and invalidate cache. */
47576     (void) do_sync();             /* force any cached blocks out of memory */
47577     invalidate(dev);              /* invalidate cache entries for this dev */
47578     if (sp == NIL_SUPER) return(EINVAL);
47579
47580     major = (dev >> MAJOR) & BYTE;        /* major device nr */
47581     task = dmap[major].dmap_task; /* device task nr */
47582     dev_mess.m_type = DEV_CLOSE;          /* distinguish from open */
47583     dev_mess.DEVICE = dev;
47584     (*dmap[major].dmap_close)(task, &dev_mess);
47585
47586     /* Finish off the unmount. */
47587     sp->s_imount->i_mount = NO_MOUNT;     /* inode returns to normal */
47588     put_inode(sp->s_imount);      /* release the inode mounted on */
47589     put_inode(sp->s_isup);        /* release the root inode of the mounted fs */
47590     sp->s_imount = NIL_INODE;
47591     sp->s_dev = NO_DEV;
47592     return(OK);
47593   }


47596   /*===========================================================================*
47597    *                              name_to_dev                                  *
47598    *===========================================================================*/
47599   PRIVATE dev_t name_to_dev(path)
47600   char *path;                   /* pointer to path name */
47601   {
47602   /* Convert the block special file 'path' to a device number.  If 'path'
47603    * is not a block special file, return error code in 'err_code'.
47604    */
47605
47606     register struct inode *rip;
47607     register dev_t dev;
47608
47609     /* If 'path' can't be opened, give up immediately. */
47610     if ( (rip = eat_path(path)) == NIL_INODE) return(NO_DEV);
47611
47612     /* If 'path' is not a block special file, return error. */
47613     if ( (rip->i_mode & I_TYPE) != I_BLOCK_SPECIAL) {
47614           err_code = ENOTBLK;
47615           put_inode(rip);
47616           return(NO_DEV);
47617     }
47618
47619     /* Extract the device number. */
47620     dev = (dev_t) rip->i_zone[0];
47621     put_inode(rip);
47622     return(dev);
47623   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/open.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

47700    /* This file contains the procedures for creating, opening, closing, and
47701     * seeking on files.
47702     *
47703     * The entry points into this file are
47704     *   do_creat:  perform the CREAT system call
47705     *   do_open:   perform the OPEN system call
47706     *   do_mknod:  perform the MKNOD system call
47707     *   do_mkdir:  perform the MKDIR system call
47708     *   do_close:  perform the CLOSE system call
47709     *   do_lseek:  perform the LSEEK system call
47710     */
47711
47712    #include "fs.h"
47713    #include <sys/stat.h>
47714    #include <fcntl.h>
47715    #include <minix/callnr.h>
47716    #include <minix/com.h>
47717    #include "buf.h"
47718    #include "dev.h"
47719    #include "file.h"
47720    #include "fproc.h"
47721    #include "inode.h"
47722    #include "lock.h"
47723    #include "param.h"
47724
47725    PRIVATE message dev_mess;
47726    PRIVATE char mode_map[] = {R_BIT, W_BIT, R_BIT|W_BIT, 0};
47727
47728    FORWARD _PROTOTYPE( int common_open, (int oflags, Mode_t omode)         );
47729    FORWARD _PROTOTYPE( int pipe_open, (struct inode *rip,Mode_t bits,int oflags));
47730    FORWARD _PROTOTYPE( struct inode *new_node, (char *path, Mode_t bits,
47731                                                 zone_t z0)      );
47732
47733
47734    /*===========================================================================*
47735     *                               do_creat                                    *
47736     *===========================================================================*/
47737    PUBLIC int do_creat()
47738    {
47739    /* Perform the creat(name, mode) system call. */
47740      int r;
47741
47742      if (fetch_name(name, name_length, M3) != OK) return(err_code);
47743      r = common_open(O_WRONLY | O_CREAT | O_TRUNC, (mode_t) mode);
47744      return(r);
47745    }


47748    /*===========================================================================*
47749     *                               do_open                                     *
47750     *===========================================================================*/
47751    PUBLIC int do_open()
47752    {
47753    /* Perform the open(name, flags,...) system call. */
47754
```

```
47755     int create_mode = 0;          /* is really mode_t but this gives problems */
47756     int r;
47757
47758     /* If O_CREAT is set, open has three parameters, otherwise two. */
47759     if (mode & O_CREAT) {
47760           create_mode = c_mode;
47761           r = fetch_name(c_name, name1_length, M1);
47762     } else {
47763           r = fetch_name(name, name_length, M3);
47764     }
47765
47766     if (r != OK) return(err_code); /* name was bad */
47767     r = common_open(mode, create_mode);
47768     return(r);
47769   }


47772   /*===========================================================================*
47773    *                               common_open                                 *
47774    *===========================================================================*/
47775   PRIVATE int common_open(oflags, omode)
47776   register int oflags;
47777   mode_t omode;
47778   {
47779   /* Common code from do_creat and do_open. */
47780
47781     register struct inode *rip;
47782     int r, b, major, task, exist = TRUE;
47783     dev_t dev;
47784     mode_t bits;
47785     off_t pos;
47786     struct filp *fil_ptr, *filp2;
47787
47788     /* Remap the bottom two bits of oflags. */
47789     bits = (mode_t) mode_map[oflags & O_ACCMODE];
47790
47791     /* See if file descriptor and filp slots are available. */
47792     if ( (r = get_fd(0, bits, &fd, &fil_ptr)) != OK) return(r);
47793
47794     /* If O_CREATE is set, try to make the file. */
47795     if (oflags & O_CREAT) {
47796           /* Create a new inode by calling new_node(). */
47797           omode = I_REGULAR | (omode & ALL_MODES & fp->fp_umask);
47798           rip = new_node(user_path, omode, NO_ZONE);
47799           r = err_code;
47800           if (r == OK) exist = FALSE;      /* we just created the file */
47801           else if (r != EEXIST) return(r); /* other error */
47802           else exist = !(oflags & O_EXCL); /* file exists, if the O_EXCL
47803                                               flag is set this is an error */
47804     } else {
47805            /* Scan path name. */
47806           if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
47807     }
47808
47809     /* Claim the file descriptor and filp slot and fill them in. */
47810     fp->fp_filp[fd] = fil_ptr;
47811     fil_ptr->filp_count = 1;
47812     fil_ptr->filp_ino = rip;
47813     fil_ptr->filp_flags = oflags;
47814
```

```
47815      /* Only do the normal open code if we didn't just create the file. */
47816      if (exist) {
47817            /* Check protections. */
47818            if ((r = forbidden(rip, bits)) == OK) {
47819                    /* Opening reg. files directories and special files differ. */
47820                    switch (rip->i_mode & I_TYPE) {
47821                       case I_REGULAR:
47822                            /* Truncate regular file if O_TRUNC. */
47823                            if (oflags & O_TRUNC) {
47824                                    if ((r = forbidden(rip, W_BIT)) !=OK) break;
47825                                    truncate(rip);
47826                                    wipe_inode(rip);
47827                                    /* Send the inode from the inode cache to the
47828                                     * block cache, so it gets written on the next
47829                                     * cache flush.
47830                                     */
47831                                    rw_inode(rip, WRITING);
47832                            }
47833                            break;
47834
47835                       case I_DIRECTORY:
47836                            /* Directories may be read but not written. */
47837                            r = (bits & W_BIT ? EISDIR : OK);
47838                            break;
47839
47840                       case I_CHAR_SPECIAL:
47841                       case I_BLOCK_SPECIAL:
47842                            /* Invoke the driver for special processing. */
47843                            dev_mess.m_type = DEV_OPEN;
47844                            dev = (dev_t) rip->i_zone[0];
47845                            dev_mess.DEVICE = dev;
47846                            dev_mess.COUNT = bits | (oflags & ~O_ACCMODE);
47847                            major = (dev >> MAJOR) & BYTE;  /* major device nr */
47848                            if (major <= 0 || major >= max_major) {
47849                                    r = ENODEV;
47850                                    break;
47851                            }
47852                            task = dmap[major].dmap_task;   /* device task nr */
47853                            (*dmap[major].dmap_open)(task, &dev_mess);
47854                            r = dev_mess.REP_STATUS;
47855                            break;
47856
47857                       case I_NAMED_PIPE:
47858                            oflags |= O_APPEND;     /* force append mode */
47859                            fil_ptr->filp_flags = oflags;
47860                            r = pipe_open(rip, bits, oflags);
47861                            if (r == OK) {
47862                                    /* See if someone else is doing a rd or wt on
47863                                     * the FIFO.  If so, use its filp entry so the
47864                                     * file position will be automatically shared.
47865                                     */
47866                                    b = (bits & R_BIT ? R_BIT : W_BIT);
47867                                    fil_ptr->filp_count = 0; /* don't find self */
47868                                    if ((filp2 = find_filp(rip, b)) != NIL_FILP) {
47869                                            /* Co-reader or writer found. Use it.*/
47870                                            fp->fp_filp[fd] = filp2;
47871                                            filp2->filp_count++;
47872                                            filp2->filp_ino = rip;
47873                                            filp2->filp_flags = oflags;
47874
```

```
47875                                           /* i_count was incremented incorrectly
47876                                            * by eatpath above, not knowing that
47877                                            * we were going to use an existing
47878                                            * filp entry.  Correct this error.
47879                                            */
47880                                           rip->i_count--;
47881                                   } else {
47882                                           /* Nobody else found.  Restore filp. */
47883                                           fil_ptr->filp_count = 1;
47884                                           if (b == R_BIT)
47885                                                pos = rip->i_zone[V2_NR_DZONES+1];
47886                                           else
47887                                                pos = rip->i_zone[V2_NR_DZONES+2];
47888                                           fil_ptr->filp_pos = pos;
47889                                   }
47890                           }
47891                           break;
47892                   }
47893           }
47894     }
47895
47896     /* If error, release inode. */
47897     if (r != OK) {
47898           fp->fp_filp[fd] = NIL_FILP;
47899           fil_ptr->filp_count= 0;
47900           put_inode(rip);
47901           return(r);
47902     }
47903
47904     return(fd);
47905   }


47908   /*===========================================================================*
47909    *                              new_node                                     *
47910    *===========================================================================*/
47911   PRIVATE struct inode *new_node(path, bits, z0)
47912   char *path;                     /* pointer to path name */
47913   mode_t bits;                    /* mode of the new inode */
47914   zone_t z0;                      /* zone number 0 for new inode */
47915   {
47916   /* New_node() is called by common_open(), do_mknod(), and do_mkdir().
47917    * In all cases it allocates a new inode, makes a directory entry for it on
47918    * the path 'path', and initializes it.  It returns a pointer to the inode if
47919    * it can do this; otherwise it returns NIL_INODE.  It always sets 'err_code'
47920    * to an appropriate value (OK or an error code).
47921    */
47922
47923     register struct inode *rlast_dir_ptr, *rip;
47924     register int r;
47925     char string[NAME_MAX];
47926
47927     /* See if the path can be opened down to the last directory. */
47928     if ((rlast_dir_ptr = last_dir(path, string)) == NIL_INODE) return(NIL_INODE);
47929
47930     /* The final directory is accessible. Get final component of the path. */
47931     rip = advance(rlast_dir_ptr, string);
47932     if ( rip == NIL_INODE && err_code == ENOENT) {
47933           /* Last path component does not exist.  Make new directory entry. */
47934           if ( (rip = alloc_inode(rlast_dir_ptr->i_dev, bits)) == NIL_INODE) {
```

```
47935                   /* Can't creat new inode: out of inodes. */
47936                   put_inode(rlast_dir_ptr);
47937                   return(NIL_INODE);
47938           }
47939
47940           /* Force inode to the disk before making directory entry to make
47941            * the system more robust in the face of a crash: an inode with
47942            * no directory entry is much better than the opposite.
47943            */
47944           rip->i_nlinks++;
47945           rip->i_zone[0] = z0;            /* major/minor device numbers */
47946           rw_inode(rip, WRITING);         /* force inode to disk now */
47947
47948           /* New inode acquired.  Try to make directory entry. */
47949           if ((r = search_dir(rlast_dir_ptr, string, &rip->i_num,ENTER)) != OK) {
47950                   put_inode(rlast_dir_ptr);
47951                   rip->i_nlinks--;        /* pity, have to free disk inode */
47952                   rip->i_dirt = DIRTY;    /* dirty inodes are written out */
47953                   put_inode(rip); /* this call frees the inode */
47954                   err_code = r;
47955                   return(NIL_INODE);
47956           }
47957
47958     } else {
47959           /* Either last component exists, or there is some problem. */
47960           if (rip != NIL_INODE)
47961                   r = EEXIST;
47962           else
47963                   r = err_code;
47964     }
47965
47966     /* Return the directory inode and exit. */
47967     put_inode(rlast_dir_ptr);
47968     err_code = r;
47969     return(rip);
47970   }


47973   /*===========================================================================*
47974    *                               pipe_open                                   *
47975    *===========================================================================*/
47976   PRIVATE int pipe_open(rip, bits, oflags)
47977   register struct inode *rip;
47978   register mode_t bits;
47979   register int oflags;
47980   {
47981   /*  This function is called from common_open. It checks if
47982    *  there is at least one reader/writer pair for the pipe, if not
47983    *  it suspends the caller, otherwise it revives all other blocked
47984    *  processes hanging on the pipe.
47985    */
47986
47987     if (find_filp(rip, bits & W_BIT ? R_BIT : W_BIT) == NIL_FILP) {
47988           if (oflags & O_NONBLOCK) {
47989                   if (bits & W_BIT) return(ENXIO);
47990           } else
47991                   suspend(XPOPEN);        /* suspend caller */
47992     } else if (susp_count > 0) {/* revive blocked processes */
47993           release(rip, OPEN, susp_count);
47994           release(rip, CREAT, susp_count);
```

```
47995     }
47996     rip->i_pipe = I_PIPE;
47997
47998     return(OK);
47999   }


48002   /*===========================================================================*
48003    *                                do_mknod                                   *
48004    *===========================================================================*/
48005   PUBLIC int do_mknod()
48006   {
48007   /* Perform the mknod(name, mode, addr) system call. */
48008
48009     register mode_t bits, mode_bits;
48010     struct inode *ip;
48011
48012     /* Only the super_user may make nodes other than fifos. */
48013     mode_bits = (mode_t) m.m1_i2; /* mode of the inode */
48014     if (!super_user && ((mode_bits & I_TYPE) != I_NAMED_PIPE)) return(EPERM);
48015     if (fetch_name(m.m1_p1, m.m1_i1, M1) != OK) return(err_code);
48016     bits = (mode_bits & I_TYPE) | (mode_bits & ALL_MODES & fp->fp_umask);
48017     ip = new_node(user_path, bits, (zone_t) m.m1_i3);
48018     put_inode(ip);
48019     return(err_code);
48020   }


48023   /*===========================================================================*
48024    *                                do_mkdir                                   *
48025    *===========================================================================*/
48026   PUBLIC int do_mkdir()
48027   {
48028   /* Perform the mkdir(name, mode) system call. */
48029
48030     int r1, r2;                   /* status codes */
48031     ino_t dot, dotdot;            /* inode numbers for . and .. */
48032     mode_t bits;                  /* mode bits for the new inode */
48033     char string[NAME_MAX];        /* last component of the new dir's path name */
48034     register struct inode *rip, *ldirp;
48035
48036     /* Check to see if it is possible to make another link in the parent dir. */
48037     if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
48038     ldirp = last_dir(user_path, string);  /* pointer to new dir's parent */
48039     if (ldirp == NIL_INODE) return(err_code);
48040     if ( (ldirp->i_nlinks & BYTE) >= LINK_MAX) {
48041           put_inode(ldirp);       /* return parent */
48042           return(EMLINK);
48043     }
48044
48045     /* Next make the inode. If that fails, return error code. */
48046     bits = I_DIRECTORY | (mode & RWX_MODES & fp->fp_umask);
48047     rip = new_node(user_path, bits, (zone_t) 0);
48048     if (rip == NIL_INODE || err_code == EEXIST) {
48049           put_inode(rip);         /* can't make dir: it already exists */
48050           put_inode(ldirp);       /* return parent too */
48051           return(err_code);
48052     }
48053
48054     /* Get the inode numbers for . and .. to enter in the directory. */
```

```
48055     dotdot = ldirp->i_num;        /* parent's inode number */
48056     dot = rip->i_num;             /* inode number of the new dir itself */
48057
48058     /* Now make dir entries for . and .. unless the disk is completely full. */
48059     /* Use dot1 and dot2, so the mode of the directory isn't important. */
48060     rip->i_mode = bits;   /* set mode */
48061     r1 = search_dir(rip, dot1, &dot, ENTER);      /* enter . in the new dir */
48062     r2 = search_dir(rip, dot2, &dotdot, ENTER);   /* enter .. in the new dir */
48063
48064     /* If both . and .. were successfully entered, increment the link counts. */
48065     if (r1 == OK && r2 == OK) {
48066           /* Normal case.  It was possible to enter . and .. in the new dir. */
48067           rip->i_nlinks++;        /* this accounts for . */
48068           ldirp->i_nlinks++;      /* this accounts for .. */
48069           ldirp->i_dirt = DIRTY;  /* mark parent's inode as dirty */
48070     } else {
48071           /* It was not possible to enter . or .. probably disk was full. */
48072           (void) search_dir(ldirp, string, (ino_t *) 0, DELETE);
48073           rip->i_nlinks--;        /* undo the increment done in new_node() */
48074     }
48075     rip->i_dirt = DIRTY;          /* either way, i_nlinks has changed */
48076
48077     put_inode(ldirp);             /* return the inode of the parent dir */
48078     put_inode(rip);               /* return the inode of the newly made dir */
48079     return(err_code);             /* new_node() always sets 'err_code' */
48080   }


48083   /*===========================================================================*
48084    *                                do_close                                   *
48085    *===========================================================================*/
48086   PUBLIC int do_close()
48087   {
48088   /* Perform the close(fd) system call. */
48089
48090     register struct filp *rfilp;
48091     register struct inode *rip;
48092     struct file_lock *flp;
48093     int rw, mode_word, major, task, lock_count;
48094     dev_t dev;
48095
48096     /* First locate the inode that belongs to the file descriptor. */
48097     if ( (rfilp = get_filp(fd)) == NIL_FILP) return(err_code);
48098     rip = rfilp->filp_ino;        /* 'rip' points to the inode */
48099
48100     if (rfilp->filp_count - 1 == 0 && rfilp->filp_mode != FILP_CLOSED) {
48101           /* Check to see if the file is special. */
48102           mode_word = rip->i_mode & I_TYPE;
48103           if (mode_word == I_CHAR_SPECIAL || mode_word == I_BLOCK_SPECIAL) {
48104                   dev = (dev_t) rip->i_zone[0];
48105                   if (mode_word == I_BLOCK_SPECIAL)  {
48106                           /* Invalidate cache entries unless special is mounted
48107                            * or ROOT
48108                            */
48109                           if (!mounted(rip)) {
48110                                   (void) do_sync();       /* purge cache */
48111                                   invalidate(dev);
48112                           }
48113                   }
48114                   /* Use the dmap_close entry to do any special processing
```

```
48115                    * required.
48116                    */
48117                   dev_mess.m_type = DEV_CLOSE;
48118                   dev_mess.DEVICE = dev;
48119                   major = (dev >> MAJOR) & BYTE;  /* major device nr */
48120                   task = dmap[major].dmap_task;   /* device task nr */
48121                   (*dmap[major].dmap_close)(task, &dev_mess);
48122           }
48123     }
48124
48125     /* If the inode being closed is a pipe, release everyone hanging on it. */
48126     if (rip->i_pipe == I_PIPE) {
48127           rw = (rfilp->filp_mode & R_BIT ? WRITE : READ);
48128           release(rip, rw, NR_PROCS);
48129     }
48130
48131     /* If a write has been done, the inode is already marked as DIRTY. */
48132     if (--rfilp->filp_count == 0) {
48133           if (rip->i_pipe == I_PIPE && rip->i_count > 1) {
48134                   /* Save the file position in the i-node in case needed later.
48135                    * The read and write positions are saved separately.  The
48136                    * last 3 zones in the i-node are not used for (named) pipes.
48137                    */
48138                   if (rfilp->filp_mode == R_BIT)
48139                           rip->i_zone[V2_NR_DZONES+1] = (zone_t) rfilp->filp_pos;
48140                   else
48141                           rip->i_zone[V2_NR_DZONES+2] = (zone_t) rfilp->filp_pos;
48142           }
48143           put_inode(rip);
48144     }
48145
48146     fp->fp_cloexec &= ~(1L << fd);        /* turn off close-on-exec bit */
48147     fp->fp_filp[fd] = NIL_FILP;
48148
48149     /* Check to see if the file is locked.  If so, release all locks. */
48150     if (nr_locks == 0) return(OK);
48151     lock_count = nr_locks;        /* save count of locks */
48152     for (flp = &file_lock[0]; flp < &file_lock[NR_LOCKS]; flp++) {
48153           if (flp->lock_type == 0) continue;      /* slot not in use */
48154           if (flp->lock_inode == rip && flp->lock_pid == fp->fp_pid) {
48155                   flp->lock_type = 0;
48156                   nr_locks--;
48157           }
48158     }
48159     if (nr_locks < lock_count) lock_revive();     /* lock released */
48160     return(OK);
48161   }


48164   /*===========================================================================*
48165    *                                do_lseek                                   *
48166    *===========================================================================*/
48167   PUBLIC int do_lseek()
48168   {
48169   /* Perform the lseek(ls_fd, offset, whence) system call. */
48170
48171     register struct filp *rfilp;
48172     register off_t pos;
48173
48174     /* Check to see if the file descriptor is valid. */
```

```
48175      if ( (rfilp = get_filp(ls_fd)) == NIL_FILP) return(err_code);
48176
48177      /* No lseek on pipes. */
48178      if (rfilp->filp_ino->i_pipe == I_PIPE) return(ESPIPE);
48179
48180      /* The value of 'whence' determines the start position to use. */
48181      switch(whence) {
48182            case 0: pos = 0;        break;
48183            case 1: pos = rfilp->filp_pos;  break;
48184            case 2: pos = rfilp->filp_ino->i_size;  break;
48185            default: return(EINVAL);
48186      }
48187
48188      /* Check for overflow. */
48189      if (((long)offset > 0) && ((long)(pos + offset) < (long)pos)) return(EINVAL);
48190      if (((long)offset < 0) && ((long)(pos + offset) > (long)pos)) return(EINVAL);
48191      pos = pos + offset;
48192
48193      if (pos != rfilp->filp_pos)
48194            rfilp->filp_ino->i_seek = ISEEK;        /* inhibit read ahead */
48195      rfilp->filp_pos = pos;
48196      reply_l1 = pos;               /* insert the long into the output message */
48197      return(OK);
48198    }
```

++
src/fs/path.c
++

```
48200    /* This file contains the procedures that look up path names in the directory
48201     * system and determine the inode number that goes with a given path name.
48202     *
48203     *  The entry points into this file are
48204     *   eat_path:   the 'main' routine of the path-to-inode conversion mechanism
48205     *   last_dir:   find the final directory on a given path
48206     *   advance:    parse one component of a path name
48207     *   search_dir: search a directory for a string and return its inode number
48208     */
48209
48210    #include "fs.h"
48211    #include <string.h>
48212    #include <minix/callnr.h>
48213    #include "buf.h"
48214    #include "file.h"
48215    #include "fproc.h"
48216    #include "inode.h"
48217    #include "super.h"
48218
48219    PUBLIC char dot1[2] = ".";      /* used for search_dir to bypass the access */
48220    PUBLIC char dot2[3] = "..";     /* permissions for . and ..                 */
48221
48222    FORWARD _PROTOTYPE( char *get_name, (char *old_name, char string [NAME_MAX]) );
48223
48224    /*===========================================================================*
48225     *                               eat_path                                    *
48226     *===========================================================================*/
48227    PUBLIC struct inode *eat_path(path)
48228    char *path;                     /* the path name to be parsed */
48229    {
```

```
48230   /* Parse the path 'path' and put its inode in the inode table. If not possible,
48231    * return NIL_INODE as function value and an error code in 'err_code'.
48232    */
48233
48234     register struct inode *ldip, *rip;
48235     char string[NAME_MAX];        /* hold 1 path component name here */
48236
48237     /* First open the path down to the final directory. */
48238     if ( (ldip = last_dir(path, string)) == NIL_INODE)
48239           return(NIL_INODE);      /* we couldn't open final directory */
48240
48241     /* The path consisting only of "/" is a special case, check for it. */
48242     if (string[0] == '\0') return(ldip);
48243
48244     /* Get final component of the path. */
48245     rip = advance(ldip, string);
48246     put_inode(ldip);
48247     return(rip);
48248   }


48251   /*===========================================================================*
48252    *                              last_dir                                     *
48253    *===========================================================================*/
48254   PUBLIC struct inode *last_dir(path, string)
48255   char *path;                     /* the path name to be parsed */
48256   char string[NAME_MAX];          /* the final component is returned here */
48257   {
48258   /* Given a path, 'path', located in the fs address space, parse it as
48259    * far as the last directory, fetch the inode for the last directory into
48260    * the inode table, and return a pointer to the inode.  In
48261    * addition, return the final component of the path in 'string'.
48262    * If the last directory can't be opened, return NIL_INODE and
48263    * the reason for failure in 'err_code'.
48264    */
48265
48266     register struct inode *rip;
48267     register char *new_name;
48268     register struct inode *new_ip;
48269
48270     /* Is the path absolute or relative?  Initialize 'rip' accordingly. */
48271     rip = (*path == '/' ? fp->fp_rootdir : fp->fp_workdir);
48272
48273     /* If dir has been removed or path is empty, return ENOENT. */
48274     if (rip->i_nlinks == 0 || *path == '\0') {
48275           err_code = ENOENT;
48276           return(NIL_INODE);
48277     }
48278
48279     dup_inode(rip);               /* inode will be returned with put_inode */
48280
48281     /* Scan the path component by component. */
48282     while (TRUE) {
48283           /* Extract one component. */
48284           if ( (new_name = get_name(path, string)) == (char*) 0) {
48285                   put_inode(rip); /* bad path in user space */
48286                   return(NIL_INODE);
48287           }
48288           if (*new_name == '\0')
48289                   if ( (rip->i_mode & I_TYPE) == I_DIRECTORY)
```

```
48290                           return(rip);    /* normal exit */
48291                   else {
48292                           /* last file of path prefix is not a directory */
48293                           put_inode(rip);
48294                           err_code = ENOTDIR;
48295                           return(NIL_INODE);
48296                   }
48297
48298           /* There is more path.  Keep parsing. */
48299           new_ip = advance(rip, string);
48300           put_inode(rip);         /* rip either obsolete or irrelevant */
48301           if (new_ip == NIL_INODE) return(NIL_INODE);
48302
48303           /* The call to advance() succeeded.  Fetch next component. */
48304           path = new_name;
48305           rip = new_ip;
48306     }
48307   }


48310   /*===========================================================================*
48311    *                              get_name                                     *
48312    *===========================================================================*/
48313   PRIVATE char *get_name(old_name, string)
48314   char *old_name;                 /* path name to parse */
48315   char string[NAME_MAX];          /* component extracted from 'old_name' */
48316   {
48317   /* Given a pointer to a path name in fs space, 'old_name', copy the next
48318    * component to 'string' and pad with zeros.  A pointer to that part of
48319    * the name as yet unparsed is returned.  Roughly speaking,
48320    * 'get_name' = 'old_name' - 'string'.
48321    *
48322    * This routine follows the standard convention that /usr/ast, /usr//ast,
48323    * //usr///ast and /usr/ast/ are all equivalent.
48324    */
48325
48326     register int c;
48327     register char *np, *rnp;
48328
48329     np = string;                  /* 'np' points to current position */
48330     rnp = old_name;               /* 'rnp' points to unparsed string */
48331     while ( (c = *rnp) == '/') rnp++;     /* skip leading slashes */
48332
48333     /* Copy the unparsed path, 'old_name', to the array, 'string'. */
48334     while ( rnp < &old_name[PATH_MAX]  &&  c != '/'   &&  c != '\0') {
48335           if (np < &string[NAME_MAX]) *np++ = c;
48336           c = *++rnp;             /* advance to next character */
48337     }
48338
48339     /* To make /usr/ast/ equivalent to /usr/ast, skip trailing slashes. */
48340     while (c == '/' && rnp < &old_name[PATH_MAX]) c = *++rnp;
48341
48342     if (np < &string[NAME_MAX]) *np = '\0';       /* Terminate string */
48343
48344     if (rnp >= &old_name[PATH_MAX]) {
48345           err_code = ENAMETOOLONG;
48346           return((char *) 0);
48347     }
48348     return(rnp);
48349   }
```

```
48352   /*===========================================================================*
48353    *                               advance                                     *
48354    *===========================================================================*/
48355   PUBLIC struct inode *advance(dirp, string)
48356   struct inode *dirp;             /* inode for directory to be searched */
48357   char string[NAME_MAX];          /* component name to look for */
48358   {
48359   /* Given a directory and a component of a path, look up the component in
48360    * the directory, find the inode, open it, and return a pointer to its inode
48361    * slot.  If it can't be done, return NIL_INODE.
48362    */
48363
48364     register struct inode *rip;
48365     struct inode *rip2;
48366     register struct super_block *sp;
48367     int r, inumb;
48368     dev_t mnt_dev;
48369     ino_t numb;
48370
48371     /* If 'string' is empty, yield same inode straight away. */
48372     if (string[0] == '\0') return(get_inode(dirp->i_dev, (int) dirp->i_num));
48373
48374     /* Check for NIL_INODE. */
48375     if (dirp == NIL_INODE) return(NIL_INODE);
48376
48377     /* If 'string' is not present in the directory, signal error. */
48378     if ( (r = search_dir(dirp, string, &numb, LOOK_UP)) != OK) {
48379           err_code = r;
48380           return(NIL_INODE);
48381     }
48382
48383     /* Don't go beyond the current root directory, unless the string is dot2. */
48384     if (dirp == fp->fp_rootdir && strcmp(string, "..") == 0 && string != dot2)
48385                   return(get_inode(dirp->i_dev, (int) dirp->i_num));
48386
48387     /* The component has been found in the directory.  Get inode. */
48388     if ( (rip = get_inode(dirp->i_dev, (int) numb)) == NIL_INODE)
48389           return(NIL_INODE);
48390
48391     if (rip->i_num == ROOT_INODE)
48392           if (dirp->i_num == ROOT_INODE) {
48393               if (string[1] == '.') {
48394                   for (sp = &super_block[1]; sp < &super_block[NR_SUPERS]; sp++){
48395                           if (sp->s_dev == rip->i_dev) {
48396                                   /* Release the root inode.  Replace by the
48397                                    * inode mounted on.
48398                                    */
48399                                   put_inode(rip);
48400                                   mnt_dev = sp->s_imount->i_dev;
48401                                   inumb = (int) sp->s_imount->i_num;
48402                                   rip2 = get_inode(mnt_dev, inumb);
48403                                   rip = advance(rip2, string);
48404                                   put_inode(rip2);
48405                                   break;
48406                           }
48407                   }
48408               }
48409           }
```

```
48410     if (rip == NIL_INODE) return(NIL_INODE);
48411
48412     /* See if the inode is mounted on.  If so, switch to root directory of the
48413      * mounted file system.  The super_block provides the linkage between the
48414      * inode mounted on and the root directory of the mounted file system.
48415      */
48416     while (rip != NIL_INODE && rip->i_mount == I_MOUNT) {
48417           /* The inode is indeed mounted on. */
48418           for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++) {
48419                 if (sp->s_imount == rip) {
48420                       /* Release the inode mounted on.  Replace by the
48421                        * inode of the root inode of the mounted device.
48422                        */
48423                       put_inode(rip);
48424                       rip = get_inode(sp->s_dev, ROOT_INODE);
48425                       break;
48426                 }
48427           }
48428     }
48429     return(rip);          /* return pointer to inode's component */
48430   }


48433   /*===========================================================================*
48434    *                              search_dir                                   *
48435    *===========================================================================*/
48436   PUBLIC int search_dir(ldir_ptr, string, numb, flag)
48437   register struct inode *ldir_ptr;        /* ptr to inode for dir to search */
48438   char string[NAME_MAX];          /* component to search for */
48439   ino_t *numb;                    /* pointer to inode number */
48440   int flag;                       /* LOOK_UP, ENTER, DELETE or IS_EMPTY */
48441   {
48442   /* This function searches the directory whose inode is pointed to by 'ldip':
48443    * if (flag == ENTER)  enter 'string' in the directory with inode # '*numb';
48444    * if (flag == DELETE) delete 'string' from the directory;
48445    * if (flag == LOOK_UP) search for 'string' and return inode # in 'numb';
48446    * if (flag == IS_EMPTY) return OK if only . and .. in dir else ENOTEMPTY;
48447    *
48448    *    if 'string' is dot1 or dot2, no access permissions are checked.
48449    */
48450
48451     register struct direct *dp;
48452     register struct buf *bp;
48453     int i, r, e_hit, t, match;
48454     mode_t bits;
48455     off_t pos;
48456     unsigned new_slots, old_slots;
48457     block_t b;
48458     struct super_block *sp;
48459     int extended = 0;
48460
48461     /* If 'ldir_ptr' is not a pointer to a dir inode, error. */
48462     if ( (ldir_ptr->i_mode & I_TYPE) != I_DIRECTORY) return(ENOTDIR);
48463
48464     r = OK;
48465
48466     if (flag != IS_EMPTY) {
48467           bits = (flag == LOOK_UP ? X_BIT : W_BIT | X_BIT);
48468
48469           if (string == dot1 || string == dot2) {
```

```
48470                 if (flag != LOOK_UP) r = read_only(ldir_ptr);
48471                                       /* only a writable device is required. */
48472           }
48473           else r = forbidden(ldir_ptr, bits); /* check access permissions */
48474      }
48475      if (r != OK) return(r);
48476
48477      /* Step through the directory one block at a time. */
48478      old_slots = (unsigned) (ldir_ptr->i_size/DIR_ENTRY_SIZE);
48479      new_slots = 0;
48480      e_hit = FALSE;
48481      match = 0;                    /* set when a string match occurs */
48482
48483      for (pos = 0; pos < ldir_ptr->i_size; pos += BLOCK_SIZE) {
48484           b = read_map(ldir_ptr, pos);    /* get block number */
48485
48486           /* Since directories don't have holes, 'b' cannot be NO_BLOCK. */
48487           bp = get_block(ldir_ptr->i_dev, b, NORMAL);     /* get a dir block */
48488
48489           /* Search a directory block. */
48490           for (dp = &bp->b_dir[0]; dp < &bp->b_dir[NR_DIR_ENTRIES]; dp++) {
48491                   if (++new_slots > old_slots) { /* not found, but room left */
48492                           if (flag == ENTER) e_hit = TRUE;
48493                           break;
48494                   }
48495
48496                   /* Match occurs if string found. */
48497                   if (flag != ENTER && dp->d_ino != 0) {
48498                           if (flag == IS_EMPTY) {
48499                                   /* If this test succeeds, dir is not empty. */
48500                                   if (strcmp(dp->d_name, "." ) != 0 &&
48501                                       strcmp(dp->d_name, "..") != 0) match = 1;
48502                           } else {
48503                                   if (strncmp(dp->d_name, string, NAME_MAX) == 0)
48504                                           match = 1;
48505                           }
48506                   }
48507
48508                   if (match) {
48509                           /* LOOK_UP or DELETE found what it wanted. */
48510                           r = OK;
48511                           if (flag == IS_EMPTY) r = ENOTEMPTY;
48512                           else if (flag == DELETE) {
48513                                   /* Save d_ino for recovery. */
48514                                   t = NAME_MAX - sizeof(ino_t);
48515                                   *((ino_t *) &dp->d_name[t]) = dp->d_ino;
48516                                   dp->d_ino = 0;  /* erase entry */
48517                                   bp->b_dirt = DIRTY;
48518                                   ldir_ptr->i_update |= CTIME | MTIME;
48519                                   ldir_ptr->i_dirt = DIRTY;
48520                           } else {
48521                                   sp = ldir_ptr->i_sp;    /* 'flag' is LOOK_UP */
48522                                   *numb = conv2(sp->s_native, (int) dp->d_ino);
48523                           }
48524                           put_block(bp, DIRECTORY_BLOCK);
48525                           return(r);
48526                   }
48527
48528
48529                   /* Check for free slot for the benefit of ENTER. */
```

```
48530                   if (flag == ENTER && dp->d_ino == 0) {
48531                           e_hit = TRUE;   /* we found a free slot */
48532                           break;
48533                   }
48534           }
48535
48536           /* The whole block has been searched or ENTER has a free slot. */
48537           if (e_hit) break;       /* e_hit set if ENTER can be performed now */
48538           put_block(bp, DIRECTORY_BLOCK); /* otherwise, continue searching dir */
48539      }
48540
48541      /* The whole directory has now been searched. */
48542      if (flag != ENTER) return(flag == IS_EMPTY ? OK : ENOENT);
48543
48544      /* This call is for ENTER.  If no free slot has been found so far, try to
48545       * extend directory.
48546       */
48547      if (e_hit == FALSE) { /* directory is full and no room left in last block */
48548           new_slots++;            /* increase directory size by 1 entry */
48549           if (new_slots == 0) return(EFBIG); /* dir size limited by slot count */
48550           if ( (bp = new_block(ldir_ptr, ldir_ptr->i_size)) == NIL_BUF)
48551                   return(err_code);
48552           dp = &bp->b_dir[0];
48553           extended = 1;
48554      }
48555
48556      /* 'bp' now points to a directory block with space. 'dp' points to slot. */
48557      (void) memset(dp->d_name, 0, (size_t) NAME_MAX); /* clear entry */
48558      for (i = 0; string[i] && i < NAME_MAX; i++) dp->d_name[i] = string[i];
48559      sp = ldir_ptr->i_sp;
48560      dp->d_ino = conv2(sp->s_native, (int) *numb);
48561      bp->b_dirt = DIRTY;
48562      put_block(bp, DIRECTORY_BLOCK);
48563      ldir_ptr->i_update |= CTIME | MTIME;  /* mark mtime for update later */
48564      ldir_ptr->i_dirt = DIRTY;
48565      if (new_slots > old_slots) {
48566           ldir_ptr->i_size = (off_t) new_slots * DIR_ENTRY_SIZE;
48567           /* Send the change to disk if the directory is extended. */
48568           if (extended) rw_inode(ldir_ptr, WRITING);
48569      }
48570      return(OK);
48571    }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/pipe.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

48600    /* This file deals with the suspension and revival of processes.  A process can
48601     * be suspended because it wants to read or write from a pipe and can't, or
48602     * because it wants to read or write from a special file and can't.  When a
48603     * process can't continue it is suspended, and revived later when it is able
48604     * to continue.
48605     *
48606     * The entry points into this file are
48607     *   do_pipe:     perform the PIPE system call
48608     *   pipe_check:  check to see that a read or write on a pipe is feasible now
48609     *   suspend:     suspend a process that cannot do a requested read or write
```

```
48610     *   release:     check to see if a suspended process can be released and do it
48611     *   revive:      mark a suspended process as able to run again
48612     *   do_unpause:  a signal has been sent to a process; see if it suspended
48613     */
48614
48615    #include "fs.h"
48616    #include <fcntl.h>
48617    #include <signal.h>
48618    #include <minix/boot.h>
48619    #include <minix/callnr.h>
48620    #include <minix/com.h>
48621    #include "dev.h"
48622    #include "file.h"
48623    #include "fproc.h"
48624    #include "inode.h"
48625    #include "param.h"
48626
48627    PRIVATE message mess;
48628
48629    /*===========================================================================*
48630     *                                do_pipe                                    *
48631     *===========================================================================*/
48632    PUBLIC int do_pipe()
48633    {
48634    /* Perform the pipe(fil_des) system call. */
48635
48636      register struct fproc *rfp;
48637      register struct inode *rip;
48638      int r;
48639      struct filp *fil_ptr0, *fil_ptr1;
48640      int fil_des[2];               /* reply goes here */
48641
48642      /* Acquire two file descriptors. */
48643      rfp = fp;
48644      if ( (r = get_fd(0, R_BIT, &fil_des[0], &fil_ptr0)) != OK) return(r);
48645      rfp->fp_filp[fil_des[0]] = fil_ptr0;
48646      fil_ptr0->filp_count = 1;
48647      if ( (r = get_fd(0, W_BIT, &fil_des[1], &fil_ptr1)) != OK) {
48648            rfp->fp_filp[fil_des[0]] = NIL_FILP;
48649            fil_ptr0->filp_count = 0;
48650            return(r);
48651      }
48652      rfp->fp_filp[fil_des[1]] = fil_ptr1;
48653      fil_ptr1->filp_count = 1;
48654
48655      /* Make the inode on the pipe device. */
48656      if ( (rip = alloc_inode(PIPE_DEV, I_REGULAR) ) == NIL_INODE) {
48657            rfp->fp_filp[fil_des[0]] = NIL_FILP;
48658            fil_ptr0->filp_count = 0;
48659            rfp->fp_filp[fil_des[1]] = NIL_FILP;
48660            fil_ptr1->filp_count = 0;
48661            return(err_code);
48662      }
48663
48664      if (read_only(rip) != OK) panic("pipe device is read only", NO_NUM);
48665
48666      rip->i_pipe = I_PIPE;
48667      rip->i_mode &= ~I_REGULAR;
48668      rip->i_mode |= I_NAMED_PIPE;  /* pipes and FIFOs have this bit set */
48669      fil_ptr0->filp_ino = rip;
```

```
48670       fil_ptr0->filp_flags = O_RDONLY;
48671       dup_inode(rip);               /* for double usage */
48672       fil_ptr1->filp_ino = rip;
48673       fil_ptr1->filp_flags = O_WRONLY;
48674       rw_inode(rip, WRITING);       /* mark inode as allocated */
48675       reply_i1 = fil_des[0];
48676       reply_i2 = fil_des[1];
48677       rip->i_update = ATIME | CTIME | MTIME;
48678       return(OK);
48679     }


48682     /*===========================================================================*
48683      *                              pipe_check                                   *
48684      *===========================================================================*/
48685     PUBLIC int pipe_check(rip, rw_flag, oflags, bytes, position, canwrite)
48686     register struct inode *rip;     /* the inode of the pipe */
48687     int rw_flag;                    /* READING or WRITING */
48688     int oflags;                     /* flags set by open or fcntl */
48689     register int bytes;             /* bytes to be read or written (all chunks) */
48690     register off_t position;        /* current file position */
48691     int *canwrite;                  /* return: number of bytes we can write */
48692     {
48693     /* Pipes are a little different.  If a process reads from an empty pipe for
48694      * which a writer still exists, suspend the reader.  If the pipe is empty
48695      * and there is no writer, return 0 bytes.  If a process is writing to a
48696      * pipe and no one is reading from it, give a broken pipe error.
48697      */
48698
48699       int r = 0;
48700
48701       /* If reading, check for empty pipe. */
48702       if (rw_flag == READING) {
48703             if (position >= rip->i_size) {
48704                     /* Process is reading from an empty pipe. */
48705                     if (find_filp(rip, W_BIT) != NIL_FILP) {
48706                             /* Writer exists */
48707                             if (oflags & O_NONBLOCK)
48708                                     r = EAGAIN;
48709                             else
48710                                     suspend(XPIPE); /* block reader */
48711
48712                             /* If need be, activate sleeping writers. */
48713                             if (susp_count > 0) release(rip, WRITE, susp_count);
48714                     }
48715                     return(r);
48716             }
48717       } else {
48718             /* Process is writing to a pipe. */
48719     /*      if (bytes > PIPE_SIZE) return(EFBIG); */
48720             if (find_filp(rip, R_BIT) == NIL_FILP) {
48721                     /* Tell kernel to generate a SIGPIPE signal. */
48722                     sys_kill((int)(fp - fproc), SIGPIPE);
48723                     return(EPIPE);
48724             }
48725
48726             if (position + bytes > PIPE_SIZE) {
48727                     if ((oflags & O_NONBLOCK) && bytes < PIPE_SIZE)
48728                             return(EAGAIN);
48729                     else if ((oflags & O_NONBLOCK) && bytes > PIPE_SIZE) {
```

```
48730                           if ( (*canwrite = (PIPE_SIZE - position)) > 0)  {
48731                                   /* Do a partial write. Need to wakeup reader */
48732                                   release(rip, READ, susp_count);
48733                                   return(1);
48734                           } else {
48735                                   return(EAGAIN);
48736                           }
48737                       }
48738                   if (bytes > PIPE_SIZE) {
48739                           if ((*canwrite = PIPE_SIZE - position) > 0) {
48740                                   /* Do a partial write. Need to wakeup reader
48741                                    * since we'll suspend ourself in read_write()
48742                                    */
48743                                   release(rip, READ, susp_count);
48744                                   return(1);
48745                           }
48746                   }
48747                   suspend(XPIPE); /* stop writer -- pipe full */
48748                   return(0);
48749           }
48750
48751           /* Writing to an empty pipe.  Search for suspended reader. */
48752           if (position == 0) release(rip, READ, susp_count);
48753     }
48754
48755     *canwrite = 0;
48756     return(1);
48757   }


48760   /*===========================================================================*
48761    *                                suspend                                    *
48762    *===========================================================================*/
48763   PUBLIC void suspend(task)
48764   int task;                       /* who is proc waiting for? (PIPE = pipe) */
48765   {
48766   /* Take measures to suspend the processing of the present system call.
48767    * Store the parameters to be used upon resuming in the process table.
48768    * (Actually they are not used when a process is waiting for an I/O device,
48769    * but they are needed for pipes, and it is not worth making the distinction.)
48770    */
48771
48772     if (task == XPIPE || task == XPOPEN) susp_count++;/* #procs susp'ed on pipe*/
48773     fp->fp_suspended = SUSPENDED;
48774     fp->fp_fd = fd << 8 | fs_call;
48775     fp->fp_task = -task;
48776     if (task == XLOCK) {
48777           fp->fp_buffer = (char *) name1; /*  third arg to fcntl() */
48778           fp->fp_nbytes =request;         /* second arg to fcntl() */
48779     } else {
48780           fp->fp_buffer = buffer;         /* for reads and writes */
48781           fp->fp_nbytes = nbytes;
48782     }
48783     dont_reply = TRUE;            /* do not send caller a reply message now */
48784   }


48787   /*===========================================================================*
48788    *                                release                                    *
48789    *===========================================================================*/
```

```
48790   PUBLIC void release(ip, call_nr, count)
48791   register struct inode *ip;      /* inode of pipe */
48792   int call_nr;                    /* READ, WRITE, OPEN or CREAT */
48793   int count;                      /* max number of processes to release */
48794   {
48795   /* Check to see if any process is hanging on the pipe whose inode is in 'ip'.
48796    * If one is, and it was trying to perform the call indicated by 'call_nr',
48797    * release it.
48798    */
48799
48800     register struct fproc *rp;
48801
48802     /* Search the proc table. */
48803     for (rp = &fproc[0]; rp < &fproc[NR_PROCS]; rp++) {
48804           if (rp->fp_suspended == SUSPENDED &&
48805                           rp->fp_revived == NOT_REVIVING &&
48806                           (rp->fp_fd & BYTE) == call_nr &&
48807                           rp->fp_filp[rp->fp_fd>>8]->filp_ino == ip) {
48808                   revive((int)(rp - fproc), 0);
48809                   susp_count--;   /* keep track of who is suspended */
48810                   if (--count == 0) return;
48811           }
48812     }
48813   }


48816   /*===========================================================================*
48817    *                                revive                                     *
48818    *===========================================================================*/
48819   PUBLIC void revive(proc_nr, bytes)
48820   int proc_nr;                    /* process to revive */
48821   int bytes;                      /* if hanging on task, how many bytes read */
48822   {
48823   /* Revive a previously blocked process. When a process hangs on tty, this
48824    * is the way it is eventually released.
48825    */
48826
48827     register struct fproc *rfp;
48828     register int task;
48829
48830     if (proc_nr < 0 || proc_nr >= NR_PROCS) panic("revive err", proc_nr);
48831     rfp = &fproc[proc_nr];
48832     if (rfp->fp_suspended == NOT_SUSPENDED || rfp->fp_revived == REVIVING)return;
48833
48834     /* The 'reviving' flag only applies to pipes.  Processes waiting for TTY get
48835      * a message right away.  The revival process is different for TTY and pipes.
48836      * For TTY revival, the work is already done, for pipes it is not: the proc
48837      * must be restarted so it can try again.
48838      */
48839     task = -rfp->fp_task;
48840     if (task == XPIPE || task == XLOCK) {
48841           /* Revive a process suspended on a pipe or lock. */
48842           rfp->fp_revived = REVIVING;
48843           reviving++;             /* process was waiting on pipe or lock */
48844     } else {
48845           rfp->fp_suspended = NOT_SUSPENDED;
48846           if (task == XPOPEN) /* process blocked in open or create */
48847                   reply(proc_nr, rfp->fp_fd>>8);
48848           else {
48849                   /* Revive a process suspended on TTY or other device. */
```

```
48850                   rfp->fp_nbytes = bytes; /*pretend it wants only what there is*/
48851                   reply(proc_nr, bytes);  /* unblock the process */
48852           }
48853     }
48854   }


48857   /*===========================================================================*
48858    *                              do_unpause                                   *
48859    *===========================================================================*/
48860   PUBLIC int do_unpause()
48861   {
48862   /* A signal has been sent to a user who is paused on the file system.
48863    * Abort the system call with the EINTR error message.
48864    */
48865
48866     register struct fproc *rfp;
48867     int proc_nr, task, fild;
48868     struct filp *f;
48869     dev_t dev;
48870
48871     if (who > MM_PROC_NR) return(EPERM);
48872     proc_nr = pro;
48873     if (proc_nr < 0 || proc_nr >= NR_PROCS) panic("unpause err 1", proc_nr);
48874     rfp = &fproc[proc_nr];
48875     if (rfp->fp_suspended == NOT_SUSPENDED) return(OK);
48876     task = -rfp->fp_task;
48877
48878     switch(task) {
48879           case XPIPE:             /* process trying to read or write a pipe */
48880                   break;
48881
48882           case XOPEN:             /* process trying to open a special file */
48883                   panic ("fs/do_unpause called with XOPEN\n", NO_NUM);
48884
48885           case XLOCK:             /* process trying to set a lock with FCNTL */
48886                   break;
48887
48888           case XPOPEN:            /* process trying to open a fifo */
48889                   break;
48890
48891           default:                /* process trying to do device I/O (e.g. tty)*/
48892                   fild = (rfp->fp_fd >> 8) & BYTE;/* extract file descriptor */
48893                   if (fild < 0 || fild >= OPEN_MAX)panic("unpause err 2",NO_NUM);
48894                   f = rfp->fp_filp[fild];
48895                   dev = (dev_t) f->filp_ino->i_zone[0];   /* device hung on */
48896                   mess.TTY_LINE = (dev >> MINOR) & BYTE;
48897                   mess.PROC_NR = proc_nr;
48898
48899                   /* Tell kernel R or W. Mode is from current call, not open. */
48900                   mess.COUNT = (rfp->fp_fd & BYTE) == READ ? R_BIT : W_BIT;
48901                   mess.m_type = CANCEL;
48902                   fp = rfp;       /* hack - call_ctty uses fp */
48903                   (*dmap[(dev >> MAJOR) & BYTE].dmap_rw)(task, &mess);
48904     }
48905
48906     rfp->fp_suspended = NOT_SUSPENDED;
48907     reply(proc_nr, EINTR);        /* signal interrupted call */
48908     return(OK);
48909   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/protect.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

49000    /* This file deals with protection in the file system.  It contains the code
49001     * for four system calls that relate to protection.
49002     *
49003     * The entry points into this file are
49004     *   do_chmod:  perform the CHMOD system call
49005     *   do_chown:  perform the CHOWN system call
49006     *   do_umask:  perform the UMASK system call
49007     *   do_access: perform the ACCESS system call
49008     *   forbidden: check to see if a given access is allowed on a given inode
49009     */
49010
49011    #include "fs.h"
49012    #include <unistd.h>
49013    #include <minix/callnr.h>
49014    #include "buf.h"
49015    #include "file.h"
49016    #include "fproc.h"
49017    #include "inode.h"
49018    #include "param.h"
49019    #include "super.h"
49020
49021    /*===========================================================================*
49022     *                                do_chmod                                   *
49023     *===========================================================================*/
49024    PUBLIC int do_chmod()
49025    {
49026    /* Perform the chmod(name, mode) system call. */
49027
49028      register struct inode *rip;
49029      register int r;
49030
49031      /* Temporarily open the file. */
49032      if (fetch_name(name, name_length, M3) != OK) return(err_code);
49033      if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
49034
49035      /* Only the owner or the super_user may change the mode of a file.
49036       * No one may change the mode of a file on a read-only file system.
49037       */
49038      if (rip->i_uid != fp->fp_effuid && !super_user)
49039            r = EPERM;
49040      else
49041            r = read_only(rip);
49042
49043      /* If error, return inode. */
49044      if (r != OK)  {
49045            put_inode(rip);
49046            return(r);
49047      }
49048
49049      /* Now make the change. Clear setgid bit if file is not in caller's grp */
49050      rip->i_mode = (rip->i_mode & ~ALL_MODES) | (mode & ALL_MODES);
49051      if (!super_user && rip->i_gid != fp->fp_effgid)rip->i_mode &= ~I_SET_GID_BIT;
49052      rip->i_update |= CTIME;
49053      rip->i_dirt = DIRTY;
49054
```

```
49055     put_inode(rip);
49056     return(OK);
49057   }


49060   /*===========================================================================*
49061    *                                do_chown                                   *
49062    *===========================================================================*/
49063   PUBLIC int do_chown()
49064   {
49065   /* Perform the chown(name, owner, group) system call. */
49066
49067     register struct inode *rip;
49068     register int r;
49069
49070     /* Temporarily open the file. */
49071     if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
49072     if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
49073
49074     /* Not permitted to change the owner of a file on a read-only file sys. */
49075     r = read_only(rip);
49076     if (r == OK) {
49077           /* FS is R/W.  Whether call is allowed depends on ownership, etc. */
49078           if (super_user) {
49079                   /* The super user can do anything. */
49080                   rip->i_uid = owner;     /* others later */
49081           } else {
49082                   /* Regular users can only change groups of their own files. */
49083                   if (rip->i_uid != fp->fp_effuid) r = EPERM;
49084                   if (rip->i_uid != owner) r = EPERM;     /* no giving away */
49085                   if (fp->fp_effgid != group) r = EPERM;
49086           }
49087     }
49088     if (r == OK) {
49089           rip->i_gid = group;
49090           rip->i_mode &= ~(I_SET_UID_BIT | I_SET_GID_BIT);
49091           rip->i_update |= CTIME;
49092           rip->i_dirt = DIRTY;
49093     }
49094
49095     put_inode(rip);
49096     return(r);
49097   }


49100   /*===========================================================================*
49101    *                                do_umask                                   *
49102    *===========================================================================*/
49103   PUBLIC int do_umask()
49104   {
49105   /* Perform the umask(co_mode) system call. */
49106     register mode_t r;
49107
49108     r = ~fp->fp_umask;            /* set 'r' to complement of old mask */
49109     fp->fp_umask = ~(co_mode & RWX_MODES);
49110     return(r);                    /* return complement of old mask */
49111   }


49114   /*===========================================================================*
```

```
49115    *                              do_access                                      *
49116    *=============================================================================*/
49117   PUBLIC int do_access()
49118   {
49119   /* Perform the access(name, mode) system call. */
49120
49121     struct inode *rip;
49122     register int r;
49123
49124     /* First check to see if the mode is correct. */
49125     if ( (mode & ~(R_OK | W_OK | X_OK)) != 0 && mode != F_OK)
49126           return(EINVAL);
49127
49128     /* Temporarily open the file whose access is to be checked. */
49129     if (fetch_name(name, name_length, M3) != OK) return(err_code);
49130     if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
49131
49132     /* Now check the permissions. */
49133     r = forbidden(rip, (mode_t) mode);
49134     put_inode(rip);
49135     return(r);
49136   }


49139   /*=============================================================================*
49140    *                              forbidden                                      *
49141    *=============================================================================*/
49142   PUBLIC int forbidden(rip, access_desired)
49143   register struct inode *rip;     /* pointer to inode to be checked */
49144   mode_t access_desired;  /* RWX bits */
49145   {
49146   /* Given a pointer to an inode, 'rip', and the access desired, determine
49147    * if the access is allowed, and if not why not.  The routine looks up the
49148    * caller's uid in the 'fproc' table.  If access is allowed, OK is returned
49149    * if it is forbidden, EACCES is returned.
49150    */
49151
49152     register struct inode *old_rip = rip;
49153     register struct super_block *sp;
49154     register mode_t bits, perm_bits;
49155     int r, shift, test_uid, test_gid;
49156
49157     if (rip->i_mount == I_MOUNT)  /* The inode is mounted on. */
49158           for (sp = &super_block[1]; sp < &super_block[NR_SUPERS]; sp++)
49159                   if (sp->s_imount == rip) {
49160                           rip = get_inode(sp->s_dev, ROOT_INODE);
49161                           break;
49162                   } /* if */
49163
49164     /* Isolate the relevant rwx bits from the mode. */
49165     bits = rip->i_mode;
49166     test_uid = (fs_call == ACCESS ? fp->fp_realuid : fp->fp_effuid);
49167     test_gid = (fs_call == ACCESS ? fp->fp_realgid : fp->fp_effgid);
49168     if (test_uid == SU_UID) {
49169           /* Grant read and write permission.  Grant search permission for
49170            * directories.  Grant execute permission (for non-directories) if
49171            * and only if one of the 'X' bits is set.
49172            */
49173           if ( (bits & I_TYPE) == I_DIRECTORY ||
49174                bits & ((X_BIT << 6) | (X_BIT << 3) | X_BIT))
```

```
49175                   perm_bits = R_BIT | W_BIT | X_BIT;
49176           else
49177                   perm_bits = R_BIT | W_BIT;
49178     } else {
49179           if (test_uid == rip->i_uid) shift = 6;          /* owner */
49180           else if (test_gid == rip->i_gid ) shift = 3;    /* group */
49181           else shift = 0;                                 /* other */
49182           perm_bits = (bits >> shift) & (R_BIT | W_BIT | X_BIT);
49183     }
49184
49185     /* If access desired is not a subset of what is allowed, it is refused. */
49186     r = OK;
49187     if ((perm_bits | access_desired) != perm_bits) r = EACCES;
49188
49189     /* Check to see if someone is trying to write on a file system that is
49190      * mounted read-only.
49191      */
49192     if (r == OK)
49193           if (access_desired & W_BIT) r = read_only(rip);
49194
49195     if (rip != old_rip) put_inode(rip);
49196
49197     return(r);
49198   }


49201   /*===========================================================================*
49202    *                               read_only                                   *
49203    *===========================================================================*/
49204   PUBLIC int read_only(ip)
49205   struct inode *ip;               /* ptr to inode whose file sys is to be cked */
49206   {
49207   /* Check to see if the file system on which the inode 'ip' resides is mounted
49208    * read only.  If so, return EROFS, else return OK.
49209    */
49210
49211     register struct super_block *sp;
49212
49213     sp = ip->i_sp;
49214     return(sp->s_rd_only ? EROFS : OK);
49215   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/putk.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

49300   /* FS must occasionally print some message.  It uses the standard library
49301    * routine prink().  (The name "printf" is really a macro defined as "printk").
49302    * Printing is done by calling the TTY task directly, not going through FS.
49303    */
49304
49305   #include "fs.h"
49306   #include <minix/com.h>
49307
49308   #define BUF_SIZE          100   /* print buffer size */
49309
```

```
49310    PRIVATE int buf_count;          /* # characters in the buffer */
49311    PRIVATE char print_buf[BUF_SIZE];       /* output is buffered here */
49312    PRIVATE message putch_msg;      /* used for message to TTY task */
49313
49314    FORWARD _PROTOTYPE( void flush, (void)                                  );
49315
49316    /*===========================================================================*
49317     *                               putk                                        *
49318     *===========================================================================*/
49319    PUBLIC void putk(c)
49320    int c;
49321    {
49322    /* Accumulate another character.  If 0 or buffer full, print it. */
49323
49324      if (c == 0 || buf_count == BUF_SIZE) flush();
49325      if (c == '\n') putk('\r');
49326      if (c != 0) print_buf[buf_count++] = c;
49327    }


49330    /*===========================================================================*
49331     *                               flush                                       *
49332     *===========================================================================*/
49333    PRIVATE void flush()
49334    {
49335    /* Flush the print buffer by calling TTY task. */
49336
49337
49338      if (buf_count == 0) return;
49339      putch_msg.m_type = DEV_WRITE;
49340      putch_msg.PROC_NR  = 1;
49341      putch_msg.TTY_LINE = 0;
49342      putch_msg.ADDRESS  = print_buf;
49343      putch_msg.COUNT = buf_count;
49344      call_task(TTY, &putch_msg);
49345      buf_count = 0;
49346    }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/read.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

49400    /* This file contains the heart of the mechanism used to read (and write)
49401     * files.  Read and write requests are split up into chunks that do not cross
49402     * block boundaries.  Each chunk is then processed in turn.  Reads on special
49403     * files are also detected and handled.
49404     *
49405     * The entry points into this file are
49406     *   do_read:    perform the READ system call by calling read_write
49407     *   read_write: actually do the work of READ and WRITE
49408     *   read_map:   given an inode and file position, look up its zone number
49409     *   rd_indir:   read an entry in an indirect block
49410     *   read_ahead: manage the block read ahead business
49411     */
49412
49413    #include "fs.h"
49414    #include <fcntl.h>
```

```
49415   #include <minix/com.h>
49416   #include "buf.h"
49417   #include "file.h"
49418   #include "fproc.h"
49419   #include "inode.h"
49420   #include "param.h"
49421   #include "super.h"
49422
49423   #define FD_MASK          077    /* max file descriptor is 63 */
49424
49425   PRIVATE message umess;          /* message for asking SYSTASK for user copy */
49426
49427   FORWARD _PROTOTYPE( int rw_chunk, (struct inode *rip, off_t position,
49428                           unsigned off, int chunk, unsigned left, int rw_flag,
49429                           char *buff, int seg, int usr)                   );
49430
49431   /*===========================================================================*
49432    *                               do_read                                     *
49433    *===========================================================================*/
49434   PUBLIC int do_read()
49435   {
49436     return(read_write(READING));
49437   }


49440   /*===========================================================================*
49441    *                               read_write                                  *
49442    *===========================================================================*/
49443   PUBLIC int read_write(rw_flag)
49444   int rw_flag;                    /* READING or WRITING */
49445   {
49446   /* Perform read(fd, buffer, nbytes) or write(fd, buffer, nbytes) call. */
49447
49448     register struct inode *rip;
49449     register struct filp *f;
49450     off_t bytes_left, f_size, position;
49451     unsigned int off, cum_io;
49452     int op, oflags, r, chunk, usr, seg, block_spec, char_spec;
49453     int regular, partial_pipe = 0, partial_cnt = 0;
49454     dev_t dev;
49455     mode_t mode_word;
49456     struct filp *wf;
49457
49458     /* MM loads segments by putting funny things in upper 10 bits of 'fd'. */
49459     if (who == MM_PROC_NR && (fd & (~BYTE)) ) {
49460           usr = (fd >> 8) & BYTE;
49461           seg = (fd >> 6) & 03;
49462           fd &= FD_MASK;          /* get rid of user and segment bits */
49463     } else {
49464           usr = who;              /* normal case */
49465           seg = D;
49466     }
49467
49468     /* If the file descriptor is valid, get the inode, size and mode. */
49469     if (nbytes < 0) return(EINVAL);
49470     if ((f = get_filp(fd)) == NIL_FILP) return(err_code);
49471     if (((f->filp_mode) & (rw_flag == READING ? R_BIT : W_BIT)) == 0) {
49472           return(f->filp_mode == FILP_CLOSED ? EIO : EBADF);
49473     }
49474     if (nbytes == 0) return(0);   /* so char special files need not check for 0*/
```

```
49475      position = f->filp_pos;
49476      if (position > MAX_FILE_POS) return(EINVAL);
49477      if (position + nbytes < position) return(EINVAL); /* unsigned overflow */
49478      oflags = f->filp_flags;
49479      rip = f->filp_ino;
49480      f_size = rip->i_size;
49481      r = OK;
49482      if (rip->i_pipe == I_PIPE) {
49483            /* fp->fp_cum_io_partial is only nonzero when doing partial writes */
49484            cum_io = fp->fp_cum_io_partial;
49485      } else {
49486            cum_io = 0;
49487      }
49488      op = (rw_flag == READING ? DEV_READ : DEV_WRITE);
49489      mode_word = rip->i_mode & I_TYPE;
49490      regular = mode_word == I_REGULAR || mode_word == I_NAMED_PIPE;
49491
49492      char_spec = (mode_word == I_CHAR_SPECIAL ? 1 : 0);
49493      block_spec = (mode_word == I_BLOCK_SPECIAL ? 1 : 0);
49494      if (block_spec) f_size = LONG_MAX;
49495      rdwt_err = OK;                /* set to EIO if disk error occurs */
49496
49497      /* Check for character special files. */
49498      if (char_spec) {
49499            dev = (dev_t) rip->i_zone[0];
49500            r = dev_io(op, oflags & O_NONBLOCK, dev, position, nbytes, who,buffer);
49501            if (r >= 0) {
49502                    cum_io = r;
49503                    position += r;
49504                    r = OK;
49505            }
49506      } else {
49507            if (rw_flag == WRITING && block_spec == 0) {
49508                    /* Check in advance to see if file will grow too big. */
49509                    if (position > rip->i_sp->s_max_size - nbytes) return(EFBIG);
49510
49511                    /* Check for O_APPEND flag. */
49512                    if (oflags & O_APPEND) position = f_size;
49513
49514                    /* Clear the zone containing present EOF if hole about
49515                     * to be created.  This is necessary because all unwritten
49516                     * blocks prior to the EOF must read as zeros.
49517                     */
49518                    if (position > f_size) clear_zone(rip, f_size, 0);
49519            }
49520
49521            /* Pipes are a little different.  Check. */
49522            if (rip->i_pipe == I_PIPE) {
49523                   r = pipe_check(rip,rw_flag,oflags,nbytes,position,&partial_cnt);
49524                   if (r <= 0) return(r);
49525            }
49526
49527            if (partial_cnt > 0) partial_pipe = 1;
49528
49529            /* Split the transfer into chunks that don't span two blocks. */
49530            while (nbytes != 0) {
49531                    off = (unsigned int) (position % BLOCK_SIZE);/* offset in blk*/
49532                    if (partial_pipe) {  /* pipes only */
49533                            chunk = MIN(partial_cnt, BLOCK_SIZE - off);
49534                    } else
```

```
49535                         chunk = MIN(nbytes, BLOCK_SIZE - off);
49536                 if (chunk < 0) chunk = BLOCK_SIZE - off;
49537
49538                 if (rw_flag == READING) {
49539                         bytes_left = f_size - position;
49540                         if (position >= f_size) break;  /* we are beyond EOF */
49541                         if (chunk > bytes_left) chunk = (int) bytes_left;
49542                 }
49543
49544                 /* Read or write 'chunk' bytes. */
49545                 r = rw_chunk(rip, position, off, chunk, (unsigned) nbytes,
49546                                 rw_flag, buffer, seg, usr);
49547                 if (r != OK) break;     /* EOF reached */
49548                 if (rdwt_err < 0) break;
49549
49550                 /* Update counters and pointers. */
49551                 buffer += chunk;        /* user buffer address */
49552                 nbytes -= chunk;        /* bytes yet to be read */
49553                 cum_io += chunk;        /* bytes read so far */
49554                 position += chunk;      /* position within the file */
49555
49556                 if (partial_pipe) {
49557                         partial_cnt -= chunk;
49558                         if (partial_cnt <= 0)  break;
49559                 }
49560         }
49561     }
49562
49563     /* On write, update file size and access time. */
49564     if (rw_flag == WRITING) {
49565         if (regular || mode_word == I_DIRECTORY) {
49566                 if (position > f_size) rip->i_size = position;
49567         }
49568     } else {
49569         if (rip->i_pipe == I_PIPE && position >= rip->i_size) {
49570                 /* Reset pipe pointers. */
49571                 rip->i_size = 0;        /* no data left */
49572                 position = 0;           /* reset reader(s) */
49573                 if ( (wf = find_filp(rip, W_BIT)) != NIL_FILP) wf->filp_pos =0;
49574         }
49575     }
49576     f->filp_pos = position;
49577
49578     /* Check to see if read-ahead is called for, and if so, set it up. */
49579     if (rw_flag == READING && rip->i_seek == NO_SEEK && position % BLOCK_SIZE== 0
49580                 && (regular || mode_word == I_DIRECTORY)) {
49581         rdahed_inode = rip;
49582         rdahedpos = position;
49583     }
49584     rip->i_seek = NO_SEEK;
49585
49586     if (rdwt_err != OK) r = rdwt_err;     /* check for disk error */
49587     if (rdwt_err == END_OF_FILE) r = OK;
49588     if (r == OK) {
49589         if (rw_flag == READING) rip->i_update |= ATIME;
49590         if (rw_flag == WRITING) rip->i_update |= CTIME | MTIME;
49591         rip->i_dirt = DIRTY;            /* inode is thus now dirty */
49592         if (partial_pipe) {
49593                 partial_pipe = 0;
49594                         /* partial write on pipe with */
```

```
49595                           /* O_NONBLOCK, return write count */
49596                   if (!(oflags & O_NONBLOCK)) {
49597                           fp->fp_cum_io_partial = cum_io;
49598                           suspend(XPIPE); /* partial write on pipe with */
49599                           return(0);      /* nbyte > PIPE_SIZE - non-atomic */
49600                   }
49601           }
49602           fp->fp_cum_io_partial = 0;
49603           return(cum_io);
49604     } else {
49605           return(r);
49606     }
49607   }


49610   /*===========================================================================*
49611    *                                rw_chunk                                   *
49612    *===========================================================================*/
49613   PRIVATE int rw_chunk(rip, position, off, chunk, left, rw_flag, buff, seg, usr)
49614   register struct inode *rip;     /* pointer to inode for file to be rd/wr */
49615   off_t position;                 /* position within file to read or write */
49616   unsigned off;                   /* off within the current block */
49617   int chunk;                      /* number of bytes to read or write */
49618   unsigned left;                  /* max number of bytes wanted after position */
49619   int rw_flag;                    /* READING or WRITING */
49620   char *buff;                     /* virtual address of the user buffer */
49621   int seg;                        /* T or D segment in user space */
49622   int usr;                        /* which user process */
49623   {
49624   /* Read or write (part of) a block. */
49625
49626     register struct buf *bp;
49627     register int r;
49628     int n, block_spec;
49629     block_t b;
49630     dev_t dev;
49631
49632     block_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
49633     if (block_spec) {
49634           b = position/BLOCK_SIZE;
49635           dev = (dev_t) rip->i_zone[0];
49636     } else {
49637           b = read_map(rip, position);
49638           dev = rip->i_dev;
49639     }
49640
49641     if (!block_spec && b == NO_BLOCK) {
49642           if (rw_flag == READING) {
49643                   /* Reading from a nonexistent block.  Must read as all zeros.*/
49644                   bp = get_block(NO_DEV, NO_BLOCK, NORMAL);    /* get a buffer */
49645                   zero_block(bp);
49646           } else {
49647                   /* Writing to a nonexistent block. Create and enter in inode.*/
49648                   if ((bp= new_block(rip, position)) == NIL_BUF)return(err_code);
49649           }
49650     } else if (rw_flag == READING) {
49651           /* Read and read ahead if convenient. */
49652           bp = rahead(rip, b, position, left);
49653     } else {
49654           /* Normally an existing block to be partially overwritten is first read
```

```
49655             * in.  However, a full block need not be read in.  If it is already in
49656             * the cache, acquire it, otherwise just acquire a free buffer.
49657             */
49658            n = (chunk == BLOCK_SIZE ? NO_READ : NORMAL);
49659            if (!block_spec && off == 0 && position >= rip->i_size) n = NO_READ;
49660            bp = get_block(dev, b, n);
49661        }
49662
49663        /* In all cases, bp now points to a valid buffer. */
49664        if (rw_flag == WRITING && chunk != BLOCK_SIZE && !block_spec &&
49665                                            position >= rip->i_size && off == 0) {
49666            zero_block(bp);
49667        }
49668        if (rw_flag == READING) {
49669            /* Copy a chunk from the block buffer to user space. */
49670            r = sys_copy(FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
49671                         usr, seg, (phys_bytes) buff,
49672                         (phys_bytes) chunk);
49673        } else {
49674            /* Copy a chunk from user space to the block buffer. */
49675            r = sys_copy(usr, seg, (phys_bytes) buff,
49676                         FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
49677                         (phys_bytes) chunk);
49678            bp->b_dirt = DIRTY;
49679        }
49680        n = (off + chunk == BLOCK_SIZE ? FULL_DATA_BLOCK : PARTIAL_DATA_BLOCK);
49681        put_block(bp, n);
49682        return(r);
49683    }


49686    /*===========================================================================*
49687     *                               read_map                                    *
49688     *===========================================================================*/
49689    PUBLIC block_t read_map(rip, position)
49690    register struct inode *rip;     /* ptr to inode to map from */
49691    off_t position;                 /* position in file whose blk wanted */
49692    {
49693    /* Given an inode and a position within the corresponding file, locate the
49694     * block (not zone) number in which that position is to be found and return it.
49695     */
49696
49697      register struct buf *bp;
49698      register zone_t z;
49699      int scale, boff, dzones, nr_indirects, index, zind, ex;
49700      block_t b;
49701      long excess, zone, block_pos;
49702
49703      scale = rip->i_sp->s_log_zone_size;   /* for block-zone conversion */
49704      block_pos = position/BLOCK_SIZE;      /* relative blk # in file */
49705      zone = block_pos >> scale;    /* position's zone */
49706      boff = (int) (block_pos - (zone << scale) ); /* relative blk # within zone */
49707      dzones = rip->i_ndzones;
49708      nr_indirects = rip->i_nindirs;
49709
49710      /* Is 'position' to be found in the inode itself? */
49711      if (zone < dzones) {
49712            zind = (int) zone;      /* index should be an int */
49713            z = rip->i_zone[zind];
49714            if (z == NO_ZONE) return(NO_BLOCK);
```

```
49715           b = ((block_t) z << scale) + boff;
49716           return(b);
49717      }
49718
49719      /* It is not in the inode, so it must be single or double indirect. */
49720      excess = zone - dzones;       /* first Vx_NR_DZONES don't count */
49721
49722      if (excess < nr_indirects) {
49723           /* 'position' can be located via the single indirect block. */
49724           z = rip->i_zone[dzones];
49725      } else {
49726           /* 'position' can be located via the double indirect block. */
49727           if ( (z = rip->i_zone[dzones+1]) == NO_ZONE) return(NO_BLOCK);
49728           excess -= nr_indirects;               /* single indir doesn't count*/
49729           b = (block_t) z << scale;
49730           bp = get_block(rip->i_dev, b, NORMAL);  /* get double indirect block */
49731           index = (int) (excess/nr_indirects);
49732           z = rd_indir(bp, index);              /* z= zone for single*/
49733           put_block(bp, INDIRECT_BLOCK);        /* release double ind block */
49734           excess = excess % nr_indirects;       /* index into single ind blk */
49735      }
49736
49737      /* 'z' is zone num for single indirect block; 'excess' is index into it. */
49738      if (z == NO_ZONE) return(NO_BLOCK);
49739      b = (block_t) z << scale;                  /* b is blk # for single ind */
49740      bp = get_block(rip->i_dev, b, NORMAL);     /* get single indirect block */
49741      ex = (int) excess;                         /* need an integer */
49742      z = rd_indir(bp, ex);                      /* get block pointed to */
49743      put_block(bp, INDIRECT_BLOCK);             /* release single indir blk */
49744      if (z == NO_ZONE) return(NO_BLOCK);
49745      b = ((block_t) z << scale) + boff;
49746      return(b);
49747    }


49750    /*===========================================================================*
49751     *                               rd_indir                                    *
49752     *===========================================================================*/
49753    PUBLIC zone_t rd_indir(bp, index)
49754    struct buf *bp;               /* pointer to indirect block */
49755    int index;                    /* index into *bp */
49756    {
49757    /* Given a pointer to an indirect block, read one entry.  The reason for
49758     * making a separate routine out of this is that there are four cases:
49759     * V1 (IBM and 68000), and V2 (IBM and 68000).
49760     */
49761
49762      struct super_block *sp;
49763      zone_t zone;                 /* V2 zones are longs (shorts in V1) */
49764
49765      sp = get_super(bp->b_dev);    /* need super block to find file sys type */
49766
49767      /* read a zone from an indirect block */
49768      if (sp->s_version == V1)
49769           zone = (zone_t) conv2(sp->s_native, (int)  bp->b_v1_ind[index]);
49770      else
49771           zone = (zone_t) conv4(sp->s_native, (long) bp->b_v2_ind[index]);
49772
49773      if (zone != NO_ZONE &&
49774                   (zone < (zone_t) sp->s_firstdatazone || zone >= sp->s_zones)) {
```

```
49775           printf("Illegal zone number %ld in indirect block, index %d\n",
49776                   (long) zone, index);
49777           panic("check file system", NO_NUM);
49778     }
49779     return(zone);
49780   }


49783   /*===========================================================================*
49784    *                              read_ahead                                   *
49785    *===========================================================================*/
49786   PUBLIC void read_ahead()
49787   {
49788   /* Read a block into the cache before it is needed. */
49789
49790     register struct inode *rip;
49791     struct buf *bp;
49792     block_t b;
49793
49794     rip = rdahed_inode;           /* pointer to inode to read ahead from */
49795     rdahed_inode = NIL_INODE;     /* turn off read ahead */
49796     if ( (b = read_map(rip, rdahedpos)) == NO_BLOCK) return;      /* at EOF */
49797     bp = rahead(rip, b, rdahedpos, BLOCK_SIZE);
49798     put_block(bp, PARTIAL_DATA_BLOCK);
49799   }


49802   /*===========================================================================*
49803    *                              rahead                                       *
49804    *===========================================================================*/
49805   PUBLIC struct buf *rahead(rip, baseblock, position, bytes_ahead)
49806   register struct inode *rip;     /* pointer to inode for file to be read */
49807   block_t baseblock;              /* block at current position */
49808   off_t position;                 /* position within file */
49809   unsigned bytes_ahead;           /* bytes beyond position for immediate use */
49810   {
49811   /* Fetch a block from the cache or the device.  If a physical read is
49812    * required, prefetch as many more blocks as convenient into the cache.
49813    * This usually covers bytes_ahead and is at least BLOCKS_MINIMUM.
49814    * The device driver may decide it knows better and stop reading at a
49815    * cylinder boundary (or after an error).  Rw_scattered() puts an optional
49816    * flag on all reads to allow this.
49817    */
49818
49819   /* Minimum number of blocks to prefetch. */
49820   # define BLOCKS_MINIMUM         (NR_BUFS < 50 ? 18 : 32)
49821
49822     int block_spec, scale, read_q_size;
49823     unsigned int blocks_ahead, fragment;
49824     block_t block, blocks_left;
49825     off_t ind1_pos;
49826     dev_t dev;
49827     struct buf *bp;
49828     static struct buf *read_q[NR_BUFS];
49829
49830     block_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
49831     if (block_spec) {
49832           dev = (dev_t) rip->i_zone[0];
49833     } else {
49834           dev = rip->i_dev;
```

```
49835       }
49836
49837       block = baseblock;
49838       bp = get_block(dev, block, PREFETCH);
49839       if (bp->b_dev != NO_DEV) return(bp);
49840
49841       /* The best guess for the number of blocks to prefetch:  A lot.
49842        * It is impossible to tell what the device looks like, so we don't even
49843        * try to guess the geometry, but leave it to the driver.
49844        *
49845        * The floppy driver can read a full track with no rotational delay, and it
49846        * avoids reading partial tracks if it can, so handing it enough buffers to
49847        * read two tracks is perfect.  (Two, because some diskette types have
49848        * an odd number of sectors per track, so a block may span tracks.)
49849        *
49850        * The disk drivers don't try to be smart.  With todays disks it is
49851        * impossible to tell what the real geometry looks like, so it is best to
49852        * read as much as you can.  With luck the caching on the drive allows
49853        * for a little time to start the next read.
49854        *
49855        * The current solution below is a bit of a hack, it just reads blocks from
49856        * the current file position hoping that more of the file can be found.  A
49857        * better solution must look at the already available zone pointers and
49858        * indirect blocks (but don't call read_map!).
49859        */
49860
49861       fragment = position % BLOCK_SIZE;
49862       position -= fragment;
49863       bytes_ahead += fragment;
49864
49865       blocks_ahead = (bytes_ahead + BLOCK_SIZE - 1) / BLOCK_SIZE;
49866
49867       if (block_spec && rip->i_size == 0) {
49868             blocks_left = NR_IOREQS;
49869       } else {
49870             blocks_left = (rip->i_size - position + BLOCK_SIZE - 1) / BLOCK_SIZE;
49871
49872             /* Go for the first indirect block if we are in its neighborhood. */
49873             if (!block_spec) {
49874                     scale = rip->i_sp->s_log_zone_size;
49875                     ind1_pos = (off_t) rip->i_ndzones * (BLOCK_SIZE << scale);
49876                     if (position <= ind1_pos && rip->i_size > ind1_pos) {
49877                             blocks_ahead++;
49878                             blocks_left++;
49879                     }
49880             }
49881       }
49882
49883       /* No more than the maximum request. */
49884       if (blocks_ahead > NR_IOREQS) blocks_ahead = NR_IOREQS;
49885
49886       /* Read at least the minimum number of blocks, but not after a seek. */
49887       if (blocks_ahead < BLOCKS_MINIMUM && rip->i_seek == NO_SEEK)
49888             blocks_ahead = BLOCKS_MINIMUM;
49889
49890       /* Can't go past end of file. */
49891       if (blocks_ahead > blocks_left) blocks_ahead = blocks_left;
49892
49893       read_q_size = 0;
49894
```

```
49895      /* Acquire block buffers. */
49896      for (;;) {
49897            read_q[read_q_size++] = bp;
49898
49899            if (--blocks_ahead == 0) break;
49900
49901            /* Don't trash the cache, leave 4 free. */
49902            if (bufs_in_use >= NR_BUFS - 4) break;
49903
49904            block++;
49905
49906            bp = get_block(dev, block, PREFETCH);
49907            if (bp->b_dev != NO_DEV) {
49908                    /* Oops, block already in the cache, get out. */
49909                    put_block(bp, FULL_DATA_BLOCK);
49910                    break;
49911            }
49912      }
49913      rw_scattered(dev, read_q, read_q_size, READING);
49914      return(get_block(dev, baseblock, NORMAL));
49915    }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/stadir.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

50000    /* This file contains the code for performing four system calls relating to
50001     * status and directories.
50002     *
50003     * The entry points into this file are
50004     *   do_chdir:  perform the CHDIR system call
50005     *   do_chroot: perform the CHROOT system call
50006     *   do_stat:   perform the STAT system call
50007     *   do_fstat:  perform the FSTAT system call
50008     */
50009
50010    #include "fs.h"
50011    #include <sys/stat.h>
50012    #include "file.h"
50013    #include "fproc.h"
50014    #include "inode.h"
50015    #include "param.h"
50016
50017    FORWARD _PROTOTYPE( int change, (struct inode **iip, char *name_ptr, int len));
50018    FORWARD _PROTOTYPE( int stat_inode, (struct inode *rip, struct filp *fil_ptr,
50019                            char *user_addr)                                );
50020
50021    /*===========================================================================*
50022     *                                do_chdir                                   *
50023     *===========================================================================*/
50024    PUBLIC int do_chdir()
50025    {
50026    /* Change directory.  This function is  also called by MM to simulate a chdir
50027     * in order to do EXEC, etc.  It also changes the root directory, the uids and
50028     * gids, and the umask.  The name of this function is now misleading and the
50029     * cd_flag argument is no longer supported.
```

```
50030     */
50031
50032      int r;
50033      register struct fproc *rfp;
50034
50035      if (who == MM_PROC_NR) {
50036            rfp = &fproc[slot1];
50037            put_inode(fp->fp_rootdir);
50038            dup_inode(fp->fp_rootdir = rfp->fp_rootdir);
50039            put_inode(fp->fp_workdir);
50040            dup_inode(fp->fp_workdir = rfp->fp_workdir);
50041
50042            /* MM uses access() to check permissions.  To make this work, pretend
50043             * that the user's real ids are the same as the user's effective ids.
50044             * FS calls other than access() do not use the real ids, so are not
50045             * affected.
50046             */
50047            fp->fp_realuid =
50048            fp->fp_effuid = rfp->fp_effuid;
50049            fp->fp_realgid =
50050            fp->fp_effgid = rfp->fp_effgid;
50051            fp->fp_umask = rfp->fp_umask;
50052            return(OK);
50053      }
50054
50055      /* Perform the chdir(name) system call. */
50056      r = change(&fp->fp_workdir, name, name_length);
50057      return(r);
50058    }


50061    /*===========================================================================*
50062     *                                do_chroot                                  *
50063     *===========================================================================*/
50064    PUBLIC int do_chroot()
50065    {
50066    /* Perform the chroot(name) system call. */
50067
50068      register int r;
50069
50070      if (!super_user) return(EPERM);       /* only su may chroot() */
50071      r = change(&fp->fp_rootdir, name, name_length);
50072      return(r);
50073    }


50076    /*===========================================================================*
50077     *                                change                                     *
50078     *===========================================================================*/
50079    PRIVATE int change(iip, name_ptr, len)
50080    struct inode **iip;             /* pointer to the inode pointer for the dir */
50081    char *name_ptr;                 /* pointer to the directory name to change to */
50082    int len;                        /* length of the directory name string */
50083    {
50084    /* Do the actual work for chdir() and chroot(). */
50085
50086      struct inode *rip;
50087      register int r;
50088
50089      /* Try to open the new directory. */
```

```
50090     if (fetch_name(name_ptr, len, M3) != OK) return(err_code);
50091     if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
50092
50093     /* It must be a directory and also be searchable. */
50094     if ( (rip->i_mode & I_TYPE) != I_DIRECTORY)
50095           r = ENOTDIR;
50096     else
50097           r = forbidden(rip, X_BIT);      /* check if dir is searchable */
50098
50099     /* If error, return inode. */
50100     if (r != OK) {
50101           put_inode(rip);
50102           return(r);
50103     }
50104
50105     /* Everything is OK.  Make the change. */
50106     put_inode(*iip);              /* release the old directory */
50107     *iip = rip;                   /* acquire the new one */
50108     return(OK);
50109   }


50112   /*===========================================================================*
50113    *                                do_stat                                    *
50114    *===========================================================================*/
50115   PUBLIC int do_stat()
50116   {
50117   /* Perform the stat(name, buf) system call. */
50118
50119     register struct inode *rip;
50120     register int r;
50121
50122     /* Both stat() and fstat() use the same routine to do the real work.  That
50123      * routine expects an inode, so acquire it temporarily.
50124      */
50125     if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
50126     if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
50127     r = stat_inode(rip, NIL_FILP, name2); /* actually do the work.*/
50128     put_inode(rip);               /* release the inode */
50129     return(r);
50130   }


50133   /*===========================================================================*
50134    *                                do_fstat                                   *
50135    *===========================================================================*/
50136   PUBLIC int do_fstat()
50137   {
50138   /* Perform the fstat(fd, buf) system call. */
50139
50140     register struct filp *rfilp;
50141
50142     /* Is the file descriptor valid? */
50143     if ( (rfilp = get_filp(fd)) == NIL_FILP) return(err_code);
50144
50145     return(stat_inode(rfilp->filp_ino, rfilp, buffer));
50146   }


50149   /*===========================================================================*
```

```
50150     *                             stat_inode                                    *
50151     *===========================================================================*/
50152    PRIVATE int stat_inode(rip, fil_ptr, user_addr)
50153    register struct inode *rip;     /* pointer to inode to stat */
50154    struct filp *fil_ptr;           /* filp pointer, supplied by 'fstat' */
50155    char *user_addr;                /* user space address where stat buf goes */
50156    {
50157    /* Common code for stat and fstat system calls. */
50158
50159      struct stat statbuf;
50160      mode_t mo;
50161      int r, s;
50162
50163      /* Update the atime, ctime, and mtime fields in the inode, if need be. */
50164      if (rip->i_update) update_times(rip);
50165
50166      /* Fill in the statbuf struct. */
50167      mo = rip->i_mode & I_TYPE;
50168      s = (mo == I_CHAR_SPECIAL || mo == I_BLOCK_SPECIAL);  /* true iff special */
50169      statbuf.st_dev = rip->i_dev;
50170      statbuf.st_ino = rip->i_num;
50171      statbuf.st_mode = rip->i_mode;
50172      statbuf.st_nlink = rip->i_nlinks & BYTE;
50173      statbuf.st_uid = rip->i_uid;
50174      statbuf.st_gid = rip->i_gid & BYTE;
50175      statbuf.st_rdev = (dev_t) (s ? rip->i_zone[0] : NO_DEV);
50176      statbuf.st_size = rip->i_size;
50177
50178      if (rip->i_pipe == I_PIPE) {
50179            statbuf.st_mode &= ~I_REGULAR;  /* wipe out I_REGULAR bit for pipes */
50180            if (fil_ptr != NIL_FILP && fil_ptr->filp_mode & R_BIT)
50181                  statbuf.st_size -= fil_ptr->filp_pos;
50182      }
50183
50184      statbuf.st_atime = rip->i_atime;
50185      statbuf.st_mtime = rip->i_mtime;
50186      statbuf.st_ctime = rip->i_ctime;
50187
50188      /* Copy the struct to user space. */
50189      r = sys_copy(FS_PROC_NR, D, (phys_bytes) &statbuf,
50190                   who, D, (phys_bytes) user_addr, (phys_bytes) sizeof(statbuf));
50191      return(r);
50192    }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/super.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

50200    /* This file manages the super block table and the related data structures,
50201     * namely, the bit maps that keep track of which zones and which inodes are
50202     * allocated and which are free.  When a new inode or zone is needed, the
50203     * appropriate bit map is searched for a free entry.
50204     *
50205     * The entry points into this file are
50206     *   alloc_bit:       somebody wants to allocate a zone or inode; find one
50207     *   free_bit:        indicate that a zone or inode is available for allocation
50208     *   get_super:       search the 'superblock' table for a device
50209     *   mounted:         tells if file inode is on mounted (or ROOT) file system
```

```
50210     *   read_super:      read a superblock
50211     */
50212
50213    #include "fs.h"
50214    #include <string.h>
50215    #include <minix/boot.h>
50216    #include "buf.h"
50217    #include "inode.h"
50218    #include "super.h"
50219
50220    #define BITCHUNK_BITS   (usizeof(bitchunk_t) * CHAR_BIT)
50221    #define BITS_PER_BLOCK  (BITMAP_CHUNKS * BITCHUNK_BITS)
50222
50223    /*===========================================================================*
50224     *                              alloc_bit                                    *
50225     *===========================================================================*/
50226    PUBLIC bit_t alloc_bit(sp, map, origin)
50227    struct super_block *sp;         /* the filesystem to allocate from */
50228    int map;                        /* IMAP (inode map) or ZMAP (zone map) */
50229    bit_t origin;                   /* number of bit to start searching at */
50230    {
50231    /* Allocate a bit from a bit map and return its bit number. */
50232
50233      block_t start_block;          /* first bit block */
50234      bit_t map_bits;               /* how many bits are there in the bit map? */
50235      unsigned bit_blocks;          /* how many blocks are there in the bit map? */
50236      unsigned block, word, bcount, wcount;
50237      struct buf *bp;
50238      bitchunk_t *wptr, *wlim, k;
50239      bit_t i, b;
50240
50241      if (sp->s_rd_only)
50242            panic("can't allocate bit on read-only filesys.", NO_NUM);
50243
50244      if (map == IMAP) {
50245            start_block = SUPER_BLOCK + 1;
50246            map_bits = sp->s_ninodes + 1;
50247            bit_blocks = sp->s_imap_blocks;
50248      } else {
50249            start_block = SUPER_BLOCK + 1 + sp->s_imap_blocks;
50250            map_bits = sp->s_zones - (sp->s_firstdatazone - 1);
50251            bit_blocks = sp->s_zmap_blocks;
50252      }
50253
50254      /* Figure out where to start the bit search (depends on 'origin'). */
50255      if (origin >= map_bits) origin = 0;   /* for robustness */
50256
50257      /* Locate the starting place. */
50258      block = origin / BITS_PER_BLOCK;
50259      word = (origin % BITS_PER_BLOCK) / BITCHUNK_BITS;
50260
50261      /* Iterate over all blocks plus one, because we start in the middle. */
50262      bcount = bit_blocks + 1;
50263      do {
50264            bp = get_block(sp->s_dev, start_block + block, NORMAL);
50265            wlim = &bp->b_bitmap[BITMAP_CHUNKS];
50266
50267            /* Iterate over the words in block. */
50268            for (wptr = &bp->b_bitmap[word]; wptr < wlim; wptr++) {
50269
```

```
50270                   /* Does this word contain a free bit? */
50271                   if (*wptr == (bitchunk_t) ~0) continue;
50272
50273                   /* Find and allocate the free bit. */
50274                   k = conv2(sp->s_native, (int) *wptr);
50275                   for (i = 0; (k & (1 << i)) != 0; ++i) {}
50276
50277                   /* Bit number from the start of the bit map. */
50278                   b = ((bit_t) block * BITS_PER_BLOCK)
50279                       + (wptr - &bp->b_bitmap[0]) * BITCHUNK_BITS
50280                       + i;
50281
50282                   /* Don't allocate bits beyond the end of the map. */
50283                   if (b >= map_bits) break;
50284
50285                   /* Allocate and return bit number. */
50286                   k |= 1 << i;
50287                   *wptr = conv2(sp->s_native, (int) k);
50288                   bp->b_dirt = DIRTY;
50289                   put_block(bp, MAP_BLOCK);
50290                   return(b);
50291           }
50292           put_block(bp, MAP_BLOCK);
50293           if (++block >= bit_blocks) block = 0;   /* last block, wrap around */
50294           word = 0;
50295     } while (--bcount > 0);
50296     return(NO_BIT);               /* no bit could be allocated */
50297   }


50300   /*===========================================================================*
50301    *                              free_bit                                     *
50302    *===========================================================================*/
50303   PUBLIC void free_bit(sp, map, bit_returned)
50304   struct super_block *sp;         /* the filesystem to operate on */
50305   int map;                        /* IMAP (inode map) or ZMAP (zone map) */
50306   bit_t bit_returned;             /* number of bit to insert into the map */
50307   {
50308   /* Return a zone or inode by turning off its bitmap bit. */
50309
50310     unsigned block, word, bit;
50311     struct buf *bp;
50312     bitchunk_t k, mask;
50313     block_t start_block;
50314
50315     if (sp->s_rd_only)
50316           panic("can't free bit on read-only filesys.", NO_NUM);
50317
50318     if (map == IMAP) {
50319           start_block = SUPER_BLOCK + 1;
50320     } else {
50321           start_block = SUPER_BLOCK + 1 + sp->s_imap_blocks;
50322     }
50323     block = bit_returned / BITS_PER_BLOCK;
50324     word = (bit_returned % BITS_PER_BLOCK) / BITCHUNK_BITS;
50325     bit = bit_returned % BITCHUNK_BITS;
50326     mask = 1 << bit;
50327
50328     bp = get_block(sp->s_dev, start_block + block, NORMAL);
50329
```

```
50330     k = conv2(sp->s_native, (int) bp->b_bitmap[word]);
50331     if (!(k & mask)) {
50332           panic(map == IMAP ? "tried to free unused inode" :
50333                 "tried to free unused block", NO_NUM);
50334     }
50335
50336     k &= ~mask;
50337     bp->b_bitmap[word] = conv2(sp->s_native, (int) k);
50338     bp->b_dirt = DIRTY;
50339
50340     put_block(bp, MAP_BLOCK);
50341   }


50344   /*===========================================================================*
50345    *                              get_super                                    *
50346    *===========================================================================*/
50347   PUBLIC struct super_block *get_super(dev)
50348   dev_t dev;                    /* device number whose super_block is sought */
50349   {
50350   /* Search the superblock table for this device.  It is supposed to be there. */
50351
50352     register struct super_block *sp;
50353
50354     for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
50355           if (sp->s_dev == dev) return(sp);
50356
50357     /* Search failed.  Something wrong. */
50358     panic("can't find superblock for device (in decimal)", (int) dev);
50359
50360     return(NIL_SUPER);          /* to keep the compiler and lint quiet */
50361   }


50364   /*===========================================================================*
50365    *                              mounted                                      *
50366    *===========================================================================*/
50367   PUBLIC int mounted(rip)
50368   register struct inode *rip;     /* pointer to inode */
50369   {
50370   /* Report on whether the given inode is on a mounted (or ROOT) file system. */
50371
50372     register struct super_block *sp;
50373     register dev_t dev;
50374
50375     dev = (dev_t) rip->i_zone[0];
50376     if (dev == ROOT_DEV) return(TRUE);    /* inode is on root file system */
50377
50378     for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
50379           if (sp->s_dev == dev) return(TRUE);
50380
50381     return(FALSE);
50382   }


50385   /*===========================================================================*
50386    *                              read_super                                   *
50387    *===========================================================================*/
50388   PUBLIC int read_super(sp)
50389   register struct super_block *sp; /* pointer to a superblock */
```

```
50390   {
50391   /* Read a superblock. */
50392
50393     register struct buf *bp;
50394     dev_t dev;
50395     int magic;
50396     int version, native;
50397
50398     dev = sp->s_dev;              /* save device (will be overwritten by copy) */
50399     bp = get_block(sp->s_dev, SUPER_BLOCK, NORMAL);
50400     memcpy( (char *) sp, bp->b_data, (size_t) SUPER_SIZE);
50401     put_block(bp, ZUPER_BLOCK);
50402     sp->s_dev = NO_DEV;           /* restore later */
50403     magic = sp->s_magic;          /* determines file system type */
50404
50405     /* Get file system version and type. */
50406     if (magic == SUPER_MAGIC || magic == conv2(BYTE_SWAP, SUPER_MAGIC)) {
50407           version = V1;
50408           native  = (magic == SUPER_MAGIC);
50409     } else if (magic == SUPER_V2 || magic == conv2(BYTE_SWAP, SUPER_V2)) {
50410           version = V2;
50411           native  = (magic == SUPER_V2);
50412     } else {
50413           return(EINVAL);
50414     }
50415
50416     /* If the super block has the wrong byte order, swap the fields; the magic
50417      * number doesn't need conversion. */
50418     sp->s_ninodes =       conv2(native, (int) sp->s_ninodes);
50419     sp->s_nzones =        conv2(native, (int) sp->s_nzones);
50420     sp->s_imap_blocks =   conv2(native, (int) sp->s_imap_blocks);
50421     sp->s_zmap_blocks =   conv2(native, (int) sp->s_zmap_blocks);
50422     sp->s_firstdatazone = conv2(native, (int) sp->s_firstdatazone);
50423     sp->s_log_zone_size = conv2(native, (int) sp->s_log_zone_size);
50424     sp->s_max_size =      conv4(native, sp->s_max_size);
50425     sp->s_zones =         conv4(native, sp->s_zones);
50426
50427     /* In V1, the device size was kept in a short, s_nzones, which limited
50428      * devices to 32K zones.  For V2, it was decided to keep the size as a
50429      * long.  However, just changing s_nzones to a long would not work, since
50430      * then the position of s_magic in the super block would not be the same
50431      * in V1 and V2 file systems, and there would be no way to tell whether
50432      * a newly mounted file system was V1 or V2.  The solution was to introduce
50433      * a new variable, s_zones, and copy the size there.
50434      *
50435      * Calculate some other numbers that depend on the version here too, to
50436      * hide some of the differences.
50437      */
50438     if (version == V1) {
50439           sp->s_zones = sp->s_nzones;     /* only V1 needs this copy */
50440           sp->s_inodes_per_block = V1_INODES_PER_BLOCK;
50441           sp->s_ndzones = V1_NR_DZONES;
50442           sp->s_nindirs = V1_INDIRECTS;
50443     } else {
50444           sp->s_inodes_per_block = V2_INODES_PER_BLOCK;
50445           sp->s_ndzones = V2_NR_DZONES;
50446           sp->s_nindirs = V2_INDIRECTS;
50447     }
50448
50449     sp->s_isearch = 0;            /* inode searches initially start at 0 */
```

```
50450       sp->s_zsearch = 0;            /* zone searches initially start at 0 */
50451       sp->s_version = version;
50452       sp->s_native  = native;
50453
50454       /* Make a few basic checks to see if super block looks reasonable. */
50455       if (sp->s_imap_blocks < 1 || sp->s_zmap_blocks < 1
50456                                     || sp->s_ninodes < 1 || sp->s_zones < 1
50457                                     || (unsigned) sp->s_log_zone_size > 4) {
50458             return(EINVAL);
50459       }
50460       sp->s_dev = dev;              /* restore device number */
50461       return(OK);
50462     }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/table.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

50500     /* This file contains the table used to map system call numbers onto the
50501      * routines that perform them.
50502      */
50503
50504     #define _TABLE
50505
50506     #include "fs.h"
50507     #include <minix/callnr.h>
50508     #include <minix/com.h>
50509     #include "buf.h"
50510     #include "dev.h"
50511     #include "file.h"
50512     #include "fproc.h"
50513     #include "inode.h"
50514     #include "lock.h"
50515     #include "super.h"
50516
50517     PUBLIC _PROTOTYPE (int (*call_vector[NCALLS]), (void) ) = {
50518             no_sys,         /*  0 = unused  */
50519             do_exit,        /*  1 = exit    */
50520             do_fork,        /*  2 = fork    */
50521             do_read,        /*  3 = read    */
50522             do_write,       /*  4 = write   */
50523             do_open,        /*  5 = open    */
50524             do_close,       /*  6 = close   */
50525             no_sys,         /*  7 = wait    */
50526             do_creat,       /*  8 = creat   */
50527             do_link,        /*  9 = link    */
50528             do_unlink,      /* 10 = unlink  */
50529             no_sys,         /* 11 = waitpid */
50530             do_chdir,       /* 12 = chdir   */
50531             do_time,        /* 13 = time    */
50532             do_mknod,       /* 14 = mknod   */
50533             do_chmod,       /* 15 = chmod   */
50534             do_chown,       /* 16 = chown   */
50535             no_sys,         /* 17 = break   */
50536             do_stat,        /* 18 = stat    */
50537             do_lseek,       /* 19 = lseek   */
50538             no_sys,         /* 20 = getpid  */
50539             do_mount,       /* 21 = mount   */
```

```
50540           do_umount,      /* 22 = umount  */
50541           do_set,         /* 23 = setuid  */
50542           no_sys,         /* 24 = getuid  */
50543           do_stime,       /* 25 = stime   */
50544           no_sys,         /* 26 = ptrace  */
50545           no_sys,         /* 27 = alarm   */
50546           do_fstat,       /* 28 = fstat   */
50547           no_sys,         /* 29 = pause   */
50548           do_utime,       /* 30 = utime   */
50549           no_sys,         /* 31 = (stty)  */
50550           no_sys,         /* 32 = (gtty)  */
50551           do_access,      /* 33 = access  */
50552           no_sys,         /* 34 = (nice)  */
50553           no_sys,         /* 35 = (ftime) */
50554           do_sync,        /* 36 = sync    */
50555           no_sys,         /* 37 = kill    */
50556           do_rename,      /* 38 = rename  */
50557           do_mkdir,       /* 39 = mkdir   */
50558           do_unlink,      /* 40 = rmdir   */
50559           do_dup,         /* 41 = dup     */
50560           do_pipe,        /* 42 = pipe    */
50561           do_tims,        /* 43 = times   */
50562           no_sys,         /* 44 = (prof)  */
50563           no_sys,         /* 45 = unused  */
50564           do_set,         /* 46 = setgid  */
50565           no_sys,         /* 47 = getgid  */
50566           no_sys,         /* 48 = (signal)*/
50567           no_sys,         /* 49 = unused  */
50568           no_sys,         /* 50 = unused  */
50569           no_sys,         /* 51 = (acct)  */
50570           no_sys,         /* 52 = (phys)  */
50571           no_sys,         /* 53 = (lock)  */
50572           do_ioctl,       /* 54 = ioctl   */
50573           do_fcntl,       /* 55 = fcntl   */
50574           no_sys,         /* 56 = (mpx)   */
50575           no_sys,         /* 57 = unused  */
50576           no_sys,         /* 58 = unused  */
50577           do_exec,        /* 59 = execve  */
50578           do_umask,       /* 60 = umask   */
50579           do_chroot,      /* 61 = chroot  */
50580           do_setsid,      /* 62 = setsid  */
50581           no_sys,         /* 63 = getpgrp */
50582
50583           no_sys,         /* 64 = KSIG: signals originating in the kernel */
50584           do_unpause,     /* 65 = UNPAUSE */
50585           no_sys,         /* 66 = unused  */
50586           do_revive,      /* 67 = REVIVE  */
50587           no_sys,         /* 68 = TASK_REPLY      */
50588           no_sys,         /* 69 = unused */
50589           no_sys,         /* 70 = unused */
50590           no_sys,         /* 71 = SIGACTION */
50591           no_sys,         /* 72 = SIGSUSPEND */
50592           no_sys,         /* 73 = SIGPENDING */
50593           no_sys,         /* 74 = SIGPROCMASK */
50594           no_sys,         /* 75 = SIGRETURN */
50595           no_sys,         /* 76 = REBOOT */
50596   };
50597
50598
50599   /* Some devices may or may not be there in the next table. */
```

```
50600   #define DT(enable, open, rw, close, task) \
50601           { (enable ? (open) : no_dev), (enable ? (rw) : no_dev), \
50602             (enable ? (close) : no_dev), (enable ? (task) : 0) },
50603
50604   /* The order of the entries here determines the mapping between major device
50605    * numbers and tasks.  The first entry (major device 0) is not used.  The
50606    * next entry is major device 1, etc.  Character and block devices can be
50607    * intermixed at random.  If this ordering is changed, the devices in
50608    * <include/minix/boot.h> must be changed to correspond to the new values.
50609    * Note that the major device numbers used in /dev are NOT the same as the
50610    * task numbers used inside the kernel (as defined in <include/minix/com.h>).
50611    * Also note that if /dev/mem is changed from 1, NULL_MAJOR must be changed
50612    * in <include/minix/com.h>.
50613    */
50614   PUBLIC struct dmap dmap[] = {
50615   /*  ?   Open       Read/Write   Close       Task #      Device  File
50616       -   ----       ----------   -----       -------     ------  ----       */
50617     DT(1, no_dev,    no_dev,      no_dev,     0)           /* 0 = not used   */
50618     DT(1, dev_opcl,  call_task,   dev_opcl,   MEM)         /* 1 = /dev/mem   */
50619     DT(1, dev_opcl,  call_task,   dev_opcl,   FLOPPY)      /* 2 = /dev/fd0   */
50620     DT(ENABLE_WINI,
50621           dev_opcl,  call_task,   dev_opcl,   WINCHESTER)  /* 3 = /dev/hd0   */
50622     DT(1, tty_open,  call_task,   dev_opcl,   TTY)         /* 4 = /dev/tty00 */
50623     DT(1, ctty_open, call_ctty,   ctty_close, TTY)         /* 5 = /dev/tty   */
50624     DT(1, dev_opcl,  call_task,   dev_opcl,    PRINTER)     /* 6 = /dev/lp    */
50625
50626   #if (MACHINE == IBM_PC)
50627     DT(ENABLE_NETWORKING,
50628           net_open,  call_task,   dev_opcl,   INET_PROC_NR)/* 7 = /dev/ip    */
50629     DT(ENABLE_CDROM,
50630           dev_opcl,  call_task,   dev_opcl,   CDROM)       /* 8 = /dev/cd0   */
50631     DT(0, 0,         0,          0,          0)           /* 9 = not used   */
50632     DT(ENABLE_SCSI,
50633           dev_opcl,  call_task,   dev_opcl,   SCSI)        /*10 = /dev/sd0   */
50634     DT(0, 0,         0,          0,          0)           /*11 = not used   */
50635     DT(0, 0,         0,          0,          0)           /*12 = not used   */
50636     DT(ENABLE_AUDIO,
50637           dev_opcl,  call_task,   dev_opcl,   AUDIO)       /*13 = /dev/audio */
50638     DT(ENABLE_AUDIO,
50639           dev_opcl,  call_task,   dev_opcl,   MIXER)       /*14 = /dev/mixer */
50640   #endif /* IBM_PC */
50641
50642   #if (MACHINE == ATARI)
50643     DT(ENABLE_SCSI,
50644           dev_opcl,  call_task,   dev_opcl,   SCSI)        /* 7 = /dev/hdscsi0 */
50645   #endif
50646   };
50647
50648   PUBLIC int max_major = sizeof(dmap)/sizeof(struct dmap);
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/time.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

50700    /* This file takes care of those system calls that deal with time.
50701     *
50702     * The entry points into this file are
50703     *   do_utime:  perform the UTIME system call
50704     *   do_time:   perform the TIME system call
50705     *   do_stime:  perform the STIME system call
50706     *   do_tims:   perform the TIMES system call
50707     */
50708
50709    #include "fs.h"
50710    #include <minix/callnr.h>
50711    #include <minix/com.h>
50712    #include "file.h"
50713    #include "fproc.h"
50714    #include "inode.h"
50715    #include "param.h"
50716
50717    PRIVATE message clock_mess;
50718
50719    /*===========================================================================*
50720     *                                do_utime                                   *
50721     *===========================================================================*/
50722    PUBLIC int do_utime()
50723    {
50724    /* Perform the utime(name, timep) system call. */
50725
50726      register struct inode *rip;
50727      register int len, r;
50728
50729      /* Adjust for case of NULL 'timep'. */
50730      len = utime_length;
50731      if (len == 0) len = m.m2_i2;
50732
50733      /* Temporarily open the file. */
50734      if (fetch_name(utime_file, len, M1) != OK) return(err_code);
50735      if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
50736
50737      /* Only the owner of a file or the super_user can change its time. */
50738      r = OK;
50739      if (rip->i_uid != fp->fp_effuid && !super_user) r = EPERM;
50740      if (utime_length == 0 && r != OK) r = forbidden(rip, W_BIT);
50741      if (read_only(rip) != OK) r = EROFS;  /* not even su can touch if R/O */
50742      if (r == OK) {
50743            if (utime_length == 0) {
50744                    rip->i_atime = clock_time();
50745                    rip->i_mtime = rip->i_atime;
50746            } else {
50747                    rip->i_atime = utime_actime;
50748                    rip->i_mtime = utime_modtime;
50749            }
50750            rip->i_update = CTIME;  /* discard any stale ATIME and MTIME flags */
50751            rip->i_dirt = DIRTY;
50752      }
50753
50754      put_inode(rip);
```

```
50755     return(r);
50756   }


50759   /*===========================================================================*
50760    *                                do_time                                    *
50761    *===========================================================================*/
50762   PUBLIC int do_time()
50763
50764   {
50765   /* Perform the time(tp) system call. */
50766
50767     reply_l1 = clock_time();      /* return time in seconds */
50768     return(OK);
50769   }


50772   /*===========================================================================*
50773    *                                do_stime                                   *
50774    *===========================================================================*/
50775   PUBLIC int do_stime()
50776   {
50777   /* Perform the stime(tp) system call. */
50778
50779     register int k;
50780
50781     if (!super_user) return(EPERM);
50782     clock_mess.m_type = SET_TIME;
50783     clock_mess.NEW_TIME = (long) tp;
50784     if ( (k = sendrec(CLOCK, &clock_mess)) != OK) panic("do_stime error", k);
50785     return(OK);
50786   }


50789   /*===========================================================================*
50790    *                                do_tims                                    *
50791    *===========================================================================*/
50792   PUBLIC int do_tims()
50793   {
50794   /* Perform the times(buffer) system call. */
50795
50796     clock_t t[5];
50797
50798     sys_times(who, t);
50799     reply_t1 = t[0];
50800     reply_t2 = t[1];
50801     reply_t3 = t[2];
50802     reply_t4 = t[3];
50803     reply_t5 = t[4];
50804     return(OK);
50805   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/utility.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

50900    /* This file contains a few general purpose utility routines.
50901     *
50902     * The entry points into this file are
50903     *   clock_time:  ask the clock task for the real time
50904     *   copy:        copy a block of data
50905     *   fetch_name:  go get a path name from user space
50906     *   no_sys:      reject a system call that FS does not handle
50907     *   panic:       something awful has occurred;  MINIX cannot continue
50908     *   conv2:       do byte swapping on a 16-bit int
50909     *   conv4:       do byte swapping on a 32-bit long
50910     */
50911
50912    #include "fs.h"
50913    #include <minix/com.h>
50914    #include <minix/boot.h>
50915    #include <unistd.h>
50916    #include "buf.h"
50917    #include "file.h"
50918    #include "fproc.h"
50919    #include "inode.h"
50920    #include "param.h"
50921
50922    PRIVATE int panicking;          /* inhibits recursive panics during sync */
50923    PRIVATE message clock_mess;
50924
50925    /*===========================================================================*
50926     *                              clock_time                                   *
50927     *===========================================================================*/
50928    PUBLIC time_t clock_time()
50929    {
50930    /* This routine returns the time in seconds since 1.1.1970.  MINIX is an
50931     * astrophysically naive system that assumes the earth rotates at a constant
50932     * rate and that such things as leap seconds do not exist.
50933     */
50934
50935      register int k;
50936
50937      clock_mess.m_type = GET_TIME;
50938      if ( (k = sendrec(CLOCK, &clock_mess)) != OK) panic("clock_time err", k);
50939
50940      return( (time_t) clock_mess.NEW_TIME);
50941    }


50944    /*===========================================================================*
50945     *                              fetch_name                                   *
50946     *===========================================================================*/
50947    PUBLIC int fetch_name(path, len, flag)
50948    char *path;                     /* pointer to the path in user space */
50949    int len;                        /* path length, including 0 byte */
50950    int flag;                       /* M3 means path may be in message */
50951    {
50952    /* Go get path and put it in 'user_path'.
50953     * If 'flag' = M3 and 'len' <= M3_STRING, the path is present in 'message'.
50954     * If it is not, go copy it from user space.
```

```
50955     */
50956
50957      register char *rpu, *rpm;
50958      int r;
50959
50960      /* Check name length for validity. */
50961      if (len <= 0) {
50962            err_code = EINVAL;
50963            return(EGENERIC);
50964      }
50965
50966      if (len > PATH_MAX) {
50967            err_code = ENAMETOOLONG;
50968            return(EGENERIC);
50969      }
50970
50971      if (flag == M3 && len <= M3_STRING) {
50972            /* Just copy the path from the message to 'user_path'. */
50973            rpu = &user_path[0];
50974            rpm = pathname;         /* contained in input message */
50975            do { *rpu++ = *rpm++; } while (--len);
50976            r = OK;
50977      } else {
50978            /* String is not contained in the message.  Get it from user space. */
50979            r = sys_copy(who, D, (phys_bytes) path,
50980                    FS_PROC_NR, D, (phys_bytes) user_path, (phys_bytes) len);
50981      }
50982      return(r);
50983    }


50986    /*===========================================================================*
50987     *                              no_sys                                       *
50988     *===========================================================================*/
50989    PUBLIC int no_sys()
50990    {
50991    /* Somebody has used an illegal system call number */
50992
50993      return(EINVAL);
50994    }


50997    /*===========================================================================*
50998     *                              panic                                        *
50999     *===========================================================================*/
51000    PUBLIC void panic(format, num)
51001    char *format;                   /* format string */
51002    int num;                        /* number to go with format string */
51003    {
51004    /* Something awful has happened.  Panics are caused when an internal
51005     * inconsistency is detected, e.g., a programming error or illegal value of a
51006     * defined constant.
51007     */
51008
51009      if (panicking) return;        /* do not panic during a sync */
51010      panicking = TRUE;             /* prevent another panic during the sync */
51011      printf("File system panic: %s ", format);
51012      if (num != NO_NUM) printf("%d",num);
51013      printf("\n");
51014      (void) do_sync();             /* flush everything to the disk */
```

```
51015       sys_abort(RBT_PANIC);
51016     }


51019     /*===========================================================================*
51020      *                              conv2                                        *
51021      *===========================================================================*/
51022     PUBLIC unsigned conv2(norm, w)
51023     int norm;                       /* TRUE if no swap, FALSE for byte swap */
51024     int w;                          /* promotion of 16-bit word to be swapped */
51025     {
51026     /* Possibly swap a 16-bit word between 8086 and 68000 byte order. */
51027
51028       if (norm) return( (unsigned) w & 0xFFFF);
51029       return( ((w&BYTE) << 8) | ( (w>>8) & BYTE));
51030     }


51033     /*===========================================================================*
51034      *                              conv4                                        *
51035      *===========================================================================*/
51036     PUBLIC long conv4(norm, x)
51037     int norm;                       /* TRUE if no swap, FALSE for byte swap */
51038     long x;                         /* 32-bit long to be byte swapped */
51039     {
51040     /* Possibly swap a 32-bit long between 8086 and 68000 byte order. */
51041
51042       unsigned lo, hi;
51043       long l;
51044
51045       if (norm) return(x);                  /* byte order was already ok */
51046       lo = conv2(FALSE, (int) x & 0xFFFF);  /* low-order half, byte swapped */
51047       hi = conv2(FALSE, (int) (x>>16) & 0xFFFF);    /* high-order half, swapped */
51048       l = ( (long) lo <<16) | hi;
51049       return(l);
51050     }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/fs/write.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

51100     /* This file is the counterpart of "read.c".  It contains the code for writing
51101      * insofar as this is not contained in read_write().
51102      *
51103      * The entry points into this file are
51104      *   do_write:     call read_write to perform the WRITE system call
51105      *   clear_zone:   erase a zone in the middle of a file
51106      *   new_block:    acquire a new block
51107      */
51108
51109     #include "fs.h"
51110     #include <string.h>
51111     #include "buf.h"
51112     #include "file.h"
51113     #include "fproc.h"
51114     #include "inode.h"
```

```
51115   #include "super.h"
51116
51117   FORWARD _PROTOTYPE( int write_map, (struct inode *rip, off_t position,
51118                           zone_t new_zone)                              );
51119
51120   FORWARD _PROTOTYPE( void wr_indir, (struct buf *bp, int index, zone_t zone) );
51121
51122   /*===========================================================================*
51123    *                               do_write                                    *
51124    *===========================================================================*/
51125   PUBLIC int do_write()
51126   {
51127   /* Perform the write(fd, buffer, nbytes) system call. */
51128
51129     return(read_write(WRITING));
51130   }


51133   /*===========================================================================*
51134    *                               write_map                                   *
51135    *===========================================================================*/
51136   PRIVATE int write_map(rip, position, new_zone)
51137   register struct inode *rip;     /* pointer to inode to be changed */
51138   off_t position;                 /* file address to be mapped */
51139   zone_t new_zone;                /* zone # to be inserted */
51140   {
51141   /* Write a new zone into an inode. */
51142     int scale, ind_ex, new_ind, new_dbl, zones, nr_indirects, single, zindex, ex;
51143     zone_t z, z1;
51144     register block_t b;
51145     long excess, zone;
51146     struct buf *bp;
51147
51148     rip->i_dirt = DIRTY;          /* inode will be changed */
51149     bp = NIL_BUF;
51150     scale = rip->i_sp->s_log_zone_size;           /* for zone-block conversion */
51151     zone = (position/BLOCK_SIZE) >> scale;        /* relative zone # to insert */
51152     zones = rip->i_ndzones;       /* # direct zones in the inode */
51153     nr_indirects = rip->i_nindirs;/* # indirect zones per indirect block */
51154
51155     /* Is 'position' to be found in the inode itself? */
51156     if (zone < zones) {
51157           zindex = (int) zone;    /* we need an integer here */
51158           rip->i_zone[zindex] = new_zone;
51159           return(OK);
51160     }
51161
51162     /* It is not in the inode, so it must be single or double indirect. */
51163     excess = zone - zones;        /* first Vx_NR_DZONES don't count */
51164     new_ind = FALSE;
51165     new_dbl = FALSE;
51166
51167     if (excess < nr_indirects) {
51168           /* 'position' can be located via the single indirect block. */
51169           z1 = rip->i_zone[zones];        /* single indirect zone */
51170           single = TRUE;
51171     } else {
51172           /* 'position' can be located via the double indirect block. */
51173           if ( (z = rip->i_zone[zones+1]) == NO_ZONE) {
51174                   /* Create the double indirect block. */
```

```
51175                   if ( (z = alloc_zone(rip->i_dev, rip->i_zone[0])) == NO_ZONE)
51176                           return(err_code);
51177                   rip->i_zone[zones+1] = z;
51178                   new_dbl = TRUE; /* set flag for later */
51179           }
51180
51181           /* Either way, 'z' is zone number for double indirect block. */
51182           excess -= nr_indirects; /* single indirect doesn't count */
51183           ind_ex = (int) (excess / nr_indirects);
51184           excess = excess % nr_indirects;
51185           if (ind_ex >= nr_indirects) return(EFBIG);
51186           b = (block_t) z << scale;
51187           bp = get_block(rip->i_dev, b, (new_dbl ? NO_READ : NORMAL));
51188           if (new_dbl) zero_block(bp);
51189           z1 = rd_indir(bp, ind_ex);
51190           single = FALSE;
51191      }
51192
51193      /* z1 is now single indirect zone; 'excess' is index. */
51194      if (z1 == NO_ZONE) {
51195           /* Create indirect block and store zone # in inode or dbl indir blk. */
51196           z1 = alloc_zone(rip->i_dev, rip->i_zone[0]);
51197           if (single)
51198                   rip->i_zone[zones] = z1;        /* update inode */
51199           else
51200                   wr_indir(bp, ind_ex, z1);       /* update dbl indir */
51201
51202           new_ind = TRUE;
51203           if (bp != NIL_BUF) bp->b_dirt = DIRTY;  /* if double ind, it is dirty*/
51204           if (z1 == NO_ZONE) {
51205                   put_block(bp, INDIRECT_BLOCK);  /* release dbl indirect blk */
51206                   return(err_code);       /* couldn't create single ind */
51207           }
51208      }
51209      put_block(bp, INDIRECT_BLOCK);        /* release double indirect blk */
51210
51211      /* z1 is indirect block's zone number. */
51212      b = (block_t) z1 << scale;
51213      bp = get_block(rip->i_dev, b, (new_ind ? NO_READ : NORMAL) );
51214      if (new_ind) zero_block(bp);
51215      ex = (int) excess;                    /* we need an int here */
51216      wr_indir(bp, ex, new_zone);
51217      bp->b_dirt = DIRTY;
51218      put_block(bp, INDIRECT_BLOCK);
51219
51220      return(OK);
51221    }


51224    /*===========================================================================*
51225     *                               wr_indir                                    *
51226     *===========================================================================*/
51227    PRIVATE void wr_indir(bp, index, zone)
51228    struct buf *bp;                 /* pointer to indirect block */
51229    int index;                      /* index into *bp */
51230    zone_t zone;                    /* zone to write */
51231    {
51232    /* Given a pointer to an indirect block, write one entry. */
51233
51234      struct super_block *sp;
```

```
51235
51236     sp = get_super(bp->b_dev);    /* need super block to find file sys type */
51237
51238     /* write a zone into an indirect block */
51239     if (sp->s_version == V1)
51240           bp->b_v1_ind[index] = (zone1_t) conv2(sp->s_native, (int)  zone);
51241     else
51242           bp->b_v2_ind[index] = (zone_t)  conv4(sp->s_native, (long) zone);
51243   }


51246   /*===========================================================================*
51247    *                              clear_zone                                   *
51248    *===========================================================================*/
51249   PUBLIC void clear_zone(rip, pos, flag)
51250   register struct inode *rip;     /* inode to clear */
51251   off_t pos;                      /* points to block to clear */
51252   int flag;                       /* 0 if called by read_write, 1 by new_block */
51253   {
51254   /* Zero a zone, possibly starting in the middle.  The parameter 'pos' gives
51255    * a byte in the first block to be zeroed.  Clearzone() is called from
51256    * read_write and new_block().
51257    */
51258
51259     register struct buf *bp;
51260     register block_t b, blo, bhi;
51261     register off_t next;
51262     register int scale;
51263     register zone_t zone_size;
51264
51265     /* If the block size and zone size are the same, clear_zone() not needed. */
51266     scale = rip->i_sp->s_log_zone_size;
51267     if (scale == 0) return;
51268
51269     zone_size = (zone_t) BLOCK_SIZE << scale;
51270     if (flag == 1) pos = (pos/zone_size) * zone_size;
51271     next = pos + BLOCK_SIZE - 1;
51272
51273     /* If 'pos' is in the last block of a zone, do not clear the zone. */
51274     if (next/zone_size != pos/zone_size) return;
51275     if ( (blo = read_map(rip, next)) == NO_BLOCK) return;
51276     bhi = (  ((blo>>scale)+1) << scale)   - 1;
51277
51278     /* Clear all the blocks between 'blo' and 'bhi'. */
51279     for (b = blo; b <= bhi; b++) {
51280           bp = get_block(rip->i_dev, b, NO_READ);
51281           zero_block(bp);
51282           put_block(bp, FULL_DATA_BLOCK);
51283     }
51284   }


51287   /*===========================================================================*
51288    *                              new_block                                    *
51289    *===========================================================================*/
51290   PUBLIC struct buf *new_block(rip, position)
51291   register struct inode *rip;     /* pointer to inode */
51292   off_t position;                 /* file pointer */
51293   {
51294   /* Acquire a new block and return a pointer to it.  Doing so may require
```

```
51295     * allocating a complete zone, and then returning the initial block.
51296     * On the other hand, the current zone may still have some unused blocks.
51297     */
51298
51299      register struct buf *bp;
51300      block_t b, base_block;
51301      zone_t z;
51302      zone_t zone_size;
51303      int scale, r;
51304      struct super_block *sp;
51305
51306      /* Is another block available in the current zone? */
51307      if ( (b = read_map(rip, position)) == NO_BLOCK) {
51308            /* Choose first zone if possible. */
51309            /* Lose if the file is nonempty but the first zone number is NO_ZONE
51310             * corresponding to a zone full of zeros.  It would be better to
51311             * search near the last real zone.
51312             */
51313            if (rip->i_zone[0] == NO_ZONE) {
51314                    sp = rip->i_sp;
51315                    z = sp->s_firstdatazone;
51316            } else {
51317                    z = rip->i_zone[0];     /* hunt near first zone */
51318            }
51319            if ( (z = alloc_zone(rip->i_dev, z)) == NO_ZONE) return(NIL_BUF);
51320            if ( (r = write_map(rip, position, z)) != OK) {
51321                    free_zone(rip->i_dev, z);
51322                    err_code = r;
51323                    return(NIL_BUF);
51324            }
51325
51326            /* If we are not writing at EOF, clear the zone, just to be safe. */
51327            if ( position != rip->i_size) clear_zone(rip, position, 1);
51328            scale = rip->i_sp->s_log_zone_size;
51329            base_block = (block_t) z << scale;
51330            zone_size = (zone_t) BLOCK_SIZE << scale;
51331            b = base_block + (block_t)((position % zone_size)/BLOCK_SIZE);
51332      }
51333
51334      bp = get_block(rip->i_dev, b, NO_READ);
51335      zero_block(bp);
51336      return(bp);
51337    }


51340    /*===========================================================================*
51341     *                              zero_block                                   *
51342     *===========================================================================*/
51343    PUBLIC void zero_block(bp)
51344    register struct buf *bp;        /* pointer to buffer to zero */
51345    {
51346    /* Zero a block. */
51347
51348      memset(bp->b_data, 0, BLOCK_SIZE);
51349      bp->b_dirt = DIRTY;
51350    }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/hton.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

51400    /*
51401    The following macro definitions convert to and from the network standard byte
51402    order. The macros with their name in lower case guarantee to evaluate their
51403    argument exectly once. The function of the macros is encoded in their names;
51404    htons means convert a (unsigned) short in host byte order to network byte order.
51405    */
51406
51407    #ifndef _NET__HTON_H
51408    #define _NET__HTON_H
51409
51410    #include <minix/config.h>
51411
51412    extern u16_t _tmp;
51413    extern u32_t _tmp_l;
51414
51415    /* Find out about the byte order. */
51416
51417    /* assume <minix/config.h> is included, let's check */
51418    #if (CHIP == 0)
51419    #include "CHIP macro not set, include <minix/config.h>"
51420    #endif
51421
51422    #if (CHIP == INTEL)
51423    #define LITTLE_ENDIAN   1
51424    #endif
51425
51426    #if (CHIP == M68000 || CHIP == SPARC)
51427    #define BIG_ENDIAN      1
51428    #endif
51429
51430
51431    #if (LITTLE_ENDIAN) && (BIG_ENDIAN)
51432    #include "both LITTLE_ENDIAN and BIG_ENDIAN are defined"
51433                            /* LITTLE_ENDIAN and BIG_ENDIAN are both defined */
51434    #endif
51435
51436    #if !(LITTLE_ENDIAN) && !(BIG_ENDIAN)
51437    #include "neither LITTLE_ENDIAN nor BIG_ENDIAN is defined"
51438                            /* LITTLE_ENDIAN and BIG_ENDIAN are both NOT defined */
51439    #endif
51440
51441    #if LITTLE_ENDIAN
51442    #define htons(x) (_tmp=(x), ((_tmp>>8) & 0xff) | ((_tmp<<8) & 0xff00))
51443    #define HTONS(x) ( ( (((unsigned short)(x)) >>8) & 0xff) | \
51444                    ((((unsigned short)(x)) & 0xff)<<8) )
51445
51446    #define ntohs(x) (_tmp=(x), ((_tmp>>8) & 0xff) | ((_tmp<<8) & 0xff00))
51447    #define NTOHS(x) ( ( (((unsigned short)(x)) >>8) & 0xff) | \
51448                    ((((unsigned short)(x)) & 0xff)<<8) )
51449
51450    #define htonl(x) (_tmp_l=(x), ((_tmp_l>>24) & 0xffL) | \
51451            ((_tmp_l>>8) & 0xff00L) | \
51452            ((_tmp_l<<8) & 0xff0000L) | ((_tmp_l<<24) & 0xff000000L))
51453    #define HTONL(x) ((((x)>>24) & 0xffL) | (((x)>>8) & 0xff00L) | \
51454            (((x)<<8) & 0xff0000L) | (((x)<<24) & 0xff000000L))
```

```
51455    #define ntohl(x) (_tmp_l=(x), ((_tmp_l>>24) & 0xffL) \
51456            | ((_tmp_l>>8) & 0xff00L) | \
51457            ((_tmp_l<<8) & 0xff0000L) | ((_tmp_l<<24) & 0xff000000L))
51458    #define NTOHL(x) ((((x)>>24) & 0xffL) | (((x)>>8) & 0xff00L) | \
51459            (((x)<<8) & 0xff0000L) | (((x)<<24) & 0xff000000L))
51460
51461    #endif /* LITTLE_ENDING */
51462
51463    #if BIG_ENDIAN
51464    #define htons(x) (x)
51465    #define HTONS(x) (x)
51466    #define ntohs(x) (x)
51467    #define NTOHS(x) (x)
51468    #define htonl(x) (x)
51469    #define HTONL(x) (x)
51470    #define ntohl(x) (x)
51471    #define NTOHL(x) (x)
51472    #endif /* BIG_ENDIAN */
51473
51474    #endif /* _NET__HTON_H */
```

++

include/net/netlib.h

++

```
51500    /*
51501    net/netlib.h
51502    */
51503
51504    #ifndef _NET__NETLIB_H_
51505    #define _NET__NETLIB_H_
51506
51507    #ifndef _ANSI
51508    #include <ansi.h>
51509    #endif
51510
51511    _PROTOTYPE (int ruserok, (const char *rhost, int superuser, const char *ruser,
51512            const char *luser) );
51513    _PROTOTYPE (int rcmd, (char **ahost, int rport, const char *locuser,
51514                    const char *remuser, const char *cmd, int *fd2p) );
51515
51516    #define ETH_DEVICE      "/dev/eth"
51517    #define IP_DEVICE       "/dev/ip"
51518    #define TCP_DEVICE      "/dev/tcp"
51519    #define UDP_DEVICE      "/dev/udp"
51520
51521    #endif /* _NET__NETLIB_H_ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/eth_hdr.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

51600    /*
51601    server/ip/gen/eth_hdr.h
51602    */
51603
51604    #ifndef __SERVER__IP__GEN__ETH_HDR_H__
51605    #define __SERVER__IP__GEN__ETH_HDR_H__
51606
51607    typedef struct eth_hdr
51608    {
51609            ether_addr_t eh_dst;
51610            ether_addr_t eh_src;
51611            ether_type_t eh_proto;
51612    } eth_hdr_t;
51613
51614    #endif /* __SERVER__IP__GEN__ETH_HDR_H__ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/eth_io.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

51700    /*
51701    server/gen/ip/eth_io.h
51702    */
51703
51704    #ifndef __SERVER__IP__GEN__ETH_IO_H__
51705    #define __SERVER__IP__GEN__ETH_IO_H__
51706
51707    typedef struct nwio_ethopt
51708    {
51709            u32_t nweo_flags;
51710            ether_addr_t nweo_multi, nweo_rem;
51711            ether_type_t nweo_type;
51712    } nwio_ethopt_t;
51713
51714    #define NWEO_NOFLAGS    0x0000L
51715    #define NWEO_ACC_MASK   0x0003L
51716    #       define NWEO_EXCL        0x00000001L
51717    #       define NWEO_SHARED      0x00000002L
51718    #       define NWEO_COPY        0x00000003L
51719    #define NWEO_LOC_MASK   0x0010L
51720    #       define NWEO_EN_LOC      0x00000010L
51721    #       define NWEO_DI_LOC      0x00100000L
51722    #define NWEO_BROAD_MASK 0x0020L
51723    #       define NWEO_EN_BROAD    0x00000020L
51724    #       define NWEO_DI_BROAD    0x00200000L
51725    #define NWEO_MULTI_MASK 0x0040L
51726    #       define NWEO_EN_MULTI    0x00000040L
51727    #       define NWEO_DI_MULTI    0x00400000L
51728    #define NWEO_PROMISC_MASK 0x0080L
51729    #       define NWEO_EN_PROMISC  0x00000080L
```

```
51730   #       define NWEO_DI_PROMISC  0x00800000L
51731   #define NWEO_REM_MASK   0x0100L
51732   #       define NWEO_REMSPEC     0x00000100L
51733   #       define NWEO_REMANY      0x01000000L
51734   #define NWEO_TYPE_MASK  0x0200L
51735   #       define NWEO_TYPESPEC    0x00000200L
51736   #       define NWEO_TYPEANY     0x02000000L
51737   #define NWEO_RW_MASK    0x1000L
51738   #       define NWEO_RWDATONLY   0x00001000L
51739   #       define NWEO_RWDATALL    0x10000000L
51740
51741   typedef struct eth_stat
51742   {
51743           unsigned long ets_recvErr,      /* # receive errors */
51744                   ets_sendErr,            /* # send error */
51745                   ets_OVW,                /* # buffer overwrite warnings,
51746                                              (packets arrive faster than
51747                                              can be processed) */
51748                   ets_CRCerr,             /* # crc errors of read */
51749                   ets_frameAll,           /* # frames not alligned (# bits
51750                                              not a mutiple of 8) */
51751                   ets_missedP,            /* # packets missed due to too
51752                                              slow packet processing */
51753                   ets_packetR,            /* # packets received */
51754                   ets_packetT,            /* # packets transmitted */
51755                   ets_transDef,           /* # transmission defered (there
51756                                              was a transmission of an
51757                                              other station in progress */
51758                   ets_collision,          /* # collissions */
51759                   ets_transAb,            /* # transmissions aborted due
51760                                              to accesive collisions */
51761                   ets_carrSense,          /* # carrier sense lost */
51762                   ets_fifoUnder,          /* # fifo underruns (processor
51763                                              is too busy) */
51764                   ets_fifoOver,           /* # fifo overruns (processor is
51765                                              too busy) */
51766                   ets_CDheartbeat,        /* # times unable to transmit
51767                                              collision signal */
51768                   ets_OWC;                /* # times out of window
51769                                              collision */
51770   } eth_stat_t;
51771
51772   typedef struct nwio_ethstat
51773   {
51774           ether_addr_t nwes_addr;
51775           eth_stat_t nwes_stat;
51776   } nwio_ethstat_t;
51777
51778   #endif /* __SERVER__IP__GEN__ETH_IO_H__ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/ether.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

51800    /*
51801    server/ip/gen/ether.h
51802    */
51803
51804    #ifndef __SERVER__IP__GEN__ETHER_H__
51805    #define __SERVER__IP__GEN__ETHER_H__
51806
51807    #define ETH_MIN_PACK_SIZE       60
51808    #define ETH_MAX_PACK_SIZE       1514
51809    #define ETH_HDR_SIZE            14
51810
51811    typedef struct ether_addr
51812    {
51813            u8_t ea_addr[6];
51814    } ether_addr_t;
51815
51816    typedef u16_t ether_type_t;
51817    typedef U16_t Ether_type_t;
51818
51819    #define ETH_RARP_PROTO  0x8035
51820    #define ETH_ARP_PROTO   0x806
51821    #define ETH_IP_PROTO    0x800
51822
51823    #endif /* __SERVER__IP__GEN__ETHER_H__ */


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/icmp.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

51900    /*
51901    server/ip/gen/icmp.h
51902    */
51903
51904    #ifndef __SERVER__IP__GEN__ICMP_H__
51905    #define __SERVER__IP__GEN__ICMP_H__
51906
51907    #define ICMP_MIN_HDR_LEN        4
51908
51909    #define ICMP_TYPE_ECHO_REPL     0
51910    #define ICMP_TYPE_DST_UNRCH     3
51911    #       define ICMP_NET_UNRCH           0
51912    #       define ICMP_HOST_UNRCH          1
51913    #define ICMP_TYPE_REDIRECT      5
51914    #       define ICMP_REDIRECT_NET        0
51915    #       define ICMP_REDIRECT_HOST       1
51916    #define ICMP_TYPE_ECHO_REQ      8
51917    #define ICMP_TYPE_ROUTER_ADVER  9
51918    #define ICMP_TYPE_ROUTE_SOL     10
51919    #define ICMP_TYPE_TIME_EXCEEDED 11
51920    #       define ICMP_TTL_EXC             0
51921
51922    #endif /* __SERVER__IP__GEN__ICMP_H__ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/icmp_hdr.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

52000    /*
52001    server/ip/gen/icmp_hdr.h
52002    */
52003
52004    #ifndef __SERVER__IP__GEN__ICMP_HDR_H__
52005    #define __SERVER__IP__GEN__ICMP_HDR_H__
52006
52007    typedef struct icmp_id_seq
52008    {
52009            u16_t   iis_id, iis_seq;
52010    } icmp_id_seq_t;
52011
52012    typedef struct icmp_ip_id
52013    {
52014            ip_hdr_t iii_hdr;
52015            /* ip_hdr_options and 64 bytes of data */
52016    } icmp_ip_id_t;
52017
52018    typedef struct icmp_ram         /* RFC 1256 */
52019    {
52020            u8_t    iram_na;
52021            u8_t    iram_aes;
52022            u16_t   iram_lt;
52023    } icmp_ram_t;
52024
52025    typedef struct icmp_hdr
52026    {
52027            u8_t ih_type, ih_code;
52028            u16_t ih_chksum;
52029            union
52030            {
52031                    u32_t ihh_unused;
52032                    icmp_id_seq_t ihh_idseq;
52033                    ipaddr_t ihh_gateway;
52034                    icmp_ram_t ihh_ram;
52035            } ih_hun;
52036            union
52037            {
52038                    icmp_ip_id_t ihd_ipid;
52039                    u8_t uhd_data[1];
52040            } ih_dun;
52041    } icmp_hdr_t;
52042
52043    #endif /* __SERVER__IP__GEN__ICMP_HDR_H__ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/if_ether.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

52100    /*
52101    server/ip/gen/if_ether.h
52102    */
52103
52104    #ifndef __SERVER__IP__GEN__IF_ETHER_H__
52105    #define __SERVER__IP__GEN__IF_ETHER_H__
52106
52107    struct ether_addr;
52108
52109    #define _PATH_ETHERS    "/etc/ethers"
52110
52111    char *ether_ntoa _ARGS(( struct ether_addr *e ));
52112    struct ether_addr *ether_aton _ARGS(( char *s ));
52113    int ether_ntohost _ARGS(( char *hostname, struct ether_addr *e ));
52114    int ether_hostton _ARGS(( char *hostname, struct ether_addr *e ));
52115    int ether_line _ARGS(( char *l, struct ether_addr *e, char *hostname ));
52116
52117    #endif /* __SERVER__IP__GEN__IF_ETHER_H__ */



++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/in.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

52200    /*
52201    server/ip/gen/in.h
52202    */
52203
52204    #ifndef __SERVER__IP__GEN__IN_H__
52205    #define __SERVER__IP__GEN__IN_H__
52206
52207    #define IP_MIN_HDR_SIZE         20
52208    #define IP_MAX_HDR_SIZE         60              /* 15 * 4 */
52209    #define IP_VERSION              4
52210    #define IP_MAX_TTL              255
52211    #define IP_MAX_PACKSIZE         40000           /* 8192 */
52212            /* Note: this restriction is not part of the IP-protocol but
52213               introduced by this implementation. */
52214
52215    #define IPPROTO_ICMP            1
52216    #define IPPROTO_TCP             6
52217    #define IPPROTO_UDP             17
52218
52219    typedef u32_t ipaddr_t;
52220    typedef u8_t ipproto_t;
52221    typedef struct ip_hdropt
52222    {
52223            u8_t iho_opt_siz;
52224            u8_t iho_data[IP_MAX_HDR_SIZE-IP_MIN_HDR_SIZE];
52225    } ip_hdropt_t;
52226
52227    #endif /* __SERVER__IP__GEN__IN_H__ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/inet.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

52300    /*
52301    server/ip/gen/inet.h
52302    */
52303
52304    #ifndef __SERVER__IP__GEN__INET_H__
52305    #define __SERVER__IP__GEN__INET_H__
52306
52307    ipaddr_t inet_addr _ARGS(( const char *addr ));
52308    ipaddr_t inet_network _ARGS(( const char *addr ));
52309    char *inet_ntoa _ARGS(( ipaddr_t addr ));
52310    int inet_aton _ARGS(( const char *cp, ipaddr_t *pin ));
52311
52312    #endif /* __SERVER__IP__GEN__INET_H__ */



++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/ip_hdr.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

52400    /*
52401    server/ip/gen/ip_hdr.h
52402    */
52403
52404    #ifndef __SERVER__IP__GEN__HDR_H__
52405    #define __SERVER__IP__GEN__HDR_H__
52406
52407    typedef struct ip_hdr
52408    {
52409            u8_t ih_vers_ihl,
52410                    ih_tos;
52411            u16_t ih_length,
52412                    ih_id,
52413                    ih_flags_fragoff;
52414            u8_t ih_ttl,
52415                    ih_proto;
52416            u16_t ih_hdr_chk;
52417            ipaddr_t ih_src,
52418                    ih_dst;
52419    } ip_hdr_t;
52420
52421    #define IH_IHL_MASK     0xf
52422    #define IH_VERSION_MASK 0xf
52423    #define IH_FRAGOFF_MASK 0x1fff
52424    #define IH_MORE_FRAGS   0x2000
52425    #define IH_DONT_FRAG    0x4000
52426    #define IH_FLAGS_UNUSED 0x8000
52427
52428    #define IP_OPT_COPIED   0x80
52429    #define IP_OPT_NUMBER   0x1f
52430
52431    #endif /* __SERVER__IP__GEN__HDR_H__ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/ip_io.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

52500    /*
52501    server/ip/gen/ip_io.h
52502    */
52503
52504    #ifndef __SERVER__IP__GEN__IP_IO_H__
52505    #define __SERVER__IP__GEN__IP_IO_H__
52506
52507    typedef struct nwio_ipconf
52508    {
52509            u32_t   nwic_flags;
52510            ipaddr_t nwic_ipaddr;
52511            ipaddr_t nwic_netmask;
52512    } nwio_ipconf_t;
52513
52514    #define NWIC_NOFLAGS            0x0
52515    #define NWIC_FLAGS              0x3
52516    #       define NWIC_IPADDR_SET          0x1
52517    #       define NWIC_NETMASK_SET         0x2
52518
52519    typedef struct nwio_ipopt
52520    {
52521            u32_t nwio_flags;
52522            ipaddr_t nwio_rem;
52523            ip_hdropt_t nwio_hdropt;
52524            u8_t nwio_tos;
52525            u8_t nwio_ttl;
52526            u8_t nwio_df;
52527            ipproto_t nwio_proto;
52528    } nwio_ipopt_t;
52529
52530    #define NWIO_NOFLAGS    0x00001
52531    #define NWIO_ACC_MASK   0x00031
52532    #       define NWIO_EXCL        0x000000011
52533    #       define NWIO_SHARED      0x000000021
52534    #       define NWIO_COPY        0x000000031
52535    #define NWIO_LOC_MASK   0x00101
52536    #       define NWIO_EN_LOC      0x000000101
52537    #       define NWIO_DI_LOC      0x001000001
52538    #define NWIO_BROAD_MASK 0x00201
52539    #       define NWIO_EN_BROAD    0x000000201
52540    #       define NWIO_DI_BROAD    0x002000001
52541    #define NWIO_REM_MASK   0x01001
52542    #       define NWIO_REMSPEC     0x000001001
52543    #       define NWIO_REMANY      0x010000001
52544    #define NWIO_PROTO_MASK 0x02001
52545    #       define NWIO_PROTOSPEC   0x000002001
52546    #       define NWIO_PROTOANY    0x020000001
52547    #define NWIO_HDR_O_MASK 0x04001
52548    #       define NWIO_HDR_O_SPEC  0x000004001
52549    #       define NWIO_HDR_O_ANY   0x040000001
52550    #define NWIO_RW_MASK    0x10001
52551    #       define NWIO_RWDATONLY   0x000010001
52552    #       define NWIO_RWDATALL    0x100000001
52553
52554    #endif /* __SERVER__IP__GEN__IP_IO_H__ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/nameser.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

52600    /*
52601     * Copyright (c) 1983, 1989 Regents of the University of California.
52602     * All rights reserved.
52603     *
52604     * Redistribution and use in source and binary forms are permitted
52605     * provided that: (1) source distributions retain this entire copyright
52606     * notice and comment, and (2) distributions including binaries display
52607     * the following acknowledgement:  'This product includes software
52608     * developed by the University of California, Berkeley and its contributors''
52609     * in the documentation or other materials provided with the distribution
52610     * and in all advertising materials mentioning features or use of this
52611     * software. Neither the name of the University nor the names of its
52612     * contributors may be used to endorse or promote products derived
52613     * from this software without specific prior written permission.
52614     * THIS SOFTWARE IS PROVIDED 'AS IS'' AND WITHOUT ANY EXPRESS OR
52615     * IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
52616     * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
52617     *
52618     *      @(#)nameser.h   5.24 (Berkeley) 6/1/90
52619     */
52620
52621    /*
52622    server/ip/gen/nameser.h
52623
52624    Created Sept 18, 1991 by Philip Homburg
52625    */
52626
52627    #ifndef __SERVER__IP__GEN__NAEMSER_H__
52628    #define __SERVER__IP__GEN__NAEMSER_H__
52629
52630    typedef struct dns_hdr
52631    {
52632            u16_t dh_id;
52633            u8_t dh_flag1;
52634            u8_t dh_flag2;
52635            u16_t dh_qdcount;
52636            u16_t dh_ancount;
52637            u16_t dh_nscount;
52638            u16_t dh_arcount;
52639    } dns_hdr_t;
52640
52641    #define DHF_QR          0x80
52642    #define DHF_OPCODE      0x78
52643    #define DHF_AA          0x04
52644    #define DHF_TC          0x02
52645    #define DHF_RD          0x01
52646
52647    #define DHF_RA          0x80
52648    #define DHF_PR          0x40
52649    #define DHF_UNUSED      0x30
52650    #define DHF_RCODE       0x0F
52651
52652
52653    /*
52654    Define constants based on rfc883
```

```
52655   */
52656   #define PACKETSZ        512             /* maximum packet size */
52657   #define MAXDNAME        256             /* maximum domain name */
52658   #define MAXCDNAME       255             /* maximum compressed domain name */
52659   #define MAXLABEL        63              /* maximum length of domain label */
52660           /* Number of bytes of fixed size data in query structure */
52661   #define QFIXEDSZ        4
52662           /* number of bytes of fixed size data in resource record */
52663   #define RRFIXEDSZ       10
52664   #define INDIR_MASK      0xc0
52665                           /* Defines for handling compressed domain names */
52666
52667   /*
52668   Opcodes for DNS
52669   */
52670
52671   #define QUERY           0x0                     /* standard query */
52672   #define IQUERY          0x1                     /* inverse query */
52673
52674   /*
52675   Error codes
52676   */
52677   #define NOERROR         0                       /* no error */
52678   #define FORMERR         1                       /* format error */
52679   #define SERVFAIL        2                       /* server failure */
52680   #define NXDOMAIN        3                       /* non existent domain */
52681   #define NOTIMP          4                       /* not implemented */
52682   #define REFUSED         5                       /* query refused */
52683           /* non standard */
52684   #define NOCHANGE        0xf                     /* update failed to change db */
52685
52686   /* Valid types */
52687
52688   #define T_A             1               /* host address */
52689   #define T_NS            2               /* authoritative server */
52690   #define T_MD            3               /* mail destination */
52691   #define T_MF            4               /* mail forwarder */
52692   #define T_CNAME         5               /* connonical name */
52693   #define T_SOA           6               /* start of authority zone */
52694   #define T_MB            7               /* mailbox domain name */
52695   #define T_MG            8               /* mail group member */
52696   #define T_MR            9               /* mail rename name */
52697   #define T_NULL          10              /* null resource record */
52698   #define T_WKS           11              /* well known service */
52699   #define T_PTR           12              /* domain name pointer */
52700   #define T_HINFO         13              /* host information */
52701   #define T_MINFO         14              /* mailbox information */
52702   #define T_MX            15              /* mail routing information */
52703   #define T_TXT           16              /* text strings */
52704           /* non standard */
52705   #define T_UINFO         100             /* user (finger) information */
52706   #define T_UID           101             /* user ID */
52707   #define T_GID           102             /* group ID */
52708   #define T_UNSPEC        103             /* Unspecified format (binary data) */
52709           /* Query type values which do not appear in resource records */
52710   #define T_AXFR          252             /* transfer zone of authority */
52711   #define T_MAILB         253             /* transfer mailbox records */
52712   #define T_MAILA         254             /* transfer mail agent records */
52713   #define T_ANY           255             /* wildcard match */
52714
```

```
52715    /* Valid classes */
52716
52717    #define C_IN            1                       /* the internet */
52718    #define C_HS            4               /* for Hesiod name server at MIT */
52719
52720    #define C_ANY           255                     /* wildcard */
52721
52722    #endif /* __SERVER__IP__GEN__NAEMSER_H__ */
```

++
include/net/gen/netdb.h
++

```
52800    /*-
52801     * Copyright (c) 1980, 1983, 1988 Regents of the University of California.
52802     * All rights reserved.
52803     *
52804     * Redistribution and use in source and binary forms, with or without
52805     * modification, are permitted provided that the following conditions
52806     * are met:
52807     * 1. Redistributions of source code must retain the above copyright
52808     *    notice, this list of conditions and the following disclaimer.
52809     * 2. Redistributions in binary form must reproduce the above copyright
52810     *    notice, this list of conditions and the following disclaimer in the
52811     *    documentation and/or other materials provided with the distribution.
52812     * 3. All advertising materials mentioning features or use of this software
52813     *    must display the following acknowledgement:
52814     *      This product includes software developed by the University of
52815     *      California, Berkeley and its contributors.
52816     * 4. Neither the name of the University nor the names of its contributors
52817     *    may be used to endorse or promote products derived from this software
52818     *    without specific prior written permission.
52819     *
52820     * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS `AS IS'' AND
52821     * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
52822     * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
52823     * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
52824     * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
52825     * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
52826     * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
52827     * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
52828     * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
52829     * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
52830     * SUCH DAMAGE.
52831     *
52832     *      @(#)netdb.h     5.15 (Berkeley) 4/3/91
52833     */
52834
52835    #ifndef _NETDB_H_
52836    #define _NETDB_H_
52837
52838    #define _PATH_HEQUIV    "/etc/hosts.equiv"
52839    #define _PATH_HOSTS     "/etc/hosts"
52840    #define _PATH_NETWORKS  "/etc/networks"
52841    #define _PATH_PROTOCOLS "/etc/protocols"
52842    #define _PATH_SERVICES  "/etc/services"
52843
52844    /*
```

```
52845     * Structures returned by network data base library.  All addresses are
52846     * supplied in host order, and returned in network order (suitable for
52847     * use in system calls).
52848     */
52849    struct  hostent {
52850            char    *h_name;        /* official name of host */
52851            char    **h_aliases;    /* alias list */
52852            int     h_addrtype;     /* host address type */
52853            int     h_length;       /* length of address */
52854            char    **h_addr_list;  /* list of addresses from name server */
52855    #define h_addr  h_addr_list[0]  /* address, for backward compatiblity */
52856    };
52857
52858    /*
52859     * Assumption here is that a network number
52860     * fits in 32 bits -- probably a poor one.
52861     */
52862    struct  netent {
52863            char            *n_name;        /* official name of net */
52864            char            **n_aliases;    /* alias list */
52865            int             n_addrtype;     /* net address type */
52866            unsigned long   n_net;          /* network # */
52867    };
52868
52869    struct  servent {
52870            char    *s_name;        /* official service name */
52871            char    **s_aliases;    /* alias list */
52872            int     s_port;         /* port # */
52873            char    *s_proto;       /* protocol to use */
52874    };
52875
52876    struct  protoent {
52877            char    *p_name;        /* official protocol name */
52878            char    **p_aliases;    /* alias list */
52879            int     p_proto;        /* protocol # */
52880    };
52881
52882    /*
52883     * Error return codes from gethostbyname() and gethostbyaddr()
52884     * (left in extern int h_errno).
52885     */
52886    extern int h_errno;
52887
52888    #define HOST_NOT_FOUND  1 /* Authoritative Answer Host not found */
52889    #define TRY_AGAIN       2 /* Non-Authoritive Host not found, or SERVERFAIL */
52890    #define NO_RECOVERY     3 /* Non recoverable errors, FORMERR, REFUSED, NOTIMP */
52891    #define NO_DATA         4 /* Valid name, no data record of requested type */
52892    #define NO_ADDRESS      NO_DATA         /* no address, look for MX record */
52893
52894    #ifndef _ANSI_H
52895    #include <ansi.h>
52896    #endif
52897
52898    void            endhostent _ARGS((void));
52899    void            endnetent _ARGS((void));
52900    void            endprotoent _ARGS((void));
52901    void            endservent _ARGS((void));
52902    struct hostent  *gethostbyaddr _ARGS((const char *, int, int));
52903    struct hostent  *gethostbyname _ARGS((char *));
52904    struct hostent  *gethostent _ARGS((void));
```

```
52905    struct netent   *getnetbyaddr _ARGS((long, int)); /* u_long? */
52906    struct netent   *getnetbyname _ARGS((const char *));
52907    struct netent   *getnetent _ARGS((void));
52908    struct protoent *getprotobyname _ARGS((const char *));
52909    struct protoent *getprotobynumber _ARGS((int));
52910    struct protoent *getprotoent _ARGS((void));
52911    struct servent  *getservbyname _ARGS((const char *, const char *));
52912    struct servent  *getservbyport _ARGS((int, const char *));
52913    struct servent  *getservent _ARGS((void));
52914    void            herror _ARGS((const char *));
52915    void            sethostent _ARGS((int));
52916    /* void         sethostfile _ARGS((const char *)); */
52917    void            setnetent _ARGS((int));
52918    void            setprotoent _ARGS((int));
52919    void            setservent _ARGS((int));
52920
52921    #endif /* !_NETDB_H_ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/oneCsum.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

53000    /*
53001    server/ip/gen/oneCsum.h
53002    */
53003
53004    #ifndef __SERVER__IP__GEN__ONECSUM_H__
53005    #define __SERVER__IP__GEN__ONECSUM_H__
53006
53007    u16_t oneC_sum _ARGS(( U16_t prev, void *data, size_t data_len ));
53008
53009    #endif /* __SERVER__IP__GEN__ONECSUM_H__ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/resolv.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

53100    /*
53101     * Copyright (c) 1983, 1987, 1989 The Regents of the University of California.
53102     * All rights reserved.
53103     *
53104     * Redistribution and use in source and binary forms are permitted
53105     * provided that: (1) source distributions retain this entire copyright
53106     * notice and comment, and (2) distributions including binaries display
53107     * the following acknowledgement:  `This product includes software
53108     * developed by the University of California, Berkeley and its contributors''
53109     * in the documentation or other materials provided with the distribution
53110     * and in all advertising materials mentioning features or use of this
53111     * software. Neither the name of the University nor the names of its
53112     * contributors may be used to endorse or promote products derived
53113     * from this software without specific prior written permission.
53114     * THIS SOFTWARE IS PROVIDED `AS IS'' AND WITHOUT ANY EXPRESS OR
```

```
53115     * IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
53116     * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
53117     *
53118     *      @(#)resolv.h    5.10 (Berkeley) 6/1/90
53119     */
53120
53121    /*
53122     * Resolver configuration file.
53123     * Normally not present, but may contain the address of the
53124     * inital name server(s) to query and the domain search list.
53125     */
53126
53127    #ifndef _PATH_RESCONF
53128    #define _PATH_RESCONF        "/etc/resolv.conf"
53129    #endif
53130
53131    /*
53132     * Global defines and variables for resolver stub.
53133     */
53134    #define MAXNS           3               /* max # name servers we'll track */
53135    #define MAXDFLSRCH      3               /* # default domain levels to try */
53136    #define MAXDNSRCH       6               /* max # domains in search path */
53137    #define LOCALDOMAINPARTS 2              /* min levels in name that is "local" */
53138
53139    #define RES_TIMEOUT     5               /* min. seconds between retries */
53140
53141    #define NAMESERVER_PORT 53
53142
53143    struct state {
53144            int     retrans;                /* retransmition time interval */
53145            int     retry;                  /* number of times to retransmit */
53146            long    options;                /* option flags - see below. */
53147            int     nscount;                /* number of name servers */
53148            ipaddr_t nsaddr_list[MAXNS];    /* address of name server */
53149    #define nsaddr  nsaddr_list[0]          /* for backward compatibility */
53150            u16_t   nsport_list[MAXNS];     /* port of name server */
53151            u16_t   id;                     /* current packet id */
53152            char    defdname[MAXDNAME];     /* default domain */
53153            char    *dnsrch[MAXDNSRCH+1];   /* components of domain to search */
53154    };
53155
53156    /*
53157     * Resolver options
53158     */
53159    #define RES_INIT        0x0001          /* address initialized */
53160    #define RES_DEBUG       0x0002          /* print debug messages */
53161    #define RES_AAONLY      0x0004          /* authoritative answers only */
53162    #define RES_USEVC       0x0008          /* use virtual circuit */
53163    #define RES_PRIMARY     0x0010          /* query primary server only */
53164    #define RES_IGNTC       0x0020          /* ignore trucation errors */
53165    #define RES_RECURSE     0x0040          /* recursion desired */
53166    #define RES_DEFNAMES    0x0080          /* use default domain name */
53167    #define RES_STAYOPEN    0x0100          /* Keep TCP socket open */
53168    #define RES_DNSRCH      0x0200          /* search up local domain tree */
53169
53170    #define RES_DEFAULT     (RES_RECURSE | RES_DEFNAMES | RES_DNSRCH )
53171
53172    extern struct state _res;
53173
53174    struct rrec;
```

```
53175
53176    int res_init _ARGS(( void ));
53177    int res_mkquery _ARGS(( int op, const char *dname, int class, int type,
53178            const char *data, int datalen, const struct rrec *newrr,
53179            char *buf, int buflen ));
53180    int res_query _ARGS(( char *name, int class, int type, u8_t *answer,
53181            int anslen ));
53182    int res_querydomain _ARGS(( char *name, char *domain, int class, int type,
53183            u8_t *answer, int anslen ));
53184    int res_search _ARGS(( char *name, int class, int type, u8_t *answer,
53185            int anslen ));
53186    int res_send _ARGS(( const char *buf, int buflen, char *answer, int anslen ));
53187    void _res_close _ARGS(( void ));
53188
53189    int dn_comp _ARGS(( const u8_t *exp_dn, u8_t *comp_dn, int length,
53190            u8_t **dnptrs, u8_t **lastdnptr ));
53191    int dn_expand  _ARGS(( const u8_t *msg, const u8_t *eomorig,
53192            const u8_t *comp_dn, u8_t *exp_dn, int length ));
53193    int __dn_skipname _ARGS(( const u8_t *comp_dn, const u8_t *eom ));
53194
53195    char *__hostalias _ARGS(( const char *name ));
53196
53197    u16_t _getshort _ARGS(( const u8_t *msgp ));
53198    u32_t _getlong _ARGS(( const u8_t *msgp ));
53199    void __putshort _ARGS(( U16_t s, u8_t *msgp ));
53200    void __putlong _ARGS(( u32_t l, u8_t *msgp ));
53201
53202    void p_query _ARGS(( char *msg ));
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/route.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
53300    /*
53301    server/ip/gen/route.h
53302    */
53303
53304    #ifndef __SERVER__IP__GEN__ROUTE_H__
53305    #define __SERVER__IP__GEN__ROUTE_H__
53306
53307    typedef struct nwio_route
53308    {
53309            u32_t nwr_ent_no;
53310            u32_t nwr_ent_count;
53311            ipaddr_t nwr_dest;
53312            ipaddr_t nwr_netmask;
53313            ipaddr_t nwr_gateway;
53314            u32_t nwr_dist;
53315            u32_t nwr_flags;
53316            u32_t nwr_pref;
53317    } nwio_route_t;
53318
53319    #define NWRF_EMPTY      0
53320    #define NWRF_INUSE      1
53321    #define NWRF_FIXED      2
53322
53323    #endif /* __SERVER__IP__GEN__ROUTE_H__ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/socket.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

53400    /*
53401    server/ip/gen/socket.h
53402    */
53403
53404    #ifndef __SERVER__IP__GEN__SOCKET_H__
53405    #define __SERVER__IP__GEN__SOCKET_H__
53406
53407    /* From SunOS: /usr/include/sys/socketh */
53408
53409    /*
53410     * Address families.
53411     */
53412    #define AF_UNSPEC       0               /* unspecified */
53413    #define AF_UNIX         1               /* local to host (pipes, portals) */
53414    #define AF_INET         2               /* internetwork: UDP, TCP, etc. */
53415    #define AF_IMPLINK      3               /* arpanet imp addresses */
53416    #define AF_PUP          4               /* pup protocols: e.g. BSP */
53417    #define AF_CHAOS        5               /* mit CHAOS protocols */
53418    #define AF_NS           6               /* XEROX NS protocols */
53419    #define AF_NBS          7               /* nbs protocols */
53420    #define AF_ECMA         8               /* european computer manufacturers */
53421    #define AF_DATAKIT      9               /* datakit protocols */
53422    #define AF_CCITT        10              /* CCITT protocols, X.25 etc */
53423    #define AF_SNA          11              /* IBM SNA */
53424    #define AF_DECnet       12              /* DECnet */
53425    #define AF_DLI          13              /* Direct data link interface */
53426    #define AF_LAT          14              /* LAT */
53427    #define AF_HYLINK       15              /* NSC Hyperchannel */
53428    #define AF_APPLETALK    16              /* Apple Talk */
53429
53430    #define AF_NIT          17              /* Network Interface Tap */
53431    #define AF_802          18              /* IEEE 802.2, also ISO 8802 */
53432    #define AF_OSI          19              /* umbrella for all families used
53433                                             * by OSI (e.g. protosw lookup) */
53434    #define AF_X25          20              /* CCITT X.25 in particular */
53435    #define AF_OSINET       21              /* AFI = 47, IDI = 4 */
53436    #define AF_GOSIP        22              /* U.S. Government OSI */
53437
53438    #define AF_MAX          21
53439
53440    #endif /* __SERVER__IP__GEN__SOCKET_H__ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/tcp.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

53500   /*
53501   server/ip/gen/tcp.h
53502   */
53503
53504   #ifndef __SERVER__IP__GEN__TCP_H__
53505   #define __SERVER__IP__GEN__TCP_H__
53506
53507   #define TCP_MIN_HDR_SIZE        20
53508   #define TCP_MAX_HDR_SIZE        60
53509
53510   #define TCPPORT_TELNET          23
53511   #define TCPPORT_FINGER          79
53512
53513   #define TCPPORT_RESERVED        1024
53514
53515   typedef u16_t tcpport_t;
53516   typedef U16_t Tcpport_t;        /* for use in prototypes */
53517
53518   #endif /* __SERVER__IP__GEN__TCP_H__ */


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/tcp_hdr.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

53600   /*
53601   server/ip/gen/tcp_hdr.h
53602   */
53603
53604   #ifndef __SERVER__IP__GEN__TCP_HDR_H__
53605   #define __SERVER__IP__GEN__TCP_HDR_H__
53606
53607   typedef struct tcp_hdr
53608   {
53609           tcpport_t th_srcport;
53610           tcpport_t th_dstport;
53611           u32_t th_seq_nr;
53612           u32_t th_ack_nr;
53613           u8_t th_data_off;
53614           u8_t th_flags;
53615           u16_t th_window;
53616           u16_t th_chksum;
53617           u16_t th_urgptr;
53618   } tcp_hdr_t;
53619
53620   #define TH_DO_MASK      0xf0
53621
53622   #define TH_FLAGS_MASK   0x3f
53623   #define THF_FIN         0x1
53624   #define THF_SYN         0x2
53625   #define THF_RST         0x4
53626   #define THF_PSH         0x8
53627   #define THF_ACK         0x10
53628   #define THF_URG         0x20
53629
```

```
53630    typedef struct tcp_hdropt
53631    {
53632            int tho_opt_siz;
53633            u8_t tho_data[TCP_MAX_HDR_SIZE-TCP_MIN_HDR_SIZE];
53634    } tcp_hdropt_t;
53635
53636    #endif /* __SERVER__IP__GEN__TCP_HDR_H__ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/tcp_io.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
53700    /*
53701    server/ip/gen/tcp_io.h
53702    */
53703
53704    #ifndef __SERVER__IP__GEN__TCP_IO_H__
53705    #define __SERVER__IP__GEN__TCP_IO_H__
53706
53707    typedef struct nwio_tcpconf
53708    {
53709            u32_t nwtc_flags;
53710            ipaddr_t nwtc_locaddr;
53711            ipaddr_t nwtc_remaddr;
53712            tcpport_t nwtc_locport;
53713            tcpport_t nwtc_remport;
53714    } nwio_tcpconf_t;
53715
53716    #define NWTC_NOFLAGS    0x0000L
53717    #define NWTC_ACC_MASK   0x0003L
53718    #       define NWTC_EXCL        0x00000001L
53719    #       define NWTC_SHARED      0x00000002L
53720    #       define NWTC_COPY        0x00000003L
53721    #define NWTC_LOCPORT_MASK       0x0030L
53722    #       define NWTC_LP_UNSET    0x00000010L
53723    #       define NWTC_LP_SET      0x00000020L
53724    #       define NWTC_LP_SEL      0x00000030L
53725    #define NWTC_REMADDR_MASK       0x0100L
53726    #       define NWTC_SET_RA      0x00000100L
53727    #       define NWTC_UNSET_RA    0x01000000L
53728    #define NWTC_REMPORT_MASK       0x0200L
53729    #       define NWTC_SET_RP      0x00000200L
53730    #       define NWTC_UNSET_RP    0x02000000L
53731
53732    typedef struct nwio_tcpcl
53733    {
53734            long nwtcl_flags;
53735            long nwtcl_ttl;
53736    } nwio_tcpcl_t;
53737
53738    typedef struct nwio_tcpatt
53739    {
53740            long nwta_flags;
53741    } nwio_tcpatt_t;
53742
53743    typedef struct nwio_tcpopt
53744    {
```

```
53745            long nwto_flags;
53746    } nwio_tcpopt_t;
53747
53748    #endif /* __SERVER__IP__GEN__TCP_IO_H__ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/udp.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
53800    /*
53801    server/ip/gen/udp.h
53802    */
53803
53804    #ifndef __SERVER__IP__GEN__UDP_H__
53805    #define __SERVER__IP__GEN__UDP_H__
53806
53807    typedef u16_t udpport_t;
53808    typedef U16_t Udpport_t;
53809
53810    #define UDP_HDR_SIZE    8
53811    #define UDP_IO_HDR_SIZE 16
53812
53813    #endif /* __SERVER__IP__GEN__UDP_H__ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/udp_hdr.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
53900    /*
53901    server/ip/gen/udp_hdr.h
53902    */
53903
53904    #ifndef __SERVER__IP__GEN__UDP_HDR_H__
53905    #define __SERVER__IP__GEN__UDP_HDR_H__
53906
53907    typedef struct udp_hdr
53908    {
53909            udpport_t uh_src_port;
53910            udpport_t uh_dst_port;
53911            u16_t uh_length;
53912            u16_t uh_chksum;
53913    } udp_hdr_t;
53914
53915    typedef struct udp_io_hdr
53916    {
53917            ipaddr_t uih_src_addr;
53918            ipaddr_t uih_dst_addr;
53919            udpport_t uih_src_port;
53920            udpport_t uih_dst_port;
53921            u16_t uih_ip_opt_len;
53922            u16_t uih_data_len;
53923    } udp_io_hdr_t;
53924
53925    #endif /* __SERVER__IP__GEN__UDP_HDR_H__ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        include/net/gen/udp_io.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

54000    /*
54001    server/ip/gen/udp_io.h
54002    */
54003
54004    #ifndef __SERVER__IP__GEN__UDP_IO_H__
54005    #define __SERVER__IP__GEN__UDP_IO_H__
54006
54007    typedef struct nwio_udpopt
54008    {
54009            unsigned long nwuo_flags;
54010            udpport_t nwuo_locport;
54011            udpport_t nwuo_remport;
54012            ipaddr_t nwuo_locaddr;
54013            ipaddr_t nwuo_remaddr;
54014    } nwio_udpopt_t;
54015
54016    #define NWUO_NOFLAGS            0x0000L
54017    #define NWUO_ACC_MASK           0x0003L
54018    #define         NWUO_EXCL               0x00000001L
54019    #define         NWUO_SHARED             0x00000002L
54020    #define         NWUO_COPY               0x00000003L
54021    #define NWUO_LOCPORT_MASK       0x000CL
54022    #define         NWUO_LP_SEL             0x00000004L
54023    #define         NWUO_LP_SET             0x00000008L
54024    #define         NWUO_LP_ANY             0x0000000CL
54025    #define NWUO_LOCADDR_MASK       0x0010L
54026    #define         NWUO_EN_LOC             0x00000010L
54027    #define         NWUO_DI_LOC             0x00100000L
54028    #define NWUO_BROAD_MASK         0x0020L
54029    #define         NWUO_EN_BROAD           0x00000020L
54030    #define         NWUO_DI_BROAD           0x00200000L
54031    #define NWUO_REMPORT_MASK       0x0100L
54032    #define         NWUO_RP_SET             0x00000100L
54033    #define         NWUO_RP_ANY             0x01000000L
54034    #define NWUO_REMADDR_MASK       0x0200L
54035    #define         NWUO_RA_SET             0x00000200L
54036    #define         NWUO_RA_ANY             0x02000000L
54037    #define NWUO_RW_MASK            0x1000L
54038    #define         NWUO_RWDATONLY          0x00001000L
54039    #define         NWUO_RWDATALL           0x10000000L
54040    #define NWUO_IPOPT_MASK         0x2000L
54041    #define         NWUO_EN_IPOPT           0x00002000L
54042    #define         NWUO_DI_IPOPT           0x20000000L
54043
54044    #endif /* __SERVER__IP__GEN__UDP_IO_H__ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/config.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

54100    /*      config.h - Enable or disable TCP/IP services.   Author: Kees J. Bot
54101     */
54102
54103    #define ENABLE_ARP      1
54104    #define ENABLE_IP       1
54105    #define ENABLE_TCP      1
54106    #define ENABLE_UDP      1



++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/const.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

54200    /*
54201    inet/const.h
54202
54203    Created:        Dec 30, 1991 by Philip Homburg
54204    */
54205
54206    #ifndef INET__CONST_H
54207    #define INET__CONST_H
54208
54209    #ifndef DEBUG
54210    #define DEBUG   0
54211    #endif
54212
54213    #ifndef NDEBUG
54214    #define NDEBUG  (!DEBUG)
54215    #endif
54216
54217    #define printf  printk
54218
54219    #define where() printf("%s, %d: ", __FILE__, __LINE__)
54220
54221    #define NW_SUSPEND      SUSPEND
54222    #define NW_OK           OK
54223
54224    #define THIS_PROC       INET_PROC_NR
54225
54226    #define BUF_S           512
54227
54228    #endif /* INET__CONST_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/inet.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

54300    /*
54301    inet/inet.h
54302
54303    Created:        Dec 30, 1991 by Philip Homburg
54304    */
54305
54306    #ifndef INET__NW_TASK_H
54307    #define INET__NW_TASK_H
54308
54309    #define _MINIX  1
54310    #define _SYSTEM 1       /* get OK and negative error codes */
54311
54312    #include <sys/types.h>
54313    #include <errno.h>
54314    #include <stddef.h>
54315    #include <stdlib.h>
54316    #include <string.h>
54317
54318    #include <ansi.h>
54319    #include <minix/config.h>
54320    #include <minix/type.h>
54321
54322    #include <minix/com.h>
54323    #include <minix/const.h>
54324    #include <minix/syslib.h>
54325    #include <net/hton.h>
54326    #include <net/gen/ether.h>
54327    #include <net/gen/eth_hdr.h>
54328    #include <net/gen/eth_io.h>
54329    #include <net/gen/in.h>
54330    #include <net/gen/ip_hdr.h>
54331    #include <net/gen/ip_io.h>
54332    #include <net/gen/icmp.h>
54333    #include <net/gen/icmp_hdr.h>
54334    #include <net/gen/oneCsum.h>
54335    #include <net/gen/route.h>
54336    #include <net/gen/tcp.h>
54337    #include <net/gen/tcp_hdr.h>
54338    #include <net/gen/tcp_io.h>
54339    #include <net/gen/udp.h>
54340    #include <net/gen/udp_hdr.h>
54341    #include <net/gen/udp_io.h>
54342    #include <sys/ioctl.h>
54343
54344    /* Ioctl's may contain size and type encoding.  It pays to extract the type. */
54345    #ifdef _IOCTYPE_MASK
54346    #define IOCTYPE_MASK    _IOCTYPE_MASK
54347    #define IOCPARM_MASK    _IOCPARM_MASK
54348    #else
54349    #define IOCTYPE_MASK    0xFFFF
54350    #endif
54351
54352    #include "const.h"
54353
54354    #define PUBLIC
```

```
54355    #define EXTERN  extern
54356    #define PRIVATE static
54357    #define FORWARD static
54358
54359    #define INIT_PANIC()    static char *ip_panic_warning_file= __FILE__
54360
54361    #define ip_panic(print_list)  \
54362            ( \
54363                    printf("panic at %s, %d: ", ip_panic_warning_file, __LINE__), \
54364                    printf print_list, \
54365                    printf("\n"), \
54366                    abort(), \
54367                    0 \
54368            )
54369
54370    #define ip_warning(print_list)  \
54371            ( \
54372                    printf("warning at %s, %d: ", ip_panic_warning_file, \
54373                                                                    __LINE__), \
54374                    printf print_list, \
54375                    printf("\n"), \
54376                    0 \
54377            )
54378
54379    #if _ANSI
54380    #define ARGS(x) x
54381    #else /* _ANSI */
54382    #define ARGS(x) ()
54383    #endif /* _ANSI */
54384
54385    #endif /* INET__NW_TASK_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/mq.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
54400    /*
54401    inet/mq.h
54402
54403    Created:        Jan 3, 1992 by Philip Homburg
54404    */
54405
54406    #ifndef INET__MQ_H
54407    #define INET__MQ_H
54408
54409    typedef struct mq
54410    {
54411            message mq_mess;
54412            struct mq *mq_next;
54413    } mq_t;
54414
54415    _PROTOTYPE( mq_t *mq_get, (void) );
54416    _PROTOTYPE( void mq_free, (mq_t *mq) );
54417    _PROTOTYPE( void mq_init, (void) );
54418
54419    #endif /* INET__MQ_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/osdep_eth.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

54500    /*
54501    inet/osdep_eth.h
54502
54503    Created:        Dec 30, 1991 by Philip Homburg
54504    */
54505
54506    #ifndef INET__OSDEP_ETH_H
54507    #define INET__OSDEP_ETH_H
54508
54509    #define IOVEC_NR        16
54510    #define RD_IOVEC        ((ETH_MAX_PACK_SIZE + BUF_S -1)/BUF_S)
54511
54512    typedef struct osdep_eth_port
54513    {
54514            int etp_minor;
54515            int etp_task;
54516            int etp_port;
54517            iovec_t etp_wr_iovec[IOVEC_NR];
54518            iovec_t etp_rd_iovec[RD_IOVEC];
54519    } osdep_eth_port_t;
54520
54521    #endif /* INET__OSDEP_ETH_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/proto.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

54600    /*
54601    inet/proto.h
54602
54603    Created:        Jan 2, 1992 by Philip Homburg
54604    */
54605
54606    /* clock.c */
54607
54608    _PROTOTYPE( void clck_tick, (message *mess) );
54609
54610    /* mnx_eth.c */
54611
54612    _PROTOTYPE( void eth_rec, (message *m) );
54613
54614    /* sr.c */
54615
54616    struct mq;
54617    _PROTOTYPE( void sr_rec, (struct mq *m) );
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/buf.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

54700    /*
54701    This file contains routines for buffer management.
54702    */
54703
54704    #include "inet.h"
54705
54706    #include <stdlib.h>
54707    #include <string.h>
54708
54709    #include "generic/assert.h"
54710    #include "generic/buf.h"
54711    #include "generic/type.h"
54712
54713    INIT_PANIC();
54714
54715    #if TRACE_ENQUEUE_PROBLEM
54716    extern enqueue_problem;
54717    #endif
54718
54719    #define USE_MALLOCS     0
54720
54721    #ifndef BUF512_NR
54722    #define BUF512_NR       (sizeof(int) == 2 ? 40 : 128)
54723    #endif
54724
54725    #define ACC_NR          200
54726    #define CLIENT_NR       5
54727
54728    typedef struct buf512
54729    {
54730            buf_t buf_header;
54731            char buf_data[512];
54732    } buf512_t;
54733
54734    #if USE_MALLOCS
54735    PRIVATE buf512_t *buffers512;
54736    PRIVATE acc_t *accessors;
54737    #else
54738    PRIVATE buf512_t buffers512[BUF512_NR];
54739    PRIVATE acc_t accessors[ACC_NR];
54740    #endif
54741
54742    PRIVATE buf512_t *buf512_free;
54743
54744    PRIVATE bf_freereq_t freereq[CLIENT_NR];
54745    PRIVATE size_t bf_buf_gran;
54746    PRIVATE acc_t *acc_free_list;
54747
54748    PUBLIC size_t bf_free_buffsize;
54749    PUBLIC acc_t *bf_temporary_acc;
54750
54751
54752    #ifdef bf_memreq
54753    PUBLIC char *bf_memreq_file;
54754    PUBLIC int bf_memreq_line;
```

```
54755   #endif
54756   #ifdef bf_cut
54757   PUBLIC char *bf_cut_file;
54758   PUBLIC int bf_cut_line;
54759   #endif
54760   #ifdef bf_packIffLess
54761   PUBLIC char *bf_pack_file;
54762   PUBLIC int bf_pack_line;
54763   #endif
54764   #ifdef bf_bufsize
54765   PUBLIC char *bf_bufsize_file;
54766   PUBLIC int bf_bufsize_line;
54767   #endif
54768
54769   FORWARD acc_t *bf_small_memreq ARGS(( size_t size ));
54770   FORWARD void bf_512free ARGS(( buf_t *buffer ));
54771
54772   PUBLIC void bf_init()
54773   {
54774           int i;
54775           size_t size;
54776           size_t buf_s;
54777
54778           bf_buf_gran= BUF_S;
54779           buf_s= 0;
54780
54781   #if USE_MALLOCS
54782           printf("buf.c: malloc %d 32K-buffers (%dK)\n", BUF32K_NR,
54783                   sizeof(*buffers32K) * BUF32K_NR / 1024);
54784           buffers32K= malloc(sizeof(*buffers32K) * BUF32K_NR);
54785           if (!buffers32K)
54786                   ip_panic(( "unable to alloc 32K-buffers" ));
54787           printf("buf.c: malloc %d 2K-buffers (%dK)\n", BUF2K_NR,
54788                   sizeof(*buffers2K) * BUF2K_NR / 1024);
54789           buffers2K= malloc(sizeof(*buffers2K) * BUF2K_NR);
54790           if (!buffers2K)
54791                   ip_panic(( "unable to alloc 2K-buffers" ));
54792           printf("buf.c: malloc %d 512-buffers (%dK)\n", BUF512_NR,
54793                   sizeof(*buffers512) * BUF512_NR / 1024);
54794           buffers512= malloc(sizeof(*buffers512) * BUF512_NR);
54795           if (!buffers512)
54796                   ip_panic(( "unable to alloc 512-buffers" ));
54797           printf("buf.c: malloc %d accessors (%dK)\n", ACC_NR,
54798                   sizeof(*accessors) * ACC_NR / 1024);
54799           accessors= malloc(sizeof(*accessors) * ACC_NR);
54800           if (!accessors)
54801                   ip_panic(( "unable to alloc accessors" ));
54802   #endif
54803
54804           for (i=0;i<BUF512_NR;i++)
54805           {
54806                   buffers512[i].buf_header.buf_linkC= 0;
54807                   buffers512[i].buf_header.buf_next= &buffers512[i+1];
54808                   buffers512[i].buf_header.buf_free= bf_512free;
54809                   buffers512[i].buf_header.buf_size= sizeof(buffers512[i].
54810                           buf_data);
54811                   buffers512[i].buf_header.buf_data_p= buffers512[i].buf_data;
54812           }
54813           buffers512[i-1].buf_header.buf_next= 0;
54814           buf512_free= &buffers512[0];
```

```
54815           if (sizeof(buffers512[0].buf_data) < bf_buf_gran)
54816                   bf_buf_gran= sizeof(buffers512[0].buf_data);
54817           if (sizeof(buffers512[0].buf_data) > buf_s)
54818                   buf_s= sizeof(buffers512[0].buf_data);
54819
54820           for (i=0;i<ACC_NR;i++)
54821           {
54822                   accessors[i].acc_linkC= 0;
54823                   accessors[i].acc_next= &accessors[i+1];
54824           }
54825           acc_free_list= accessors;
54826           accessors[i-1].acc_next= 0;
54827
54828           for (i=0;i<CLIENT_NR;i++)
54829                   freereq[i]=0;
54830
54831           assert (buf_s == BUF_S);
54832   }

54834   PUBLIC void bf_logon(func)
54835   bf_freereq_t func;
54836   {
54837           int i;
54838
54839           for (i=0;i<CLIENT_NR;i++)
54840                   if (!freereq[i])
54841                   {
54842                           freereq[i]=func;
54843                           return;
54844                   }
54845
54846           ip_panic(( "buf.c: to many clients" ));
54847   }

54849   /*
54850   bf_memreq
54851   */
54852
54853   #ifndef bf_memreq
54854   PUBLIC acc_t *bf_memreq(size)
54855   #else
54856   PUBLIC acc_t *_bf_memreq(size)
54857   #endif
54858   size_t size;
54859   {
54860           acc_t *head, *tail, *new_acc;
54861           int i,j;
54862           size_t count;
54863
54864   #if TRACE_ENQUEUE_PROBLEM
54865    { if (enqueue_problem)
54866      { where(); printf("bf_memreq(%d) called with enqueue_problem\n", size); } }
54867   #endif
54868   #ifdef bf_memreq
54869    { where(); printf("bf_memreq(%d) called by %s, %d\n", size, bf_memreq_file,
54870           bf_memreq_line); }
54871   #endif
54872   assert (size>0);
54873
54874           head= NULL;
```

```
54875           while (size)
54876           {
54877                   if (!acc_free_list)
54878                   {
54879   #if DEBUG
54880    { where(); printf("freeing accessors\n"); }
54881   #endif
54882                           for (i=0; !acc_free_list && i<MAX_BUFREQ_PRI; i++)
54883                           {
54884                                   for (j=0; !acc_free_list && j<CLIENT_NR; j++)
54885                                   {
54886                                           bf_free_buffsize= 0;
54887                                           if (freereq[j])
54888                                                   (*freereq[j])(i, BUF_S);
54889                                   }
54890                           }
54891                   }
54892                   if (!acc_free_list)
54893                           ip_panic(( "To few accessors" ));
54894                   new_acc= acc_free_list;
54895                   acc_free_list= acc_free_list->acc_next;
54896   #if DEBUG & 256
54897    { where(); printf("got accessor %d\n", new_acc-accessors); }
54898   #endif
54899                   new_acc->acc_linkC= 1;
54900                   new_acc->acc_buffer= 0;
54901
54902   #if DEBUG & 256
54903    { where(); printf("looking for 512 byte buffer\n"); }
54904   #endif
54905                   if (buf512_free)
54906                   {
54907                           buf512_t *buf512;
54908
54909   #if DEBUG & 256
54910    { where(); printf("found a 512 byte buffer\n"); }
54911   #endif
54912                           buf512= buf512_free;
54913                           buf512_free= buf512->buf_header.buf_next;
54914   assert (!buf512->buf_header.buf_linkC);
54915                           buf512->buf_header.buf_linkC= 1;
54916   assert (buf512->buf_header.buf_free == bf_512free);
54917   assert (buf512->buf_header.buf_size == sizeof(buf512->buf_data));
54918   assert (buf512->buf_header.buf_data_p == buf512->buf_data);
54919                           new_acc->acc_buffer= &buf512->buf_header;
54920                           buf512->buf_header.buf_next= buf512;
54921                   }
54922   #if DEBUG
54923                   else
54924    { where(); printf("unable to find a 512 byte buffer\n"); }
54925   #endif
54926                   if (!new_acc->acc_buffer)
54927                   {
54928   #if DEBUG
54929    { where(); printf("freeing buffers\n"); }
54930   #endif
54931                           bf_free_buffsize= 0;
54932                           for (i=0; bf_free_buffsize<size && i<MAX_BUFREQ_PRI;
54933                                   i++)
54934                                   for (j=0; bf_free_buffsize<size && j<CLIENT_NR;
```

```
54935                                           j++)
54936                                           if (freereq[j])
54937                                                   (*freereq[j])(i, size);
54938
54939                           if (bf_free_buffsize<size)
54940                                   ip_panic(( "not enough buffers freed" ));
54941
54942                           continue;
54943                   }
54944
54945
54946                   if (!head)
54947                           head= new_acc;
54948                   else
54949                           tail->acc_next= new_acc;
54950                   tail= new_acc;
54951
54952                   count= tail->acc_buffer->buf_size;
54953                   if (count > size)
54954                           count= size;
54955
54956                   tail->acc_offset= 0;
54957                   tail->acc_length=  count;
54958                   size -= count;
54959           }
54960           tail->acc_next= 0;
54961   #if DEBUG
54962   bf_chkbuf(head);
54963   #endif
54964   #if DEBUG & 256
54965    { where(); printf("acc 0x%x has buffer 0x%x\n", head, head->acc_buffer); }
54966   #endif
54967           return head;
54968   }

54970   /*
54971   bf_small_memreq
54972   */
54973
54974   PRIVATE acc_t *bf_small_memreq(size)
54975   size_t size;
54976   {
54977           acc_t *head, *tail, *new_acc;
54978           int i,j;
54979           size_t count;
54980
54981   #if TRACE_ENQUEUE_PROBLEM
54982    { if (enqueue_problem)
54983     { where(); printf("bf_small_memreq(%d) called with enqueue_problem\n", size);
54984           } }
54985   #endif
54986   #if DEBUG & 256
54987    { where(); printf("bf_small_memreq(%d)\n", size); }
54988   #endif
54989
54990   assert (size>0);
54991
54992           head= NULL;
54993           while (size)
54994           {
```

```
54995                   if (!acc_free_list)
54996                   {
54997   #if DEBUG
54998    { where(); printf("freeing accessors\n"); }
54999   #endif
55000                           for (i=0; !acc_free_list && i<MAX_BUFREQ_PRI; i++)
55001                           {
55002                                   for (j=0; !acc_free_list && j<CLIENT_NR; j++)
55003                                   {
55004                                           bf_free_buffsize= 0;
55005                                           if (freereq[j])
55006                                                   (*freereq[j])(i, BUF_S);
55007                                   }
55008                           }
55009                   }
55010                   new_acc= acc_free_list;
55011                   if (!new_acc)
55012                           ip_panic(( "buf.c: out of accessors" ));
55013                   acc_free_list= new_acc->acc_next;
55014   #if DEBUG & 256
55015    { where(); printf("got accessor %d\n", new_acc-accessors); }
55016   #endif
55017                   new_acc->acc_linkC= 1;
55018
55019                   if (size >= sizeof(buf512_free->buf_data))
55020                   {
55021                           if (buf512_free)
55022                           {
55023                                   buf512_t *buf512;
55024
55025   #if DEBUG & 256
55026    { where(); printf("found a 512 byte buffer\n"); }
55027   #endif
55028                                   buf512= buf512_free;
55029                                   buf512_free= buf512->buf_header.buf_next;
55030   assert (!buf512->buf_header.buf_linkC);
55031                                   buf512->buf_header.buf_linkC= 1;
55032   assert (buf512->buf_header.buf_free == bf_512free);
55033   assert (buf512->buf_header.buf_size == sizeof(buf512->buf_data));
55034   assert (buf512->buf_header.buf_data_p == buf512->buf_data);
55035                                   new_acc->acc_buffer= &buf512->buf_header;
55036                                   buf512->buf_header.buf_next= buf512;
55037                           }
55038                           else
55039                                   break;
55040                   }
55041                   else
55042                           break;
55043
55044                   if (!head)
55045                           head= new_acc;
55046                   else
55047                           tail->acc_next= new_acc;
55048                   tail= new_acc;
55049
55050                   count= tail->acc_buffer->buf_size;
55051                   if (count > size)
55052                           count= size;
55053
55054                   tail->acc_offset= 0;
```

```
55055                   tail->acc_length=  count;
55056                   size -= count;
55057           }
55058           if (size)
55059           {
55060                   new_acc->acc_linkC= 0;
55061                   new_acc->acc_next= acc_free_list;
55062                   acc_free_list= new_acc;
55063                   new_acc= bf_memreq(size);
55064                   if (!head)
55065                           head= new_acc;
55066                   else
55067                           tail->acc_next= new_acc;
55068           }
55069           else
55070                   tail->acc_next= 0;
55071           return head;
55072   }

55074   PUBLIC void bf_afree(acc_ptr)
55075   acc_t *acc_ptr;
55076   {
55077           acc_t *tmp_acc;
55078           buf_t *tmp_buf;
55079
55080           while (acc_ptr)
55081           {
55082   assert (acc_ptr->acc_linkC);
55083                   acc_ptr->acc_linkC--;
55084                   if (!acc_ptr->acc_linkC)
55085                   {
55086                           tmp_buf= acc_ptr->acc_buffer;
55087   assert (tmp_buf);
55088   assert (tmp_buf->buf_linkC);
55089                           if (!--tmp_buf->buf_linkC)
55090                           {
55091                                   bf_free_buffsize += tmp_buf->buf_size;
55092                                   tmp_buf->buf_free(tmp_buf);
55093                           }
55094                           tmp_acc= acc_ptr;
55095                           acc_ptr= acc_ptr->acc_next;
55096                           tmp_acc->acc_next= acc_free_list;
55097                           acc_free_list= tmp_acc;
55098                   }
55099                   else
55100                           break;
55101           }
55102   }

55104   PUBLIC acc_t *bf_dupacc(acc_ptr)
55105   register acc_t *acc_ptr;
55106   {
55107           register acc_t *new_acc;
55108           int i, j;
55109
55110   #if TRACE_ENQUEUE_PROBLEM
55111    { if (enqueue_problem)
55112      { where(); printf("bf_dupacc(0x%x) called with enqueue_problem\n", acc_ptr);
55113           } }
55114   #endif
```

```
55115
55116
55117           if (!acc_free_list)
55118           {
55119   #if DEBUG
55120            { where(); printf("freeing accessors\n"); }
55121   #endif
55122                   for (i=0; !acc_free_list && i<MAX_BUFREQ_PRI; i++)
55123                   {
55124                           for (j=0; !acc_free_list && j<CLIENT_NR; j++)
55125                           {
55126                                   bf_free_buffsize= 0;
55127                                   if (freereq[j])
55128                                           (*freereq[j])(i, BUF_S);
55129                           }
55130                   }
55131           }
55132           new_acc= acc_free_list;
55133           if (!new_acc)
55134                   ip_panic(( "buf.c: out of accessors" ));
55135           acc_free_list= new_acc->acc_next;
55136   #if DEBUG & 256
55137            { where(); printf("got accessor %d\n", new_acc-accessors); }
55138   #endif
55139
55140           *new_acc= *acc_ptr;
55141           if (acc_ptr->acc_next)
55142                   acc_ptr->acc_next->acc_linkC++;
55143           if (acc_ptr->acc_buffer)
55144                   acc_ptr->acc_buffer->buf_linkC++;
55145           new_acc->acc_linkC= 1;
55146           return new_acc;
55147   }

55149   #ifdef bf_bufsize
55150   PUBLIC size_t _bf_bufsize(acc_ptr)
55151   #else
55152   PUBLIC size_t bf_bufsize(acc_ptr)
55153   #endif
55154   register acc_t *acc_ptr;
55155   {
55156           register size_t size;
55157
55158   #ifdef bf_bufsize
55159            { where(); printf("bf_bufsize(0x%x) called by %s, %d\n", acc_ptr,
55160           bf_bufsize_file, bf_bufsize_line); }
55161   #endif
55162
55163   assert(acc_ptr);
55164
55165           size=0;
55166
55167           while (acc_ptr)
55168           {
55169   assert(acc_ptr >= accessors && acc_ptr <= &accessors[ACC_NR-1]);
55170                   size += acc_ptr->acc_length;
55171                   acc_ptr= acc_ptr->acc_next;
55172           }
55173   #if DEBUG & 256
55174            { where(); printf("bf_bufsize(...)= %d\n", size); }
```

```
55175   #endif
55176           return size;
55177   }

55179   #ifndef bf_packIffLess
55180   PUBLIC acc_t *bf_packIffLess(pack, min_len)
55181   #else
55182   PUBLIC acc_t *_bf_packIffLess(pack, min_len)
55183   #endif
55184   acc_t *pack;
55185   int min_len;
55186   {
55187           if (!pack || pack->acc_length >= min_len)
55188                   return pack;
55189
55190   #ifdef bf_packIffLess
55191    { where(); printf("calling bf_pack because of %s %d: %d\n", bf_pack_file,
55192           bf_pack_line, min_len); }
55193   #endif
55194           return bf_pack(pack);
55195   }

55197   PUBLIC acc_t *bf_pack(old_acc)
55198   acc_t *old_acc;
55199   {
55200           acc_t *new_acc, *acc_ptr_old, *acc_ptr_new;
55201           size_t size, offset_old, offset_new, block_size, block_size_old;
55202
55203           /* Check if old acc is good enough. */
55204           if (!old_acc || !old_acc->acc_next && old_acc->acc_linkC == 1 &&
55205                   (!old_acc->acc_buffer || old_acc->acc_buffer->buf_linkC == 1))
55206                   return old_acc;
55207
55208           size= bf_bufsize(old_acc);
55209           new_acc= bf_memreq(size);
55210           acc_ptr_old= old_acc;
55211           acc_ptr_new= new_acc;
55212           offset_old= 0;
55213           offset_new= 0;
55214           while (size)
55215           {
55216   assert (acc_ptr_old);
55217                   if (offset_old == acc_ptr_old->acc_length)
55218                   {
55219                           offset_old= 0;
55220                           acc_ptr_old= acc_ptr_old->acc_next;
55221                           continue;
55222                   }
55223   assert (offset_old < acc_ptr_old->acc_length);
55224                   block_size_old= acc_ptr_old->acc_length - offset_old;
55225   assert (acc_ptr_new);
55226                   if (offset_new == acc_ptr_new->acc_length)
55227                   {
55228                           offset_new= 0;
55229                           acc_ptr_new= acc_ptr_new->acc_next;
55230                           continue;
55231                   }
55232   assert (offset_new < acc_ptr_new->acc_length);
55233                   block_size= acc_ptr_new->acc_length - offset_new;
55234                   if (block_size > block_size_old)
```

```
55235                           block_size= block_size_old;
55236                   memcpy(ptr2acc_data(acc_ptr_new)+offset_new,
55237                           ptr2acc_data(acc_ptr_old)+offset_old, block_size);
55238                   offset_new += block_size;
55239                   offset_old += block_size;
55240                   size -= block_size;
55241           }
55242           bf_afree(old_acc);
55243           return new_acc;
55244   }

55246   #ifndef bf_cut
55247   PUBLIC acc_t *bf_cut (data, offset, length)
55248   #else
55249   PUBLIC acc_t *_bf_cut (data, offset, length)
55250   #endif
55251   register acc_t *data;
55252   register unsigned offset;
55253   register unsigned length;
55254   {
55255           register acc_t *head, *tail;
55256
55257   #if DEBUG & 256
55258    { where(); printf("bf_cut(.., %u, %u) called\n", offset, length); }
55259   #ifdef bf_cut
55260    { where(); printf("bf_cut_file= %s, bf_cut_line= %d\n", bf_cut_file,
55261           bf_cut_line); }
55262   #endif
55263   #endif
55264           if (!data && !offset && !length)
55265                   return 0;
55266   #ifdef bf_cut
55267   if (!data)
55268    { where(); printf("bf_cut_file= %s, bf_cut_line= %d\n", bf_cut_file,
55269           bf_cut_line); }
55270   #endif
55271   assert(data);
55272   #if DEBUG
55273   bf_chkbuf(data);
55274   #endif
55275           if (!length)
55276           {
55277                   head= bf_dupacc(data);
55278                   bf_afree(head->acc_next);
55279                   head->acc_next= 0;
55280                   head->acc_length= 0;
55281   #if DEBUG
55282   bf_chkbuf(data);
55283   #endif
55284                   return head;
55285           }
55286           while (data && offset>=data->acc_length)
55287           {
55288                   offset -= data->acc_length;
55289                   data= data->acc_next;
55290           }
55291   #ifdef bf_cut
55292   if (!data)
55293    { where(); printf("bf_cut_file= %s, bf_cut_line= %d\n", bf_cut_file,
55294           bf_cut_line); }
```

```
55295   #endif
55296   assert (data);
55297           head= bf_dupacc(data);
55298           bf_afree(head->acc_next);
55299           head->acc_next= 0;
55300           head->acc_offset += offset;
55301           head->acc_length -= offset;
55302           if (length >= head->acc_length)
55303                   length -= head->acc_length;
55304           else
55305           {
55306                   head->acc_length= length;
55307                   length= 0;
55308           }
55309           tail= head;
55310           data= data->acc_next;
55311           while (data && length && length>=data->acc_length)
55312           {
55313                   tail->acc_next= bf_dupacc(data);
55314                   tail= tail->acc_next;
55315                   bf_afree(tail->acc_next);
55316                   tail->acc_next= 0;
55317                   data= data->acc_next;
55318                   length -= tail->acc_length;
55319           }
55320           if (length)
55321           {
55322   #ifdef bf_cut
55323   if (!data)
55324    { where(); printf("bf_cut_file= %s, bf_cut_line= %d\n", bf_cut_file,
55325           bf_cut_line); }
55326   #endif
55327   assert (data);
55328                   tail->acc_next= bf_dupacc(data);
55329                   tail= tail->acc_next;
55330                   bf_afree(tail->acc_next);
55331                   tail->acc_next= 0;
55332                   tail->acc_length= length;
55333           }
55334   #if DEBUG
55335   bf_chkbuf(data);
55336   #endif
55337           return head;
55338   }

55340   /*
55341   bf_append
55342   */
55343
55344   PUBLIC acc_t *bf_append(data_first, data_second)
55345   acc_t *data_first;
55346   acc_t  *data_second;
55347   {
55348           acc_t *head, *tail, *new_acc, *acc_ptr_new, tmp_acc, *curr;
55349           char *src_ptr, *dst_ptr;
55350           size_t size, offset_old, offset_new, block_size_old, block_size;
55351
55352   #if TRACE_ENQUEUE_PROBLEM
55353    { if (enqueue_problem)
55354      { where(); printf("bf_append(0x%x, 0x%x) called with enqueue_problem\n",
```

```
55355           data_first, data_second); } }
55356   #endif
55357   #if DEBUG & 256
55358    { where(); printf("BF_Append(0x%x, 0x%x) called\n", data_first, data_second); }
55359   #endif
55360           if (!data_first)
55361                   return data_second;
55362           if (!data_second)
55363                   return data_first;
55364
55365           head= 0;
55366           while (data_first)
55367           {
55368                   if (data_first->acc_linkC == 1)
55369                           curr= data_first;
55370                   else
55371                   {
55372                           curr= bf_dupacc(data_first);
55373                           assert (curr->acc_linkC == 1);
55374                           bf_afree(data_first);
55375                   }
55376                   data_first= curr->acc_next;
55377                   if (!curr->acc_length)
55378                   {
55379                           curr->acc_next= 0;
55380                           bf_afree(curr);
55381                           continue;
55382                   }
55383                   if (!head)
55384                           head= curr;
55385                   else
55386                           tail->acc_next= curr;
55387                   tail= curr;
55388           }
55389           if (!head)
55390                   return data_second;
55391           tail->acc_next= 0;
55392
55393           while (data_second && !data_second->acc_length)
55394           {
55395                   curr= data_second;
55396                   data_second= data_second->acc_next;
55397                   if (data_second)
55398                           data_second->acc_linkC++;
55399                   bf_afree(curr);
55400           }
55401           if (!data_second)
55402                   return head;
55403
55404           if (tail->acc_length + data_second->acc_length >
55405                   tail->acc_buffer->buf_size)
55406           {
55407                   tail->acc_next= data_second;
55408                   return head;
55409           }
55410
55411           if (tail->acc_buffer->buf_size == bf_buf_gran &&
55412                   tail->acc_buffer->buf_linkC == 1)
55413           {
55414                   if (tail->acc_offset)
```

```
55415                   {
55416                           memmove(tail->acc_buffer->buf_data_p,
55417                                   ptr2acc_data(tail), tail->acc_length);
55418                           tail->acc_offset= 0;
55419                   }
55420                   dst_ptr= ptr2acc_data(tail) + tail->acc_length;
55421                   src_ptr= ptr2acc_data(data_second);
55422                   memcpy(dst_ptr, src_ptr, data_second->acc_length);
55423                   tail->acc_length += data_second->acc_length;
55424                   tail->acc_next= data_second->acc_next;
55425                   if (data_second->acc_next)
55426                           data_second->acc_next->acc_linkC++;
55427                   bf_afree(data_second);
55428                   return head;
55429           }
55430
55431           new_acc= bf_small_memreq(tail->acc_length+data_second->acc_length);
55432           acc_ptr_new= new_acc;
55433           offset_old= 0;
55434           offset_new= 0;
55435           size= tail->acc_length;
55436           while (size)
55437           {
55438   assert (acc_ptr_new);
55439                   if (offset_new == acc_ptr_new->acc_length)
55440                   {
55441                           offset_new= 0;
55442                           acc_ptr_new= acc_ptr_new->acc_next;
55443                           continue;
55444                   }
55445   assert (offset_new < acc_ptr_new->acc_length);
55446   assert (offset_old < tail->acc_length);
55447                   block_size_old= tail->acc_length - offset_old;
55448                   block_size= acc_ptr_new->acc_length - offset_new;
55449                   if (block_size > block_size_old)
55450                           block_size= block_size_old;
55451                   memcpy(ptr2acc_data(acc_ptr_new)+offset_new,
55452                           ptr2acc_data(tail)+offset_old, block_size);
55453                   offset_new += block_size;
55454                   offset_old += block_size;
55455                   size -= block_size;
55456           }
55457           offset_old= 0;
55458           size= data_second->acc_length;
55459           while (size)
55460           {
55461   assert (acc_ptr_new);
55462                   if (offset_new == acc_ptr_new->acc_length)
55463                   {
55464                           offset_new= 0;
55465                           acc_ptr_new= acc_ptr_new->acc_next;
55466                           continue;
55467                   }
55468   assert (offset_new < acc_ptr_new->acc_length);
55469   assert (offset_old < data_second->acc_length);
55470                   block_size_old= data_second->acc_length - offset_old;
55471                   block_size= acc_ptr_new->acc_length - offset_new;
55472                   if (block_size > block_size_old)
55473                           block_size= block_size_old;
55474                   memcpy(ptr2acc_data(acc_ptr_new)+offset_new,
```

```
55475                           ptr2acc_data(data_second)+offset_old, block_size);
55476                   offset_new += block_size;
55477                   offset_old += block_size;
55478                   size -= block_size;
55479           }
55480           tmp_acc= *tail;
55481           *tail= *new_acc;
55482           *new_acc= tmp_acc;
55483
55484           bf_afree(new_acc);
55485           while (tail->acc_next)
55486                   tail= tail->acc_next;
55487
55488           tail->acc_next= data_second->acc_next;
55489           if (data_second->acc_next)
55490                   data_second->acc_next->acc_linkC++;
55491           bf_afree(data_second);
55492           return head;
55493   }

55495   PRIVATE void bf_512free(buffer)
55496   buf_t *buffer;
55497   {
55498           buf512_t *buf512;
55499
55500           buf512= buffer->buf_next;
55501           buf512->buf_header.buf_next= buf512_free;
55502           buf512_free= buf512;
55503   }

55505   PUBLIC void bf_check_all_bufs()
55506   {
55507           int j;
55508           int accs;
55509           acc_t *acc;
55510           int bufs;
55511           buf512_t *buf512;
55512
55513           for (j=0; j<CLIENT_NR; j++)
55514           {
55515                   if (freereq[j])
55516                           (*freereq[j])(-1, 0);
55517           }
55518
55519           /* Check the number of accessors */
55520           accs= 0;
55521           for(acc= acc_free_list; acc; acc= acc->acc_next)
55522                   accs++;
55523           printf("number of free accs is %d, expected %d\n", accs, ACC_NR);
55524
55525           /* Check the number of 512 byte buffers */
55526           bufs= 0;
55527           for(buf512= buf512_free; buf512; buf512=
55528                                           (buf512_t *)buf512->buf_header.buf_next)
55529                   bufs++;
55530
55531           printf("number of free 512 byte buffers is %d, expected %d\n", bufs,
55532                   BUF512_NR);
55533   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/clock.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

55600    /* clock.c */
55601
55602    #include "inet.h"
55603    #include "proto.h"
55604    #include "generic/assert.h"
55605    #include "generic/buf.h"
55606    #include "generic/clock.h"
55607    #include "generic/type.h"
55608
55609    INIT_PANIC();
55610
55611    FORWARD _PROTOTYPE( void clck_fast_release, (timer_t *timer) );
55612    FORWARD _PROTOTYPE( void set_timer, (void) );
55613
55614    PRIVATE time_t curr_time;
55615    PRIVATE timer_t *timer_chain;
55616    PRIVATE time_t next_timeout;
55617
55618    PUBLIC time_t get_time()
55619    {
55620            if (!curr_time)
55621            {
55622                    static message mess;
55623
55624                    mess.m_type= GET_UPTIME;
55625                    if (sendrec (CLOCK, &mess) < 0)
55626                            ip_panic(("unable to sendrec"));
55627                    if (mess.m_type != OK)
55628                            ip_panic(("can't read clock"));
55629                    curr_time= mess.NEW_TIME;
55630            }
55631            return curr_time;
55632    }

55634    PUBLIC void set_time (tim)
55635    time_t tim;
55636    {
55637            curr_time= tim;
55638    }

55640    PUBLIC void clck_init()
55641    {
55642            curr_time= 0;
55643            next_timeout= 0;
55644            timer_chain= 0;
55645    }

55647    PUBLIC void reset_time()
55648    {
55649            curr_time= 0;
55650    }

55652    PUBLIC void clck_timer(timer, timeout, func, fd)
55653    timer_t *timer;
55654    time_t timeout;
```

```
55655   timer_func_t func;
55656   int fd;
55657   {
55658           timer_t *timer_index;
55659
55660   #if DEBUG & 256
55661    { time_t curr_tim= get_time(); where();
55662           printf("clck_timer(0x%x, now%c%d HZ, 0x%x, %d)\n", timer,
55663           timeout >= curr_tim ? '+' : '-',
55664           timeout >= curr_tim ? timeout - curr_tim : curr_tim - timeout,
55665           func, fd); }
55666   #endif
55667           clck_fast_release(timer);
55668           timer->tim_next= 0;
55669           timer->tim_func= func;
55670           timer->tim_ref= fd;
55671           timer->tim_time= timeout;
55672
55673           if (!timer_chain)
55674                   timer_chain= timer;
55675           else if (timeout < timer_chain->tim_time)
55676           {
55677                   timer->tim_next= timer_chain;
55678                   timer_chain= timer;
55679           }
55680           else
55681           {
55682                   timer_index= timer_chain;
55683                   while (timer_index->tim_next &&
55684                           timer_index->tim_next->tim_time < timeout)
55685                           timer_index= timer_index->tim_next;
55686                   timer->tim_next= timer_index->tim_next;
55687                   timer_index->tim_next= timer;
55688           }
55689           if (timer_chain->tim_time != next_timeout)
55690                   set_timer();
55691   }

55693   PUBLIC void clck_tick (mess)
55694   message *mess;
55695   {
55696   #if DEBUG & 256
55697    { where(); printf("in clck_tick()\n"); }
55698   #endif
55699           next_timeout= 0;
55700           set_timer();
55701   }

55703   PRIVATE void clck_fast_release (timer)
55704   timer_t *timer;
55705   {
55706           timer_t *timer_index;
55707
55708           if (timer == timer_chain)
55709                   timer_chain= timer_chain->tim_next;
55710           else
55711           {
55712                   timer_index= timer_chain;
55713                   while (timer_index && timer_index->tim_next != timer)
55714                           timer_index= timer_index->tim_next;
```

```
55715                   if (timer_index)
55716                           timer_index->tim_next= timer->tim_next;
55717           }
55718   }

55720   PRIVATE void set_timer()
55721   {
55722           time_t new_time;
55723           time_t curr_time;
55724           timer_t *timer_index;
55725
55726   #if DEBUG & 256
55727    { where(); printf("in set_timer()\n"); }
55728   #endif
55729           curr_time= get_time();
55730
55731           while (timer_chain && timer_chain->tim_time<=curr_time)
55732           {
55733                   timer_index= timer_chain;
55734                   timer_chain= timer_chain->tim_next;
55735   #if DEBUG & 256
55736    { where(); printf("calling tim_func: 0x%x(%d, ..)\n",
55737           timer_index->tim_func, timer_index->tim_ref); }
55738   #endif
55739                   (*timer_index->tim_func)(timer_index->tim_ref, timer_index);
55740           }
55741           if (timer_chain)
55742                   new_time= timer_chain->tim_time;
55743           else
55744                   new_time= 0;
55745           if (new_time != next_timeout)
55746           {
55747                   static message mess;
55748
55749                   next_timeout= new_time;
55750                   assert (!new_time || new_time > curr_time);
55751
55752                   if (new_time)
55753                           new_time -= curr_time;
55754
55755                   mess.m_type= SET_SYNC_AL;
55756                   mess.CLOCK_PROC_NR= THIS_PROC;
55757                   mess.DELTA_TICKS= new_time;
55758                   if (sendrec (CLOCK, &mess) < 0)
55759                           ip_panic(("unable to sendrec"));
55760                   if (mess.m_type != OK)
55761                           ip_panic(("can't set timer"));
55762           }
55763   }

55765   void clck_untimer (timer)
55766   timer_t *timer;
55767   {
55768           clck_fast_release (timer);
55769           set_timer();
55770   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/inet.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

55800   /*      this file contains the interface of the network software with rest of
55801           minix. Furthermore it contains the main loop of the network task.
55802
55803   The valid messages and their parameters are:
55804
55805   from FS:
55806    ____________________________________________________________________
55807   |                |           |          |         |           |         |
55808   | m_type         |   DEVICE  | PROC_NR  | COUNT   | POSITION  | ADDRESS |
55809   |________________|___________|__________|_________|___________|_________|
55810   |                |           |          |         |           |         |
55811   | NW_OPEN        | minor dev | proc nr  | mode    |           |         |
55812   |________________|___________|__________|_________|___________|_________|
55813   |                |           |          |         |           |         |
55814   | NW_CLOSE       | minor dev | proc nr  |         |           |         |
55815   |________________|___________|__________|_________|___________|_________|
55816   |                |           |          |         |           |         |
55817   | NW_IOCTL       | minor dev | proc nr  |         | NWIO..    | address |
55818   |________________|___________|__________|_________|___________|_________|
55819   |                |           |          |         |           |         |
55820   | NW_READ        | minor dev | proc nr  | count   |           | address |
55821   |________________|___________|__________|_________|___________|_________|
55822   |                |           |          |         |           |         |
55823   | NW_WRITE       | minor dev | proc nr  | count   |           | address |
55824   |________________|___________|__________|_________|___________|_________|
55825   |                |           |          |         |           |         |
55826   | NW_CANCEL      | minor dev | proc nr  |         |           |         |
55827   |________________|___________|__________|_________|___________|_________|
55828
55829   from DL_ETH:
55830    ____________________________________________________________________
55831   |                |           |          |          |           |         |
55832   | m_type         |  DL_PORT  | DL_PROC  | DL_COUNT |  DL_STAT  | DL_TIME |
55833   |________________|___________|__________|__________|___________|_________|
55834   |                |           |          |          |           |         |
55835   | DL_TASK_INT    | minor dev | proc nr  | rd_count |  0  | stat |  time   |
55836   |________________|___________|__________|__________|___________|_________|
55837   |                |           |          |          |           |         |
55838   | DL_TASK_REPLY  | minor dev | proc nr  | rd_count | err | stat |  time   |         |
55839   |________________|___________|__________|__________|___________|_________|
55840   */
55841
55842   #include "inet.h"
55843   #include <unistd.h>
55844   #include "mq.h"
55845   #include "proto.h"
55846   #include "generic/assert.h"
55847   #include "generic/arp.h"
55848   #include "generic/buf.h"
55849   #include "generic/clock.h"
55850   #include "generic/eth.h"
55851   #include "generic/ip.h"
55852   #include "generic/sr.h"
55853   #include "generic/tcp.h"
55854   #include "generic/type.h"
```

```
55855   #include "generic/udp.h"
55856   #include "config.h"
55857
55858   INIT_PANIC();
55859
55860   _PROTOTYPE( void main, (void) );
55861
55862   FORWARD _PROTOTYPE( void nw_init, (void) );
55863
55864   PUBLIC void main()
55865   {
55866           mq_t *mq;
55867           int result;
55868
55869           nw_init();
55870           while (TRUE)
55871           {
55872                   mq= mq_get();
55873                   if (!mq)
55874                           ip_panic(("out of messages"));
55875
55876                   result= receive (ANY, &mq->mq_mess);
55877                   if (result<0)
55878                   {
55879                           ip_panic(("unable to receive: %d", result));
55880                   }
55881                   reset_time();
55882   #if DEBUG & 256
55883    { where(); printf("got message from %d, type %d\n",
55884           mq->mq_mess.m_source, mq->mq_mess.m_type); }
55885   #endif
55886                   switch (mq->mq_mess.m_source)
55887                   {
55888                   case FS_PROC_NR:
55889   #if DEBUG & 256
55890    { where(); printf("got message from fs, type %d\n", mq->mq_mess.m_type); }
55891   #endif
55892                           sr_rec(mq);
55893                           break;
55894                   case DL_ETH:
55895   #if DEBUG & 256
55896    { where(); printf("calling eth_rec\n"); }
55897   #endif
55898                           eth_rec(&mq->mq_mess);
55899                           mq_free(mq);
55900                           break;
55901                   case SYN_ALRM_TASK:
55902                           clck_tick (&mq->mq_mess);
55903                           mq_free(mq);
55904                           break;
55905                   default:
55906                           ip_panic(("message from unknown source: %d",
55907                                   mq->mq_mess.m_source));
55908                   }
55909           }
55910           ip_panic(("task is not allowed to terminate"));
55911   }

55913   PRIVATE void nw_init()
55914   {
```

```
55915   #if DEBUG & 256
55916    { where(); printf("starting mq_init()\n"); }
55917   #endif
55918           mq_init();
55919   #if DEBUG & 256
55920    { where(); printf("starting bf_init()\n"); }
55921   #endif
55922           bf_init();
55923   #if DEBUG & 256
55924    { where(); printf("starting clck_init()\n"); }
55925   #endif
55926           clck_init();
55927   #if DEBUG & 256
55928    { where(); printf("starting sr_init()\n"); }
55929   #endif
55930           sr_init();
55931   #if DEBUG & 256
55932    { where(); printf("starting eth_init()\n"); }
55933   #endif
55934           eth_init();
55935   #if DEBUG & 256
55936    { where(); printf("starting arp_init()\n"); }
55937   #endif
55938   #if ENABLE_ARP
55939           arp_init();
55940   #endif
55941   #if DEBUG & 256
55942    { where(); printf("starting ip_init()\n"); }
55943   #endif
55944   #if ENABLE_IP
55945           ip_init();
55946   #endif
55947   #if DEBUG & 256
55948    { where(); printf("starting tcp_init()\n"); }
55949   #endif
55950   #if ENABLE_TCP
55951           tcp_init();
55952   #endif
55953   #if DEBUG & 256
55954    { where(); printf("starting udp_init()\n"); }
55955   #endif
55956   #if ENABLE_UDP
55957           udp_init();
55958   #endif
55959   }

55961   void abort()
55962   {
55963           sys_abort(RBT_PANIC);
55964   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/mnx_eth.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

56000    /*
56001    inet/mnx_eth.c
56002
56003    Created:        Jan 2, 1992 by Philip Homburg
56004    */
56005
56006    #include "inet.h"
56007    #include "proto.h"
56008    #include "generic/assert.h"
56009    #include "generic/buf.h"
56010    #include "osdep_eth.h"
56011    #include "generic/clock.h"
56012    #include "generic/eth.h"
56013    #include "generic/eth_int.h"
56014    #include "generic/sr.h"
56015
56016    INIT_PANIC();
56017
56018    FORWARD _PROTOTYPE( void setup_read, (eth_port_t *eth_port) );
56019    FORWARD _PROTOTYPE( int do_sendrec, (int task, message *m1, message *m2) );
56020    FORWARD _PROTOTYPE( void read_int, (eth_port_t *eth_port, int count) );
56021    FORWARD _PROTOTYPE( void write_int, (eth_port_t *eth_port) );
56022
56023    PUBLIC void eth_init0()
56024    {
56025            int result;
56026            eth_port_t *eth_port;
56027            static message mess, repl_mess;
56028
56029            eth_port= &eth_port_table[0];
56030
56031            eth_port->etp_osdep.etp_port= 0;
56032            eth_port->etp_osdep.etp_task= DL_ETH;
56033            eth_port->etp_osdep.etp_minor= ETH_DEV;
56034
56035    #if XXX
56036            mess.m_type= DL_STOP;
56037            mess.DL_PORT= eth_port->etp_osdep.etp_port;
56038    #if DEBUG & 256
56039     { where(); printf("sending DL_STOP\n"); }
56040    #endif
56041    assert (eth_port->etp_osdep.etp_task != MM_PROC_NR);
56042            result= send(eth_port->etp_osdep.etp_task, &mess);
56043            if (result < 0)
56044            {
56045                    printf("send failed with error %d\n",result);
56046                    printf("eth_init0: unable to stop ethernet task\n");
56047                    return;
56048            }
56049    #endif
56050
56051    #if DEBUG & 256
56052     { where(); printf("sending DL_INIT\n"); }
56053    #endif
56054            mess.m_type= DL_INIT;
```

```
56055           mess.DL_PORT= eth_port->etp_osdep.etp_port;
56056           mess.DL_PROC= THIS_PROC;
56057           mess.DL_MODE= DL_NOMODE;
56058   assert (eth_port->etp_osdep.etp_task != MM_PROC_NR);
56059           result= send(eth_port->etp_osdep.etp_task, &mess);
56060           if (result<0)
56061           {
56062                   printf(
56063                   "eth_init0: unable to send to ethernet task, error= %d\n",
56064                           result);
56065                   return;
56066           }
56067
56068           if (receive(eth_port->etp_osdep.etp_task, &mess)<0)
56069                   ip_panic(("unable to receive"));
56070
56071           if (mess.m3_i1 != eth_port->etp_osdep.etp_port)
56072           {
56073                   printf("eth_init0: got reply for wrong port\n");
56074                   return;
56075           }
56076
56077           eth_port->etp_ethaddr= *(ether_addr_t *)mess.m3_ca1;
56078
56079           if (sr_add_minor (eth_port->etp_osdep.etp_minor,
56080                   eth_port- eth_port_table, eth_open, eth_close, eth_read,
56081                   eth_write, eth_ioctl, eth_cancel)<0)
56082                   ip_panic(("can't sr_init"));
56083
56084           eth_port->etp_flags |= EPF_ENABLED;
56085           eth_port->etp_wr_pack= 0;
56086           eth_port->etp_rd_pack= 0;
56087           setup_read (eth_port);
56088   }

56090   PUBLIC void eth_write_port(eth_port)
56091   eth_port_t *eth_port;
56092   {
56093           static message mess1, mess2;
56094           int i, pack_size, result;
56095           acc_t *pack, *pack_ptr;
56096           iovec_t *iovec;
56097
56098   #if DEBUG & 256
56099    { where(); bf_check_all_bufs(); }
56100   #endif
56101   #if DEBUG & 256
56102    { where(); printf("send_packet(&eth_port_table[%d], ..) called\n",
56103           eth_port-eth_port_table); }
56104   #endif
56105   assert (!(eth_port->etp_flags & EPF_WRITE_IP));
56106
56107           eth_port->etp_flags |= EPF_WRITE_IP;
56108
56109           pack= eth_port->etp_wr_pack;
56110           eth_port->etp_wr_pack= 0;
56111
56112           iovec= eth_port->etp_osdep.etp_wr_iovec;
56113           pack_size= 0;
56114   #if DEBUG & 256
```

```
56115     { where(); printf("bf_bufsize= %d\n", bf_bufsize(pack)); }
56116     #endif
56117             for (i=0, pack_ptr= pack; i<IOVEC_NR && pack_ptr; i++,
56118                     pack_ptr= pack_ptr->acc_next)
56119             {
56120                     iovec[i].iov_addr= (vir_bytes)ptr2acc_data(
56121                             pack_ptr);
56122                     pack_size += iovec[i].iov_size=
56123                             pack_ptr->acc_length;
56124             }
56125             if (i>= IOVEC_NR)
56126             {
56127     #if DEBUG
56128     { where(); printf("compacting fragment\n"); }
56129     #endif
56130                     pack= bf_pack(pack);            /* packet is too fragmented */
56131                     pack_size= 0;
56132                     for (i=0, pack_ptr= pack; i<IOVEC_NR &&
56133                             pack_ptr; i++, pack_ptr= pack_ptr->
56134                             acc_next)
56135                     {
56136                             iovec[i].iov_addr= (vir_bytes)
56137                                     ptr2acc_data(pack_ptr);
56138                             pack_size += iovec[i].iov_size=
56139                                     pack_ptr->acc_length;
56140                     }
56141             }
56142     #if DEBUG & 256
56143     { where(); printf("bf_bufsize= %d\n", bf_bufsize(pack));
56144             where(); printf("i= %d\n", i); }
56145     #endif
56146     assert (i< IOVEC_NR);
56147     assert (pack_size >= ETH_MIN_PACK_SIZE);
56148
56149             if (i==1)
56150             /* simple packets can be sent using DL_WRITE instead of DL_WRITEV */
56151             {
56152                     mess1.DL_COUNT= iovec[0].iov_size;
56153                     mess1.DL_ADDR= (char *)iovec[0].iov_addr;
56154                     mess1.m_type= DL_WRITE;
56155             }
56156             else
56157             {
56158                     mess1.DL_COUNT= i;
56159                     mess1.DL_ADDR= (char *)iovec;
56160                     mess1.m_type= DL_WRITEV;
56161             }
56162             mess1.DL_PORT= eth_port->etp_osdep.etp_port;
56163             mess1.DL_PROC= THIS_PROC;
56164             mess1.DL_MODE= DL_NOMODE;
56165
56166     #if DEBUG & 256
56167     { where(); printf("calling do_sendrec\n"); }
56168     #endif
56169     assert (eth_port->etp_osdep.etp_task != MM_PROC_NR);
56170             result= do_sendrec (eth_port->etp_osdep.etp_task, &mess1, &mess2);
56171     #if DEBUG & 256
56172     { where(); printf("do_sendrec done\n"); }
56173     #endif
56174
```

```
56175   #if DEBUG & 256
56176    { where(); printf("got reply from DLL\n"); }
56177   #endif
56178   #if DEBUG
56179    if (mess1.m_type != DL_TASK_REPLY)
56180    { where(); printf("wrong m_type (=%d)\n", mess1.m_type); }
56181    if (mess1.DL_PORT != eth_port->etp_osdep.etp_port)
56182    { where(); printf("wrong DL_PORT (=%d)\n", mess1.DL_PORT); }
56183    if (mess1.DL_PROC != THIS_PROC)
56184    { where(); printf("wrong DL_PROC (=%d)\n", mess1.DL_PROC); }
56185   #endif
56186
56187   assert (mess1.m_type == DL_TASK_REPLY && mess1.DL_PORT == mess1.DL_PORT &&
56188           mess1.DL_PROC == THIS_PROC);
56189
56190   assert((mess1.DL_STAT >> 16) == OK);
56191
56192           if (!(mess1.DL_STAT & DL_PACK_SEND))
56193           /* packet is not sent, suspend */
56194           {
56195   #if DEBUG & 256
56196    { where(); printf("setting EPF_WRITE_SP\n"); }
56197   #endif
56198                   eth_port->etp_flags |= EPF_WRITE_SP;
56199                   eth_port->etp_wr_pack= pack;
56200           }
56201           else
56202           /* packet is sent */
56203           {
56204                   eth_port->etp_flags &= ~EPF_WRITE_IP;
56205                   eth_arrive(eth_port, pack);
56206   #if DEBUG & 256
56207    { where(); printf("write done\n"); }
56208   #endif
56209           }
56210
56211           if (result == 1)        /* got an INT_TASK */
56212           {
56213   assert(mess2.DL_STAT == DL_PACK_RECV);
56214   assert(!(mess1.DL_STAT & DL_PACK_RECV));
56215   compare(mess2.DL_PORT, ==, eth_port->etp_osdep.etp_port);
56216   compare(mess2.DL_PROC, ==, THIS_PROC);
56217                   read_int(eth_port, mess2.DL_COUNT);
56218           }
56219           else if (mess1.DL_STAT & DL_PACK_RECV)
56220           {
56221                   read_int(eth_port, mess1.DL_COUNT);
56222           }
56223   }

56225   PUBLIC void eth_rec(m)
56226   message *m;
56227   {
56228           int i;
56229           eth_port_t *loc_port;
56230           int stat;
56231
56232   assert (m->m_source == DL_ETH);
56233
56234           set_time (m->DL_CLCK);
```

```
56235
56236           for (i=0, loc_port= eth_port_table; i<ETH_PORT_NR; i++, loc_port++)
56237           {
56238                   if (loc_port->etp_osdep.etp_port == m->DL_PORT &&
56239                           loc_port->etp_osdep.etp_task == m->m_source)
56240                           break;
56241           }
56242
56243   assert (i<ETH_PORT_NR);
56244
56245           stat= m->DL_STAT & 0xffff;
56246
56247   assert(stat & (DL_PACK_SEND|DL_PACK_RECV));
56248           if (stat & DL_PACK_SEND)
56249           {
56250                   write_int(loc_port);
56251           }
56252           if (stat & DL_PACK_RECV)
56253           {
56254                   read_int(loc_port, m->DL_COUNT);
56255           }
56256   }

56258   PUBLIC int eth_get_stat(eth_port, eth_stat)
56259   eth_port_t *eth_port;
56260   eth_stat_t *eth_stat;
56261   {
56262           acc_t *acc;
56263           int result;
56264           static message mess, mlocked;
56265
56266   #if DEBUG
56267    { where(); printf("eth_get_stat called\n"); }
56268   #endif
56269           mess.m_type= DL_GETSTAT;
56270           mess.DL_PORT= eth_port->etp_osdep.etp_port;
56271           mess.DL_PROC= THIS_PROC;
56272           mess.DL_ADDR= (char *)eth_stat;
56273
56274           for (;;)
56275           {
56276   assert (eth_port->etp_osdep.etp_task != MM_PROC_NR);
56277                   result= send(eth_port->etp_osdep.etp_task, &mess);
56278                   if (result != ELOCKED)
56279                           break;
56280                   result= receive(eth_port->etp_osdep.etp_task, &mlocked);
56281   assert(result == OK);
56282   #if DEBUG
56283    { where(); printf("calling eth_rec()\n"); }
56284   #endif
56285                   eth_rec(&mlocked);
56286           }
56287   assert(result == OK);
56288
56289           result= receive(eth_port->etp_osdep.etp_task, &mess);
56290   assert(result == OK);
56291   assert(mess.m_type == DL_TASK_REPLY);
56292
56293           result= mess.DL_STAT >> 16;
56294   assert (result == 0);
```

```
56295
56296           if (mess.DL_STAT)
56297           {
56298   #if DEBUG
56299    { where(); printf("calling eth_rec()\n"); }
56300   #endif
56301                   eth_rec(&mess);
56302           }
56303           return OK;
56304   }

56306   PUBLIC void eth_set_rec_conf (eth_port, flags)
56307   eth_port_t *eth_port;
56308   u32_t flags;
56309   {
56310           int result;
56311           unsigned dl_flags;
56312           static message mess, repl_mess;
56313
56314   #if DEBUG
56315    { where(); printf("eth_chk_rec_conf(&eth_port_table[%d])\n",
56316           eth_port-eth_port_table); }
56317   #endif
56318           dl_flags= DL_NOMODE;
56319           if (flags & NWEO_EN_BROAD)
56320                   dl_flags |= DL_BROAD_REQ;
56321           if (flags & NWEO_EN_MULTI)
56322                   dl_flags |= DL_MULTI_REQ;
56323           if (flags & NWEO_EN_PROMISC)
56324                   dl_flags |= DL_PROMISC_REQ;
56325
56326           mess.m_type= DL_INIT;
56327           mess.DL_PORT= eth_port->etp_osdep.etp_port;
56328           mess.DL_PROC= THIS_PROC;
56329           mess.DL_MODE= dl_flags;
56330
56331           do
56332           {
56333   assert (eth_port->etp_osdep.etp_task != MM_PROC_NR);
56334                   result= send (eth_port->etp_osdep.etp_task, &mess);
56335                   if (result == ELOCKED)  /* etp_task is sending to this task,
56336                                              I hope */
56337                   {
56338                           if (receive (eth_port->etp_osdep.etp_task,
56339                                   &repl_mess)< 0)
56340                                   ip_panic(("unable to receive"));
56341   #if DEBUG
56342    { where(); printf("calling eth_rec\n"); }
56343   #endif
56344                           eth_rec(&repl_mess);
56345                   }
56346           } while (result == ELOCKED);
56347
56348           if (result < 0)
56349                   ip_panic(("unable to send(%d)", result));
56350
56351           if (receive (eth_port->etp_osdep.etp_task, &repl_mess) < 0)
56352                   ip_panic(("unable to receive"));
56353
56354   assert (repl_mess.m_type == DL_INIT_REPLY);
```

```
56355           if (repl_mess.m3_i1 != eth_port->etp_osdep.etp_port)
56356           {
56357                   ip_panic(("got reply for wrong port"));
56358                   return;
56359           }
56360   }

56362   PRIVATE int do_sendrec (tofrom, mptr1, mptr2)
56363   int tofrom;
56364   message *mptr1;
56365   message *mptr2;
56366   {
56367           int result;
56368           int extra;
56369
56370   assert (tofrom != MM_PROC_NR);
56371           result= send (tofrom, mptr1);
56372           if (result == ELOCKED)
56373           {
56374                   /* ethernet task is sending to this task, I hope */
56375                   result= receive(tofrom, mptr2);
56376
56377                   if (result < 0)
56378                           ip_panic(("unable to receive"));
56379                   extra= 1;
56380   assert (tofrom != MM_PROC_NR);
56381                   result= send (tofrom, mptr1);
56382           }
56383           else
56384                   extra= 0;
56385
56386           if (result < 0)
56387                   ip_panic(("unable to send"));
56388
56389           result= receive (tofrom, mptr1);
56390           if (result < 0)
56391                   ip_panic(("unable to receive"));
56392
56393   assert (mptr1->m_type == DL_TASK_REPLY);
56394           return extra;
56395   }

56397   PRIVATE void write_int(eth_port)
56398   eth_port_t *eth_port;
56399   {
56400           acc_t *pack;
56401
56402   #if DEBUG & 256
56403    { where(); printf("write_int called\n"); }
56404   #endif
56405
56406   assert(eth_port->etp_flags & (EPF_WRITE_IP|EPF_WRITE_SP) ==
56407           (EPF_WRITE_IP|EPF_WRITE_SP));
56408
56409           pack= eth_port->etp_wr_pack;
56410           eth_port->etp_wr_pack= NULL;
56411           eth_arrive(eth_port, pack);
56412           eth_port->etp_flags &= ~(EPF_WRITE_IP|EPF_WRITE_SP);
56413           while (eth_get_work(eth_port))
56414                   ;
```

```
56415   }

56417   PRIVATE void read_int(eth_port, count)
56418   eth_port_t *eth_port;
56419   int count;
56420   {
56421           acc_t *pack, *cut_pack;
56422
56423           pack= eth_port->etp_rd_pack;
56424           eth_port->etp_rd_pack= NULL;
56425
56426           cut_pack= bf_cut(pack, 0, count);
56427           bf_afree(pack);
56428
56429           eth_arrive(eth_port, cut_pack);
56430
56431           if (!(eth_port->etp_flags & EPF_READ_SP))
56432           {
56433                   eth_port->etp_flags &= ~EPF_READ_IP;
56434                   return;
56435           }
56436           eth_port->etp_flags &= ~(EPF_READ_IP|EPF_READ_SP);
56437           setup_read(eth_port);
56438   }

56440   PRIVATE void setup_read(eth_port)
56441   eth_port_t *eth_port;
56442   {
56443           acc_t *pack, *pack_ptr;
56444           static message mess1, mess2;
56445           iovec_t *iovec;
56446           int i, result;
56447
56448   assert(!(eth_port->etp_flags & (EPF_READ_IP|EPF_READ_SP)));
56449
56450           do
56451           {
56452
56453   assert (!eth_port->etp_rd_pack);
56454
56455                   iovec= eth_port->etp_osdep.etp_rd_iovec;
56456                   pack= bf_memreq (ETH_MAX_PACK_SIZE);
56457
56458                   for (i=0, pack_ptr= pack; i<RD_IOVEC && pack_ptr;
56459                           i++, pack_ptr= pack_ptr->acc_next)
56460                   {
56461                           iovec[i].iov_addr= (vir_bytes)ptr2acc_data(pack_ptr);
56462                           iovec[i].iov_size= (vir_bytes)pack_ptr->acc_length;
56463   #if DEBUG & 256
56464    { where(); printf("filling iovec[%d] with iov_addr= %x, iov_size= %x\n",
56465           i, iovec[i].iov_addr, iovec[i].iov_size); }
56466   #endif
56467                   }
56468
56469   assert (!pack_ptr);
56470
56471                   mess1.m_type= DL_READV;
56472                   mess1.DL_PORT= eth_port->etp_osdep.etp_port;
56473                   mess1.DL_PROC= THIS_PROC;
56474                   mess1.DL_COUNT= i;
```

```
56475                   mess1.DL_ADDR= (char *)iovec;
56476
56477                   result= do_sendrec (eth_port->etp_osdep.etp_task, &mess1,
56478                           &mess2);
56479
56480   #if DEBUG
56481    if (mess1.m_type != DL_TASK_REPLY)
56482    { where(); printf("wrong m_type (=%d)\n", mess1.m_type); }
56483    if (mess1.DL_PORT != mess1.DL_PORT)
56484    { where(); printf("wrong DL_PORT (=%d)\n", mess1.DL_PORT); }
56485    if (mess1.DL_PROC != THIS_PROC)
56486    { where(); printf("wrong DL_PROC (=%d)\n", mess1.DL_PROC); }
56487   #endif
56488
56489   assert (mess1.m_type == DL_TASK_REPLY && mess1.DL_PORT == mess1.DL_PORT &&
56490           mess1.DL_PROC == THIS_PROC);
56491   compare((mess1.DL_STAT >> 16), ==, OK);
56492
56493                   if (mess1.DL_STAT & DL_PACK_RECV)
56494                   /* packet received */
56495                   {
56496                           pack_ptr= bf_cut(pack, 0, mess1.DL_COUNT);
56497                           bf_afree(pack);
56498
56499   assert(!(eth_port->etp_flags & EPF_READ_IP));
56500                           eth_arrive(eth_port, pack_ptr);
56501   assert(!(eth_port->etp_flags & EPF_READ_IP));
56502                   }
56503                   else
56504                   /* no packet received */
56505                   {
56506                           eth_port->etp_rd_pack= pack;
56507                           eth_port->etp_flags |= EPF_READ_IP;
56508                   }
56509
56510                   if (result == 1)        /* got an INT_TASK */
56511                   {
56512   assert(mess2.DL_STAT == DL_PACK_SEND);
56513   assert(!(mess1.DL_STAT & DL_PACK_SEND));
56514   assert (mess2.DL_PORT == mess2.DL_PORT &&
56515           mess2.DL_PROC == THIS_PROC);
56516                           write_int(eth_port);
56517                   }
56518                   else if (mess1.DL_STAT & DL_PACK_SEND)
56519                   {
56520                           write_int(eth_port);
56521                   }
56522           } while (!(eth_port->etp_flags & EPF_READ_IP));
56523           eth_port->etp_flags |= EPF_READ_SP;
56524   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/mq.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

56600   /*
56601   inet/mq.c
56602
56603   Created:        Jan 3, 1992 by Philip Homburg
56604   */
56605
56606   #include "inet.h"
56607   #include "mq.h"
56608
56609   #define MQ_SIZE         64
56610
56611   PRIVATE mq_t mq_list[MQ_SIZE];
56612   PRIVATE mq_t *mq_freelist;
56613
56614   void mq_init()
56615   {
56616           int i;
56617
56618           mq_freelist= NULL;
56619           for (i= 0; i<MQ_SIZE; i++)
56620           {
56621                   mq_list[i].mq_next= mq_freelist;
56622                   mq_freelist= &mq_list[i];
56623           }
56624   }

56626   mq_t *mq_get()
56627   {
56628           mq_t *mq;
56629
56630           mq= mq_freelist;
56631           if (mq)
56632           {
56633                   mq_freelist= mq->mq_next;
56634                   mq->mq_next= NULL;
56635           }
56636           return mq;
56637   }

56639   void mq_free(mq)
56640   mq_t *mq;
56641   {
56642           mq->mq_next= mq_freelist;
56643           mq_freelist= mq;
56644   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/putk.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

56700    /* INET must occasionally print some message.  It uses the standard library
56701     * routine prink().  (The name "printf" is really a macro defined as "printk").
56702     * Printing is done by calling the TTY task directly, not going through FS.
56703     */
56704
56705    #include "inet.h"
56706
56707    #define BUF_SIZE          100   /* print buffer size */
56708
56709    PRIVATE int buf_count;          /* # characters in the buffer */
56710    PRIVATE char print_buf[BUF_SIZE];       /* output is buffered here */
56711    PRIVATE message putch_msg;      /* used for message to TTY task */
56712
56713    FORWARD _PROTOTYPE (void flush, (void) );
56714    _PROTOTYPE (void putk, (int c) );
56715
56716
56717    /*============================================================================*
56718     *                                putk                                        *
56719     *============================================================================*/
56720    PUBLIC void putk(c)
56721    int c;
56722    {
56723    /* Accumulate another character.  If 0 or buffer full, print it. */
56724
56725      if (c == 0 || buf_count == BUF_SIZE) flush();
56726      if (c == '\n') putk('\r');
56727      if (c != 0) print_buf[buf_count++] = c;
56728    }


56731    /*============================================================================*
56732     *                                flush                                       *
56733     *============================================================================*/
56734    PRIVATE void flush()
56735    {
56736    /* Flush the print buffer by calling TTY task. */
56737
56738      if (buf_count == 0) return;
56739      putch_msg.m_type = DEV_WRITE;
56740      putch_msg.PROC_NR  = THIS_PROC;
56741      putch_msg.TTY_LINE = 0;
56742      putch_msg.ADDRESS  = print_buf;
56743      putch_msg.COUNT = buf_count;
56744      sendrec(TTY, &putch_msg);
56745      if (putch_msg.REP_STATUS == SUSPEND)
56746            receive(TTY, &putch_msg);
56747      buf_count = 0;
56748    }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/sr.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

56800   /*      this file contains the interface of the network software with the file
56801           system.
56802
56803   The valid messages and their parameters are:
56804
56805    _____________________________________________________________________
56806   |                  |             |           |         |             |           |
56807   | m_type           |   DEVICE    | PROC_NR   | COUNT   | REQUEST     | ADDRESS   |
56808   |__________________|_____________|___________|_________|_____________|___________|
56809   |                  |             |           |         |             |           |
56810   | NW_OPEN          | minor dev   | proc nr   | mode    |             |           |
56811   |__________________|_____________|___________|_________|_____________|___________|
56812   |                  |             |           |         |             |           |
56813   | NW_CLOSE         | minor dev   | proc nr   |         |             |           |
56814   |__________________|_____________|___________|_________|_____________|___________|
56815   |                  |             |           |         |             |           |
56816   | NW_IOCTL         | minor dev   | proc nr   |         | NWIO..      | address   |
56817   |__________________|_____________|___________|_________|_____________|___________|
56818   |                  |             |           |         |             |           |
56819   | NW_READ          | minor dev   | proc nr   | count   |             | address   |
56820   |__________________|_____________|___________|_________|_____________|___________|
56821   |                  |             |           |         |             |           |
56822   | NW_WRITE         | minor dev   | proc nr   | count   |             | address   |
56823   |__________________|_____________|___________|_________|_____________|___________|
56824   |                  |             |           |         |             |           |
56825   | NW_CANCEL        | minor dev   | proc nr   |         |             |           |
56826   |__________________|_____________|___________|_________|_____________|___________|
56827
56828   */
56829
56830   #include "inet.h"
56831   #include <minix/callnr.h>
56832   #include "mq.h"
56833   #include "proto.h"
56834   #include "generic/assert.h"
56835   #include "generic/buf.h"
56836   #include "generic/sr.h"
56837   #include "generic/type.h"
56838
56839   INIT_PANIC();
56840
56841   #define FD_NR                   32
56842
56843   typedef struct sr_fd
56844   {
56845           int srf_flags;
56846           int srf_fd;
56847           int srf_port;
56848           sr_open_t srf_open;
56849           sr_close_t srf_close;
56850           sr_write_t srf_write;
56851           sr_read_t srf_read;
56852           sr_ioctl_t srf_ioctl;
56853           sr_cancel_t srf_cancel;
56854           mq_t *srf_ioctl_q, *srf_ioctl_q_tail;
```

```
56855           mq_t *srf_read_q, *srf_read_q_tail;
56856           mq_t *srf_write_q, *srf_write_q_tail;
56857   } sr_fd_t;
56858
56859   #define SFF_FLAGS       0x0F
56860   #       define SFF_FREE         0x00
56861   #       define SFF_MINOR        0x01
56862   #       define SFF_INUSE        0x02
56863   #       define SFF_BUSY         0x3C
56864   #               define SFF_IOCTL_IP     0x04
56865   #               define SFF_READ_IP      0x08
56866   #               define SFF_WRITE_IP     0x10
56867   #       define SFF_PENDING_REQ  0x30
56868   #       define SFF_SUSPENDED    0x1C0
56869   #               define SFF_IOCTL_SUSP   0x40
56870   #               define SFF_READ_SUSP    0x80
56871   #               define SFF_WRITE_SUSP   0x100
56872
56873   FORWARD _PROTOTYPE ( int sr_open, (message *m) );
56874   FORWARD _PROTOTYPE ( void sr_close, (message *m) );
56875   FORWARD _PROTOTYPE ( int sr_rwio, (mq_t *m) );
56876   FORWARD _PROTOTYPE ( int sr_cancel, (message *m) );
56877   FORWARD _PROTOTYPE ( void sr_reply, (message *mes_ptr, int reply) );
56878   FORWARD _PROTOTYPE ( void sr_revive, (mq_t *mes_ptr, int reply) );
56879   FORWARD _PROTOTYPE ( sr_fd_t *sr_getchannel, (int minor));
56880   FORWARD _PROTOTYPE ( acc_t *sr_get_userdata, (int fd, vir_bytes offset,
56881                                           vir_bytes count, int for_ioctl) );
56882   FORWARD _PROTOTYPE ( int sr_put_userdata, (int fd, vir_bytes offset,
56883                                                 acc_t *data, int for_ioctl) );
56884   FORWARD _PROTOTYPE ( int sr_repl_queue, (int proc) );
56885   FORWARD _PROTOTYPE ( int walk_queue, (sr_fd_t *sr_fd, mq_t *q_head,
56886                                   mq_t **q_tail_ptr, int type, int proc_n) );
56887   FORWARD _PROTOTYPE ( void process_req_q, (mq_t *mq, mq_t *tail,
56888                                                       mq_t **tail_ptr) );
56889   FORWARD _PROTOTYPE ( int cp_u2b, (int proc, char *src, acc_t **var_acc_ptr,
56890                                                             int size) );
56891   FORWARD _PROTOTYPE ( int cp_b2u, (acc_t *acc_ptr, int proc, char *dest) );
56892
56893   PRIVATE sr_fd_t sr_fd_table[FD_NR];
56894   PRIVATE mq_t *repl_queue, *repl_queue_tail;
56895   PRIVATE cpvec_t cpvec[CPVEC_NR];
56896
56897   PUBLIC void sr_init()
56898   {
56899           int i;
56900
56901           for (i=0; i<FD_NR; i++)
56902                   sr_fd_table[i].srf_flags= SFF_FREE;
56903           repl_queue= NULL;
56904   }

56906   PUBLIC void sr_rec(m)
56907   mq_t *m;
56908   {
56909           message mess, *mp;
56910           int result;
56911           int send_reply, free_mess;
56912
56913           if (repl_queue)
56914           {
```

```
56915                   if (m->mq_mess.m_type == NW_CANCEL)
56916                   {
56917                           result= sr_repl_queue(m->mq_mess.PROC_NR);
56918                           if (result)
56919                                   return; /* canceled request in queue */
56920                   }
56921                   else
56922                           sr_repl_queue(ANY);
56923           }
56924
56925   #if DEBUG & 256
56926    { where(); printf("sr_rec: message from %d for %d type %d, minor %d\n",
56927           m->mq_mess.m_source, m->mq_mess.PROC_NR, m->mq_mess.m_type,
56928           m->mq_mess.DEVICE); }
56929   #endif
56930           switch (m->mq_mess.m_type)
56931           {
56932           case NW_OPEN:
56933                   result= sr_open(&m->mq_mess);
56934                   send_reply= 1;
56935                   free_mess= 1;
56936                   break;
56937           case NW_CLOSE:
56938                   sr_close(&m->mq_mess);
56939                   result= OK;
56940                   send_reply= 1;
56941                   free_mess= 1;
56942                   break;
56943           case NW_READ:
56944           case NW_WRITE:
56945           case NW_IOCTL:
56946   #if DEBUG & 256
56947    { where(); printf("calling rwio\n"); }
56948   #endif
56949                   result= sr_rwio(m);
56950   #if DEBUG & 256
56951    { where(); printf("rwio()= %d\n", result); }
56952   #endif
56953   assert(result == OK || result == SUSPEND);
56954                   send_reply= (result == SUSPEND);
56955                   free_mess= 0;
56956                   break;
56957           case NW_CANCEL:
56958                   result= sr_cancel(&m->mq_mess);
56959   assert(result == OK || result == EINTR);
56960                   send_reply= (result == EINTR);
56961                   free_mess= 1;
56962                   break;
56963           default:
56964                   ip_panic(("unknown message, type= %d", m->mq_mess.m_type));
56965           }
56966           if (send_reply)
56967           {
56968                   if (free_mess)
56969                           mp= &m->mq_mess;
56970                   else
56971                   {
56972                           mess= m->mq_mess;
56973                           mp= &mess;
56974                   }
```

```
56975                   sr_reply(mp, result);
56976           }
56977           if (free_mess)
56978                   mq_free(m);
56979   }

56981   PUBLIC int sr_add_minor(minor, port, openf, closef, readf, writef,
56982           ioctlf, cancelf)
56983   int minor;
56984   int port;
56985   sr_open_t openf;
56986   sr_close_t closef;
56987   sr_read_t readf;
56988   sr_write_t writef;
56989   sr_ioctl_t ioctlf;
56990   sr_cancel_t cancelf;
56991   {
56992           sr_fd_t *sr_fd;
56993
56994           assert (minor>=0 && minor<FD_NR);
56995
56996           sr_fd= &sr_fd_table[minor];
56997
56998           if (sr_fd->srf_flags & SFF_INUSE)
56999                   return EGENERIC;
57000
57001           sr_fd->srf_flags= SFF_INUSE | SFF_MINOR;
57002           sr_fd->srf_port= port;
57003           sr_fd->srf_open= openf;
57004           sr_fd->srf_close= closef;
57005           sr_fd->srf_write= writef;
57006           sr_fd->srf_read= readf;
57007           sr_fd->srf_ioctl= ioctlf;
57008           sr_fd->srf_cancel= cancelf;
57009
57010           return OK;
57011   }

57013   PRIVATE int sr_open(m)
57014   message *m;
57015   {
57016           sr_fd_t *sr_fd;
57017
57018           int minor= m->DEVICE;
57019           int i, fd;
57020
57021           if (minor<0 || minor>FD_NR)
57022           {
57023   #if DEBUG
57024    { where(); printf("replying EINVAL\n"); }
57025   #endif
57026                   return EINVAL;
57027           }
57028           if (!(sr_fd_table[minor].srf_flags & SFF_MINOR))
57029           {
57030   #if DEBUG
57031    { where(); printf("replying ENXIO\n"); }
57032   #endif
57033                   return ENXIO;
57034           }
```

```
57035           for (i=0; i<FD_NR && (sr_fd_table[i].srf_flags & SFF_INUSE); i++);
57036
57037           if (i>=FD_NR)
57038           {
57039   #if DEBUG
57040    { where(); printf("replying ENFILE\n"); }
57041   #endif
57042                   return ENFILE;
57043           }
57044
57045           sr_fd= &sr_fd_table[i];
57046           *sr_fd= sr_fd_table[minor];
57047           sr_fd->srf_flags= SFF_INUSE;
57048           fd= (*sr_fd->srf_open)(sr_fd->srf_port, i, sr_get_userdata,
57049                   sr_put_userdata);
57050   #if DEBUG & 256
57051    { where(); printf("srf_open: 0x%x(%d, %d, .., ..)= %d\n", sr_fd->srf_open,
57052           sr_fd->srf_port, i, fd); }
57053   #endif
57054           if (fd<0)
57055           {
57056                   sr_fd->srf_flags= SFF_FREE;
57057   #if DEBUG
57058    { where(); printf("replying %d\n", fd); }
57059   #endif
57060                   return fd;
57061           }
57062           sr_fd->srf_fd= fd;
57063   #if DEBUG & 256
57064    { where(); printf("replying %d\n", i); }
57065   #endif
57066           return i;
57067   }

57069   PRIVATE void sr_close(m)
57070   message *m;
57071   {
57072           sr_fd_t *sr_fd;
57073
57074           sr_fd= sr_getchannel(m->DEVICE);
57075           assert (sr_fd);
57076
57077           if (sr_fd->srf_flags & SFF_BUSY)
57078                   ip_panic(("close on busy channel"));
57079
57080           assert (!(sr_fd->srf_flags & SFF_MINOR));
57081           (*sr_fd->srf_close)(sr_fd->srf_fd);
57082   #if DEBUG & 256
57083    { where(); printf("srf_close: 0x%x(%d)\n", sr_fd->srf_close, sr_fd->srf_fd); }
57084   #endif
57085           sr_fd->srf_flags= SFF_FREE;
57086   }

57088   PRIVATE int sr_rwio(m)
57089   mq_t *m;
57090   {
57091           sr_fd_t *sr_fd;
57092           mq_t **q_head_ptr, **q_tail_ptr;
57093           int ip_flag, susp_flag;
57094           int result;
```

```
57095           unsigned long request;
57096           size_t size;
57097
57098           sr_fd= sr_getchannel(m->mq_mess.DEVICE);
57099   assert (sr_fd);
57100
57101           switch(m->mq_mess.m_type)
57102           {
57103           case NW_READ:
57104                   q_head_ptr= &sr_fd->srf_read_q;
57105                   q_tail_ptr= &sr_fd->srf_read_q_tail;
57106                   ip_flag= SFF_READ_IP;
57107                   susp_flag= SFF_READ_SUSP;
57108                   break;
57109           case NW_WRITE:
57110                   q_head_ptr= &sr_fd->srf_write_q;
57111                   q_tail_ptr= &sr_fd->srf_write_q_tail;
57112                   ip_flag= SFF_WRITE_IP;
57113                   susp_flag= SFF_WRITE_SUSP;
57114                   break;
57115           case NW_IOCTL:
57116                   q_head_ptr= &sr_fd->srf_ioctl_q;
57117                   q_tail_ptr= &sr_fd->srf_ioctl_q_tail;
57118                   ip_flag= SFF_IOCTL_IP;
57119                   susp_flag= SFF_IOCTL_SUSP;
57120                   break;
57121           default:
57122                   ip_panic(("illegal case entry"));
57123           }
57124
57125           if (sr_fd->srf_flags & ip_flag)
57126           {
57127   assert(sr_fd->srf_flags & susp_flag);
57128   assert(*q_head_ptr);
57129                   (*q_tail_ptr)->mq_next= m;
57130                   *q_tail_ptr= m;
57131                   return SUSPEND;
57132           }
57133   assert(!*q_head_ptr);
57134
57135           *q_tail_ptr= *q_head_ptr= m;
57136           sr_fd->srf_flags |= ip_flag;
57137
57138           switch(m->mq_mess.m_type)
57139           {
57140           case NW_READ:
57141   #if DEBUG&256
57142    { where(); printf("calling 0x%x(%d, %d)\n", sr_fd->srf_read, sr_fd->srf_fd,
57143           m->mq_mess.COUNT); }
57144   #endif
57145                   result= (*sr_fd->srf_read)(sr_fd->srf_fd, m->mq_mess.COUNT);
57146                   break;
57147           case NW_WRITE:
57148   #if DEBUG&256
57149    { where(); printf("calling 0x%x(%d, %d)\n", sr_fd->srf_write, sr_fd->srf_fd,
57150           m->mq_mess.COUNT); }
57151   #endif
57152                   result= (*sr_fd->srf_write)(sr_fd->srf_fd, m->mq_mess.COUNT);
57153                   break;
57154           case NW_IOCTL:
```

```
57155                   request= m->mq_mess.REQUEST;
57156   #ifdef IOCPARM_MASK
57157                   size= (request >> 16) & IOCPARM_MASK;
57158                   if (size>MAX_IOCTL_S)
57159                   {
57160   #if DEBUG
57161    { where(); printf("replying EINVAL\n"); }
57162   #endif
57163                           result= sr_put_userdata(sr_fd-sr_fd_table, EINVAL,
57164                                                                   NULL, 1);
57165   assert(result == OK);
57166                           return OK;
57167                   }
57168   #endif /* IOCPARM_MASK */
57169   #if DEBUG
57170    { where(); printf("calling 0x%x(%d, 0x%lx)\n", sr_fd->srf_ioctl, sr_fd->srf_fd,
57171           request); }
57172   #endif
57173                   result=(*sr_fd->srf_ioctl)(sr_fd->srf_fd, request);
57174                   break;
57175           default:
57176                   ip_panic(("illegal case entry"));
57177           }
57178
57179   #if DEBUG
57180    if (result != OK && result != SUSPEND)
57181    { where(); printf("result= %d\n", result); }
57182   #endif
57183   assert(result == OK || result == SUSPEND);
57184           if (result == SUSPEND)
57185                   sr_fd->srf_flags |= susp_flag;
57186           return result;
57187   }

57189   PRIVATE int sr_cancel(m)
57190   message *m;
57191   {
57192           sr_fd_t *sr_fd;
57193           int i, result;
57194           mq_t *q_ptr, *q_ptr_prv;
57195           int proc_nr;
57196
57197           result=EINTR;
57198           proc_nr=  m->PROC_NR;
57199           sr_fd= sr_getchannel(m->DEVICE);
57200   assert (sr_fd);
57201
57202           result= walk_queue(sr_fd, sr_fd->srf_ioctl_q, &sr_fd->srf_ioctl_q_tail,
57203                   SR_CANCEL_IOCTL, proc_nr);
57204           if (result != EAGAIN)
57205                   return result;
57206           result= walk_queue(sr_fd, sr_fd->srf_read_q, &sr_fd->srf_read_q_tail,
57207                   SR_CANCEL_READ, proc_nr);
57208           if (result != EAGAIN)
57209                   return result;
57210           result= walk_queue(sr_fd, sr_fd->srf_write_q, &sr_fd->srf_write_q_tail,
57211                   SR_CANCEL_WRITE, proc_nr);
57212           if (result != EAGAIN)
57213                   return result;
57214           ip_panic(("request not found"));
```

```
57215   }

57217   PRIVATE int walk_queue(sr_fd, q_head, q_tail_ptr, type, proc_nr)
57218   sr_fd_t *sr_fd;
57219   mq_t *q_head, **q_tail_ptr;
57220   int type;
57221   int proc_nr;
57222   {
57223           mq_t *q_ptr_prv, *q_ptr;
57224           int result;
57225
57226           for(q_ptr_prv= NULL, q_ptr= q_head; q_ptr;
57227                   q_ptr_prv= q_ptr, q_ptr= q_ptr->mq_next)
57228           {
57229                   if (q_ptr->mq_mess.PROC_NR != proc_nr)
57230                           continue;
57231                   if (!q_ptr_prv)
57232                   {
57233   #if DEBUG & 256
57234    { where(); printf("calling 0x%x(%d, %d)\n", sr_fd->srf_cancel, sr_fd->srf_fd,
57235           type); }
57236   #endif
57237
57238                           result= (*sr_fd->srf_cancel)(sr_fd->srf_fd, type);
57239   assert(result == OK);
57240                           return OK;
57241                   }
57242                   q_ptr_prv->mq_next= q_ptr->mq_next;
57243                   mq_free(q_ptr);
57244                   if (!q_ptr_prv->mq_next)
57245                           *q_tail_ptr= q_ptr_prv;
57246                   return EINTR;
57247           }
57248           return EAGAIN;
57249   }

57251   PRIVATE sr_fd_t *sr_getchannel(minor)
57252   int minor;
57253   {
57254           sr_fd_t *loc_fd;
57255
57256   compare(minor, >=, 0);
57257   compare(minor, <, FD_NR);
57258
57259           loc_fd= &sr_fd_table[minor];
57260
57261   #if DEBUG
57262    if ((loc_fd->srf_flags & SFF_MINOR) || !(loc_fd->srf_flags & SFF_INUSE))
57263    { where(); printf("got req for ill minor (= %d)\n", minor); }
57264   #endif
57265   assert (!(loc_fd->srf_flags & SFF_MINOR) && (loc_fd->srf_flags & SFF_INUSE));
57266
57267           return loc_fd;
57268   }

57270   PRIVATE void sr_reply (mess_ptr, status)
57271   message *mess_ptr;
57272   int status;
57273   {
57274           static message reply;
```

```
57275           int result;
57276
57277   #if DEBUG & 256
57278    { where(); printf("replying %d to %d for proc %d\n", status,
57279           mess_ptr->m_source, mess_ptr->PROC_NR); }
57280   #endif
57281           reply.m_type= REVIVE;   /* There no use for TASK_REPLY */
57282           reply.REP_PROC_NR= mess_ptr->PROC_NR;
57283           reply.REP_STATUS= status;
57284   #if DEBUG & 256
57285    { where(); printf("sending %d to %d for %d\n", reply.m_type,
57286           mess_ptr->m_source, reply.REP_PROC_NR); }
57287   #endif
57288   assert(mess_ptr->m_source != MM_PROC_NR);
57289           result= send (mess_ptr->m_source, &reply);
57290           if (result != OK)
57291                   ip_panic(("unable to send"));
57292   }

57294   PRIVATE acc_t *sr_get_userdata (fd, offset, count, for_ioctl)
57295   int fd;
57296   vir_bytes offset;
57297   vir_bytes count;
57298   int for_ioctl;
57299   {
57300           sr_fd_t *loc_fd;
57301           mq_t **head_ptr, **tail_ptr, *m, *tail, *mq;
57302           int ip_flag, susp_flag;
57303           int result;
57304           int suspended;
57305           char *src;
57306           acc_t *acc;
57307
57308   #if DEBUG & 256
57309    { where(); printf("sr_get_userdata(%d, %u, %u, %d)\n",
57310           fd, offset, count, for_ioctl); }
57311   #endif
57312           loc_fd= &sr_fd_table[fd];
57313
57314           if (for_ioctl)
57315           {
57316                   head_ptr= &loc_fd->srf_ioctl_q;
57317                   tail_ptr= &loc_fd->srf_ioctl_q_tail;
57318                   ip_flag= SFF_IOCTL_IP;
57319                   susp_flag= SFF_IOCTL_SUSP;
57320           }
57321           else
57322           {
57323                   head_ptr= &loc_fd->srf_write_q;
57324                   tail_ptr= &loc_fd->srf_write_q_tail;
57325                   ip_flag= SFF_WRITE_IP;
57326                   susp_flag= SFF_WRITE_SUSP;
57327           }
57328
57329   assert (loc_fd->srf_flags & ip_flag);
57330
57331           if (!count)
57332           {
57333                   m= *head_ptr;
57334                   *head_ptr= NULL;
```

```
57335                   tail= *tail_ptr;
57336   assert(m);
57337                   mq= m->mq_next;
57338                   result= (int)offset;
57339                   sr_revive (m, result);
57340                   suspended= (loc_fd->srf_flags & susp_flag);
57341                   loc_fd->srf_flags &= ~(ip_flag|susp_flag);
57342                   if (suspended)
57343                   {
57344                           process_req_q(mq, tail, tail_ptr);
57345                   }
57346                   else
57347                   {
57348   assert(!mq);
57349                   }
57350                   return NULL;
57351           }
57352
57353           src= (*head_ptr)->mq_mess.ADDRESS + offset;
57354           result= cp_u2b ((*head_ptr)->mq_mess.PROC_NR, src, &acc, count);
57355
57356           return result<0 ? NULL : acc;
57357   }

57359   PRIVATE int sr_put_userdata (fd, offset, data, for_ioctl)
57360   int fd;
57361   vir_bytes offset;
57362   acc_t *data;
57363   int for_ioctl;
57364   {
57365           sr_fd_t *loc_fd;
57366           mq_t **head_ptr, **tail_ptr, *m, *tail, *mq;
57367           int ip_flag, susp_flag;
57368           int result;
57369           int suspended;
57370           char *dst;
57371
57372   #if DEBUG & 256
57373    { where(); printf("sr_put_userdata(%d, %u, 0x%x, %d)\n",
57374           fd, offset, data, for_ioctl); }
57375   #endif
57376
57377           loc_fd= &sr_fd_table[fd];
57378
57379           if (for_ioctl)
57380           {
57381                   head_ptr= &loc_fd->srf_ioctl_q;
57382                   tail_ptr= &loc_fd->srf_ioctl_q_tail;
57383                   ip_flag= SFF_IOCTL_IP;
57384                   susp_flag= SFF_IOCTL_SUSP;
57385           }
57386           else
57387           {
57388                   head_ptr= &loc_fd->srf_read_q;
57389                   tail_ptr= &loc_fd->srf_read_q_tail;
57390                   ip_flag= SFF_READ_IP;
57391                   susp_flag= SFF_READ_SUSP;
57392           }
57393
57394   assert (loc_fd->srf_flags & ip_flag);
```

```
57395
57396           if (!data)
57397           {
57398                   m= *head_ptr;
57399                   *head_ptr= NULL;
57400                   tail= *tail_ptr;
57401   assert(m);
57402                   mq= m->mq_next;
57403                   result= (int)offset;
57404                   sr_revive (m, result);
57405                   suspended= (loc_fd->srf_flags & susp_flag);
57406                   loc_fd->srf_flags &= ~(ip_flag|susp_flag);
57407                   if (suspended)
57408                   {
57409                           process_req_q(mq, tail, tail_ptr);
57410                   }
57411                   else
57412                   {
57413   assert(!mq);
57414                   }
57415                   return OK;
57416           }
57417
57418           dst= (*head_ptr)->mq_mess.ADDRESS + offset;
57419           return cp_b2u (data, (*head_ptr)->mq_mess.PROC_NR, dst);
57420   }

57422   PRIVATE void sr_revive (m, status)
57423   mq_t *m;
57424   int status;
57425   {
57426           static message reply;
57427           int result;
57428
57429   #if DEBUG & 256
57430    { where(); printf("sr_revive: replying %d to %d for proc %d\n", status,
57431           m->mq_mess.m_source, m->mq_mess.PROC_NR); }
57432   #endif
57433           reply.m_type= REVIVE;
57434           reply.REP_PROC_NR= m->mq_mess.PROC_NR;
57435           reply.REP_STATUS= status;
57436   #if DEBUG & 256
57437    { where(); printf("sending %d to %d for %d\n", reply.m_type,
57438      m->mq_mess.m_source, reply.REP_PROC_NR); }
57439   #endif
57440   assert(m->mq_mess.m_source != MM_PROC_NR);
57441           result= send (m->mq_mess.m_source, &reply);
57442           if (result<0)
57443           {
57444                   if (result == ELOCKED)
57445                   {
57446   #if DEBUG
57447    { where(); printf("send locked\n"); }
57448   #endif
57449                           reply.m_source= m->mq_mess.m_source;
57450                           m->mq_mess= reply;
57451                           if (repl_queue)
57452                                   repl_queue_tail->mq_next= m;
57453                           else
57454                                   repl_queue= m;
```

```
57455                           repl_queue_tail= m;
57456                           return;
57457                   }
57458           else
57459                   ip_panic(("unable to send"));
57460           }
57461           mq_free(m);
57462   }

57464   PRIVATE void process_req_q(mq, tail, tail_ptr)
57465   mq_t *mq, *tail, **tail_ptr;
57466   {
57467           mq_t *m;
57468           int result;
57469
57470           for(;mq;)
57471           {
57472                   m= mq;
57473                   mq= mq->mq_next;
57474
57475   #if DEBUG
57476    { where(); printf("calling rwio\n"); }
57477   #endif
57478                   result= sr_rwio(m);
57479                   if (result == SUSPEND)
57480                   {
57481                           if (mq)
57482                           {
57483                                   (*tail_ptr)->mq_next= mq;
57484                                   *tail_ptr= tail;
57485                           }
57486                           return;
57487                   }
57488           }
57489           return;
57490   }

57492   PRIVATE int cp_u2b (proc, src, var_acc_ptr, size)
57493   int proc;
57494   char *src;
57495   acc_t **var_acc_ptr;
57496   int size;
57497   {
57498           static message mess;
57499           acc_t *acc;
57500           int i;
57501
57502           acc= bf_memreq(size);
57503           *var_acc_ptr= acc;
57504           i=0;
57505
57506           while (acc)
57507           {
57508                   size= (vir_bytes)acc->acc_length;
57509
57510                   cpvec[i].cpv_src= (vir_bytes)src;
57511                   cpvec[i].cpv_dst= (vir_bytes)ptr2acc_data(acc);
57512                   cpvec[i].cpv_size= size;
57513
57514                   src += size;
```

```
57515                   acc= acc->acc_next;
57516                   i++;
57517
57518                   if (i == CPVEC_NR)
57519                   {
57520                           mess.m_type= SYS_VCOPY;
57521                           mess.m1_i1= proc;
57522                           mess.m1_i2= THIS_PROC;
57523                           mess.m1_i3= i;
57524                           mess.m1_p1= (char *)cpvec;
57525                           if (sendrec(SYSTASK, &mess) <0)
57526                                   ip_panic(("unable to sendrec"));
57527                           if (mess.m_type <0)
57528                           {
57529                                   bf_afree(*var_acc_ptr);
57530                                   *var_acc_ptr= 0;
57531                                   return mess.m_type;
57532                           }
57533                           i= 0;
57534                   }
57535           }
57536           if (i)
57537           {
57538                   mess.m_type= SYS_VCOPY;
57539                   mess.m1_i1= proc;
57540                   mess.m1_i2= THIS_PROC;
57541                   mess.m1_i3= i;
57542                   mess.m1_p1= (char *)cpvec;
57543                   if (sendrec(SYSTASK, &mess) <0)
57544                           ip_panic(("unable to sendrec"));
57545                   if (mess.m_type <0)
57546                   {
57547                           bf_afree(*var_acc_ptr);
57548                           *var_acc_ptr= 0;
57549                           return mess.m_type;
57550                   }
57551           }
57552           return OK;
57553   }

57555   PRIVATE int cp_b2u (acc_ptr, proc, dest)
57556   acc_t *acc_ptr;
57557   int proc;
57558   char *dest;
57559   {
57560           static message mess;
57561           acc_t *acc;
57562           int i, size;
57563
57564           acc= acc_ptr;
57565           i=0;
57566
57567           while (acc)
57568           {
57569                   size= (vir_bytes)acc->acc_length;
57570
57571                   if (size)
57572                   {
57573                           cpvec[i].cpv_src= (vir_bytes)ptr2acc_data(acc);
57574                           cpvec[i].cpv_dst= (vir_bytes)dest;
```

```
57575                           cpvec[i].cpv_size= size;
57576                           i++;
57577                   }
57578
57579                   dest += size;
57580                   acc= acc->acc_next;
57581
57582                   if (i == CPVEC_NR)
57583                   {
57584                           mess.m_type= SYS_VCOPY;
57585                           mess.m1_i1= THIS_PROC;
57586                           mess.m1_i2= proc;
57587                           mess.m1_i3= i;
57588                           mess.m1_p1= (char *)cpvec;
57589                           if (sendrec(SYSTASK, &mess) <0)
57590                                   ip_panic(("unable to sendrec"));
57591                           if (mess.m_type <0)
57592                           {
57593                                   bf_afree(acc_ptr);
57594                                   return mess.m_type;
57595                           }
57596                           i= 0;
57597                   }
57598           }
57599           if (i)
57600           {
57601                   mess.m_type= SYS_VCOPY;
57602                   mess.m1_i1= THIS_PROC;
57603                   mess.m1_i2= proc;
57604                   mess.m1_i3= i;
57605                   mess.m1_p1= (char *)cpvec;
57606                   if (sendrec(SYSTASK, &mess) <0)
57607                           ip_panic(("unable to sendrec"));
57608                   if (mess.m_type <0)
57609                   {
57610                           bf_afree(acc_ptr);
57611                           return mess.m_type;
57612                   }
57613           }
57614           bf_afree(acc_ptr);
57615           return OK;
57616   }

57618   PRIVATE int sr_repl_queue(proc)
57619   int proc;
57620   {
57621           mq_t *m, *m_cancel, *tmp;
57622           int result;
57623
57624           m_cancel= NULL;
57625
57626           for (m= repl_queue; m;)
57627           {
57628                   if (m->mq_mess.REP_PROC_NR == proc)
57629                   {
57630   assert(!m_cancel);
57631                           m_cancel= m;
57632                           m= m->mq_next;
57633                           continue;
57634                   }
```

```
57635    assert(m->mq_mess.m_source != MM_PROC_NR);
57636                    result= send(m->mq_mess.m_source, &m->mq_mess);
57637                    if (result != OK)
57638                            ip_panic(("unable to send: %d", result));
57639                    tmp= m;
57640                    m= m->mq_next;
57641                    mq_free(tmp);
57642            }
57643            repl_queue= NULL;
57644            if (m_cancel)
57645            {
57646    assert(m_cancel->mq_mess.m_source != MM_PROC_NR);
57647                    result= send(m_cancel->mq_mess.m_source, &m_cancel->mq_mess);
57648                    if (result != OK)
57649                            ip_panic(("unable to send: %d", result));
57650                    mq_free(m_cancel);
57651                    return 1;
57652            }
57653            return 0;
57654    }
```

++
src/inet/generic/arp.h
++

```
57700    /*
57701    arp.h
57702    */
57703
57704    #ifndef ARP_H
57705    #define ARP_H
57706
57707    #define ARP_ETHERNET    1
57708
57709    #define ARP_REQUEST     1
57710    #define ARP_REPLY       2
57711    #define RARP_REQUEST    3
57712    #define RARP_REPLY      4
57713
57714    /* Prototypes */
57715
57716    void arp_init ARGS(( void ));
57717    int rarp_req ARGS(( int eth_port, int ref,
57718            void (*func)(int fd, ipaddr_t ipaddr) ));
57719    int arp_ip_eth_nonbl ARGS(( int eth_port, ipaddr_t ipaddr,
57720            ether_addr_t *ethaddr ));
57721    int arp_ip_eth ARGS(( int eth_port, int ref, ipaddr_t,
57722            void (*func)(int fd, ether_addr_t*ethadd) ));
57723    void set_ipaddr ARGS(( int eth_port, ipaddr_t ipaddr ));
57724
57725    #endif /* ARP_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/assert.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

57800    /*
57801    assert.h
57802    */
57803    #ifndef INET_ASSERT_H
57804    #define INET_ASSERT_H
57805
57806    #include "const.h"
57807
57808    #ifndef assert
57809    #if NDEBUG
57810    #define assert(x) 0
57811    #define assertN(n,x)    0
57812    #define compare(a,t,b)  0
57813    #else
57814    #define assert(x) (!(x) ? (ip_panic (( "assertion failed" )) \
57815                    ,0) : 0)
57816    #define assertN(n,x) (!(x) ? (ip_panic (( "assertion %d failed", n )),0) : 0)
57817    #define compare(a,t,b) (!((a) t (b)) ? (ip_panic(( "compare failed (%d, %d)", \
57818            a, b )),0) : 0)
57819    #endif
57820    #endif
57821
57822    #endif /* INET_ASSERT_H */
57823


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/buf.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

57900    /*
57901    buf.h
57902    */
57903
57904    #ifndef BUF_H
57905    #define BUF_H
57906
57907    #ifndef BUF_S
57908    #define BUF_S           32768
57909    #endif
57910
57911    #define MAX_BUFREQ_PRI  10
57912
57913    #define ETH_PRI_PORTBUFS        3
57914    #define ETH_PRI_EXP_FDBUFS      4
57915    #define ETH_PRI_FDBUFS          5
57916
57917    #define ICMP_PRI_QUEUE          1
57918
57919    #define TCP_PRI_FRAG2SEND       4
57920    #define TCP_PRI_CONNwoUSER      6
57921    #define TCP_PRI_CONN_INUSE      9
57922
57923    #define UDP_PRI_EXP_FDBUFS      5
57924    #define UDP_PRI_FDBUFS          6
```

```
57925
57926   struct buf;
57927   typedef void (*buffree_t) ARGS(( struct buf *buffer ));
57928   typedef void (*bf_freereq_t) ARGS(( int priority, size_t reqsize ));
57929
57930   typedef struct buf
57931   {
57932           int buf_linkC;
57933           void *buf_next;
57934           buffree_t buf_free;
57935           size_t buf_size;
57936           char *buf_data_p;
57937   } buf_t;
57938
57939   typedef struct acc
57940   {
57941           int acc_linkC;
57942           int acc_offset, acc_length;
57943           buf_t *acc_buffer;
57944           struct acc *acc_next, *acc_ext_link;
57945   } acc_t;
57946
57947   extern size_t bf_free_buffsize;
57948   extern acc_t *bf_temporary_acc;
57949
57950   /* For debugging... */
57951
57952   #if DEBUG & 256
57953   #define bf_memreq(a) (bf_memreq_file= __FILE__, bf_memreq_line= __LINE__, \
57954            _bf_memreq(a))
57955   #endif
57956
57957   #if DEBUG & 256
57958   #define bf_cut(a,b,c) (bf_cut_file= __FILE__, bf_cut_line= __LINE__, \
57959            _bf_cut(a,b,c))
57960   #endif
57961
57962   #if DEBUG & 256
57963   #define bf_packIffLess(a,b) (bf_pack_file= __FILE__, bf_pack_line= __LINE__, \
57964            _bf_packIffLess(a,b))
57965   #endif
57966
57967   #if DEBUG & 256
57968   #define bf_bufsize(a) (bf_bufsize_file= __FILE__, bf_bufsize_line= __LINE__, \
57969            _bf_bufsize(a))
57970   #endif
57971
57972   #ifdef bf_memreq
57973   extern char *bf_memreq_file;
57974   extern int bf_memreq_line;
57975   #endif
57976   #ifdef bf_cut
57977   extern char *bf_cut_file;
57978   extern int bf_cut_line;
57979   #endif
57980   #ifdef bf_packIffLess
57981   extern char *bf_pack_file;
57982   extern int bf_pack_line;
57983   #endif
57984   #ifdef bf_bufsize
```

```
57985   extern char *bf_bufsize_file;
57986   extern int bf_bufsize_line;
57987   #endif
57988
57989   /* Prototypes */
57990
57991   void bf_init ARGS(( void ));
57992   void bf_logon ARGS(( bf_freereq_t func ));
57993
57994   #ifndef bf_memreq
57995   acc_t *bf_memreq ARGS(( unsigned size));
57996   #else
57997   acc_t *_bf_memreq ARGS(( unsigned size));
57998   #endif
57999   /* the result is an acc with linkC == 1 */
58000
58001   acc_t *bf_dupacc ARGS(( acc_t *acc ));
58002   /* the result is an acc with linkC == 1 identical to the given one */
58003
58004   void bf_afree ARGS(( acc_t *acc));
58005   /* this reduces the linkC off the given acc with one */
58006
58007   acc_t *bf_pack ARGS(( acc_t *pack));
58008   /* this gives a packed copy of the given acc, the linkC of the given acc is
58009      reduced by one, the linkC of the result == 1 */
58010
58011   #ifndef bf_packIffLess
58012   acc_t *bf_packIffLess ARGS(( acc_t *pack, int min_len ));
58013   #else
58014   acc_t *_bf_packIffLess ARGS(( acc_t *pack, int min_len ));
58015   #endif
58016   /* this performs a bf_pack iff pack->acc_length<min_len */
58017
58018   #ifndef bf_bufsize
58019   size_t bf_bufsize ARGS(( acc_t *pack));
58020   #else
58021   size_t _bf_bufsize ARGS(( acc_t *pack));
58022   #endif
58023   /* this gives the length of the buffer specified by the given acc. The linkC
58024      of the given acc remains the same */
58025
58026   #ifndef bf_cut
58027   acc_t *bf_cut ARGS(( acc_t *data, unsigned offset,
58028           unsigned length ));
58029   #else
58030   acc_t *_bf_cut ARGS(( acc_t *data, unsigned offset,
58031           unsigned length ));
58032   #endif
58033   /* the result is a cut of the buffer from offset with length length.
58034      The linkC of the result == 1, the linkC of the given acc remains the
58035      same. */
58036
58037   acc_t *bf_append ARGS(( acc_t *data_first, acc_t  *data_second ));
58038   /* data_second is appended after data_first, a link is returned to the
58039           result and the linkCs of data_first and data_second are reduced.
58040           further more, if the contents of the last part of data_first and
58041           the first part of data_second fit in a buffer, both parts are
58042           copied into a (possibly fresh) buffer
58043   */
58044
```

```
58045    #define ptr2acc_data(/* acc_t * */ a) (bf_temporary_acc=(a), \
58046            (&bf_temporary_acc->acc_buffer->buf_data_p[bf_temporary_acc-> \
58047                    acc_offset]))
58048
58049    #define bf_chkbuf(buf) ((buf)? (assert((buf)->acc_linkC>0), \
58050            assert((buf)->acc_buffer), \
58051            assert((buf)->acc_buffer->buf_linkC>0)) : 0)
58052
58053    void bf_check_all_bufs ARGS(( void ));
58054    /* try to get all buffers back for debug purposes */
58055
58056
58057    #endif /* BUF_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/clock.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
58100    /*
58101    clock.h
58102    */
58103
58104    #ifndef CLOCK_H
58105    #define CLOCK_H
58106
58107    struct timer;
58108
58109    typedef void (*timer_func_t) ARGS(( int fd, struct timer *timer ));
58110
58111    typedef struct timer
58112    {
58113            struct timer *tim_next;
58114            timer_func_t tim_func;
58115            int tim_ref;
58116            time_t tim_time;
58117    } timer_t;
58118
58119    void clck_init ARGS(( void ));
58120    void set_time ARGS(( time_t time ));
58121    time_t get_time ARGS(( void ));
58122    void reset_time ARGS(( void ));
58123    void clck_timer ARGS(( struct timer *timer, time_t timeout, timer_func_t func,
58124                                                            int fd ));
58125                    /* set a timer to go off at the time specified by timeout */
58126    void clck_untimer ARGS(( struct timer *timer ));
58127
58128
58129    #endif /* CLOCK_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/eth.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

58200    /*
58201    eth.h
58202    */
58203
58204    #ifndef ETH_H
58205    #define ETH_H
58206
58207    #define NWEO_DEFAULT    (NWEO_EN_LOC | NWEO_DI_BROAD | NWEO_DI_MULTI | \
58208            NWEO_DI_PROMISC | NWEO_REMANY | NWEO_RWDATALL)
58209
58210    #define ETH0    0       /* port number of ethernet port 0 */
58211
58212    #define eth_addrcmp(a,b) (memcmp((_VOIDSTAR)&a, (_VOIDSTAR)&b, \
58213            sizeof(a)))
58214
58215    /* Forward declatations */
58216
58217    struct acc;
58218
58219    /* prototypes */
58220
58221    void eth_init ARGS(( void ));
58222    int eth_open ARGS(( int port, int srfd,
58223            struct acc *(*get_userdata)(int fd, size_t offset,size_t count,
58224                    int for_ioctl),
58225            int (*put_userdata)(int fd, size_t offset, struct acc *data,
58226                    int for_ioctl) ));
58227    int eth_ioctl ARGS(( int fd, int req));
58228    int eth_read ARGS(( int port, size_t count ));
58229    int eth_write ARGS(( int port, size_t count ));
58230    int eth_cancel ARGS(( int fd, int which_operation ));
58231    void eth_close ARGS(( int fd ));
58232
58233    #endif /* ETH_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/eth_int.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

58300    /*
58301    eth_int.h
58302    */
58303
58304    #ifndef ETH_INT_H
58305    #define ETH_INT_H
58306
58307    #define ETH_PORT_NR     1       /* 1 ethernet connection */
58308
58309    typedef struct eth_port
58310    {
58311            int etp_flags;
58312            ether_addr_t etp_ethaddr;
58313            acc_t *etp_wr_pack, *etp_rd_pack;
58314
```

```
58315           osdep_eth_port_t etp_osdep;
58316   } eth_port_t;
58317
58318   #define EPF_EMPTY       0x0
58319   #define EPF_ENABLED     0x1
58320   #define EPF_WRITE_IP    0x2
58321   #define EPF_WRITE_SP    0x4
58322   #define EPF_MORE2WRITE  0x8
58323   #define EPF_READ_IP     0x10
58324   #define EPF_READ_SP     0x20
58325
58326   #define EPS_EMPTY       0x0
58327   #define EPS_LOC         0x1
58328   #define EPS_BROAD       0x2
58329   #define EPS_MULTI       0x4
58330   #define EPS_PROMISC     0x8
58331
58332   extern eth_port_t eth_port_table[ETH_PORT_NR];
58333
58334   void eth_init0 ARGS(( void ));
58335   int eth_get_stat ARGS(( eth_port_t *eth_port, eth_stat_t *eth_stat ));
58336   void eth_write_port ARGS(( eth_port_t *eth_port ));
58337   void eth_arrive ARGS(( eth_port_t *port, acc_t *pack ));
58338   void eth_set_rec_conf ARGS(( eth_port_t *eth_port, u32_t flags ));
58339   int eth_get_work ARGS(( eth_port_t *eth_port ));
58340
58341   #endif /* ETH_INT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/icmp.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
58400   /*
58401   icmp.h
58402   */
58403
58404   #ifndef ICMP_H
58405   #define ICMP_H
58406
58407   #define ICMP_MAX_DATAGRAM       8196
58408   #define ICMP_DEF_TTL            60
58409
58410   /* Prototypes */
58411
58412   void icmp_init ARGS(( void ));
58413
58414
58415   #endif /* ICMP_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/icmp_lib.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

58500    /*
58501    icmp_lib.h
58502
58503    Created Sept 30, 1991 by Philip Homburg
58504    */
58505
58506    #ifndef ICMP_LIB_H
58507    #define ICMP_LIB_H
58508
58509    /* Prototypes */
58510
58511    void icmp_getnetmask ARGS((  int ip_port ));
58512    void icmp_parmproblem ARGS(( acc_t *pack ));
58513    void icmp_frag_ass_tim ARGS(( acc_t *pack ));
58514    void icmp_dont_frag ARGS(( acc_t *pack ));
58515    void icmp_ttl_exceded ARGS(( acc_t *pack ));
58516
58517    #endif /* ICMP_LIB_H */
58518


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/io.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

58600    /*
58601    io.h
58602
58603    Created Sept 30, 1991 by Philip Homburg
58604    */
58605
58606    #ifndef IO_H
58607    #define IO_H
58608
58609    /* Prototypes */
58610
58611    void writeIpAddr ARGS(( ipaddr_t addr ));
58612    void writeEtherAddr ARGS(( ether_addr_t *addr ));
58613
58614    #endif /* IO_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/ip.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

58700    /*
58701    ip.h
58702    */
58703
58704    #ifndef INET_IP_H
58705    #define INET_IP_H
58706
58707    #define IP0     0
58708
58709    /* Prototypes */
58710
58711    struct acc;
58712
58713    void ip_init ARGS(( void ));
58714    int  ip_open ARGS(( int port, int srfd,
58715            struct acc *(*get_userdata) (int fd, size_t offset, size_t count,
58716                    int for_ioctl),
58717            int (*put_userdata) (int fd, size_t offset, struct acc *data,
58718                    int for_ioctl) ));
58719    int ip_ioctl ARGS(( int fd, int req ));
58720    int ip_read ARGS(( int fd, size_t count ));
58721    int ip_write ARGS(( int fd, size_t count ));
58722
58723    #endif /* INET_IP_H */


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/ip_int.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

58800    /*
58801    ip.h
58802    */
58803
58804    #ifndef INET_IP_INT_H
58805    #define INET_IP_INT_H
58806
58807    /* this include file depends on:
58808    #include <ansi.h>
58809    #include <sys/types.h>
58810    #include <minix/type.h>
58811    #include <inet/ether.h>
58812    #include <inet/in.h>
58813    #include <inet/ip_io.h>
58814    #include "buf.h"
58815    #include "type.h"
58816    */
58817
58818    #define IP_FD_NR        32
58819    #define IP_PORT_NR      1
58820    #define IP_ASS_NR       3
58821
58822    #define IP_SUN_BROADCAST        1       /* hostnumber 0 is also network
58823                                               broadcast */
58824    #define IP_ROUTER               0       /* this implementation isn't a
```

```
58825                                           gateway */
58826
58827   typedef struct ip_port
58828   {
58829           int ip_flags, ip_dl_type;
58830           union
58831           {
58832                   struct
58833                   {
58834                           int de_state;
58835                           int de_flags;
58836                           int de_port;
58837                           int de_fd;
58838                           acc_t *de_wr_frag;
58839                           acc_t *de_wr_frame;
58840                           ether_addr_t de_wr_ethaddr;
58841                           ipaddr_t de_wr_ipaddr;
58842                           acc_t *de_arp_pack;
58843                           ether_addr_t de_arp_ethaddr;
58844                   } dl_eth;
58845           } ip_dl;
58846           int ip_minor;
58847           ipaddr_t ip_ipaddr, ip_netmask;
58848           u16_t ip_frame_id;
58849   } ip_port_t;
58850
58851   #define IES_EMPTY       0x0
58852   #define IES_SETPROTO    0x1
58853   #define IES_GETIPADDR   0x2
58854   #define IES_MAIN        0x3
58855   #define IES_ERROR       0x4
58856
58857   #define IEF_EMPTY       0x1
58858   #define IEF_WRITE_IP    0x2
58859   #define IEF_WRITE_SP    0x4
58860   #define IEF_SUSPEND     0x8
58861   #define IEF_READ_IP     0x10
58862   #define IEF_READ_SP     0x20
58863   #define IEF_ARP_MASK    0x1c0
58864   #       define IEF_ARP_IP       0x40
58865   #       define IEF_ARP_SP       0x80
58866   #       define IEF_ARP_COMPL    0x100
58867
58868   #define IPF_EMPTY       0x0
58869   #define IPF_IPADDRSET   0x1
58870   #define IPF_NETMASKSET  0x2
58871
58872   #define IPDL_ETH        0
58873
58874   typedef struct ip_ass
58875   {
58876           acc_t *ia_frags;
58877           int ia_min_ttl;
58878           ip_port_t *ia_port;
58879           time_t ia_first_time;
58880           ipaddr_t ia_srcaddr, ia_dstaddr;
58881           int ia_proto, ia_id;
58882   } ip_ass_t;
58883
58884   typedef struct ip_fd
```

```
58885   {
58886           int if_flags;
58887           struct nwio_ipopt if_ipopt;
58888           ip_port_t *if_port;
58889           int if_srfd;
58890           acc_t *if_rd_buf;
58891           get_userdata_t if_get_userdata;
58892           put_userdata_t if_put_userdata;
58893           time_t if_exp_tim;
58894           size_t if_rd_count;
58895           ipaddr_t if_wr_dstaddr;
58896           size_t if_wr_count;
58897           ip_port_t *if_wr_port;
58898   } ip_fd_t;
58899
58900   #define IFF_EMPTY       0x0
58901   #define IFF_INUSE       0x1
58902   #define IFF_OPTSET      0x2
58903   #define IFF_BUSY        0x7f4
58904   #       define IFF_READ_IP      0x4
58905   #       define IFF_WRITE_MASK   0x3f0
58906   #               define IFF_WRITE_IP     0x10
58907   #               define IFF_DLL_WR_IP    0x20
58908   #               define IFF_ROUTED       0x40
58909   #               define IFF_NETBROAD_IP  0x200
58910   #       define IFF_GIPCONF_IP   0x400
58911
58912
58913   /* ip_lib.c */
58914   ipaddr_t ip_get_netmask ARGS(( ipaddr_t hostaddr ));
58915   int ip_chk_hdropt ARGS(( u8_t *opt, int optlen ));
58916   void ip_print_frags ARGS(( acc_t *acc ));
58917
58918   /* ip_read.c */
58919   void ip_port_arrive ARGS(( ip_port_t *port, acc_t *pack, ip_hdr_t *ip_hdr ));
58920   void ip_eth_arrived ARGS(( ip_port_t *port, acc_t *pack ));
58921   int ip_ok_for_fd ARGS(( ip_fd_t *ip_fd, acc_t *pack ));
58922   int ip_packet2user ARGS(( ip_fd_t *ip_fd ));
58923
58924   /* ip_write.c */
58925   void dll_eth_write_frame ARGS(( ip_port_t *port ));
58926
58927   extern ip_fd_t ip_fd_table[IP_FD_NR];
58928   extern ip_port_t ip_port_table[IP_PORT_NR];
58929   extern ip_ass_t ip_ass_table[IP_ASS_NR];
58930
58931
58932   #define NWIO_DEFAULT    (NWIO_EN_LOC | NWIO_EN_BROAD | NWIO_REMANY | \
58933           NWIO_RWDATALL | NWIO_HDR_O_SPEC)
58934
58935   #endif /* INET_IP_INT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/ipr.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

59000    /*
59001    ipr.h
59002    */
59003
59004    #ifndef IPR_H
59005    #define IPR_H
59006
59007    typedef struct route
59008    {
59009            ipaddr_t rt_dest;
59010            ipaddr_t rt_gateway;
59011            ipaddr_t rt_netmask;
59012            time_t rt_exp_tim;
59013            time_t rt_timestamp;
59014            int rt_dist;
59015            int rt_port;
59016            int rt_flags;
59017            i32_t rt_pref;
59018    } route_t;
59019
59020    #define RTF_EMPTY       0
59021    #define RTF_INUSE       1
59022    #define RTF_FIXED       2
59023
59024    #define IPR_MAX_FIXED_ROUTES    16
59025
59026    #define IPR_UNRCH_TIMEOUT       (60L * HZ)
59027    #define IPR_TTL_TIMEOUT         (60L * HZ)
59028    #define IPR_REDIRECT_TIMEOUT    (20 * 60L * HZ)
59029    #define IPR_GW_DOWN_TIMEOUT     (60L * HZ)
59030
59031    /* Prototypes */
59032
59033    int iproute_frag ARGS(( ipaddr_t dest, int ttl, ipaddr_t *nexthop,
59034            int *port ));
59035    void ipr_init ARGS(( void ));
59036    route_t *ipr_add_route ARGS(( ipaddr_t dest, ipaddr_t netmask,
59037            ipaddr_t gateway, int port, time_t timeout, int dist, int fixed,
59038            i32_t freference ));
59039    void ipr_gateway_down ARGS(( ipaddr_t gateway, time_t timeout ));
59040    void ipr_redirect ARGS(( ipaddr_t dest, ipaddr_t netmask,
59041            ipaddr_t old_gateway, ipaddr_t new_gateway, int new_port,
59042            time_t timeout ));
59043    void ipr_destunrch ARGS(( ipaddr_t dest, ipaddr_t netmask,
59044            time_t timeout ));
59045    void ipr_ttl_exc ARGS(( ipaddr_t dest, ipaddr_t netmask,
59046            time_t timeout ));
59047    int ipr_get_route ARGS(( int ent_no, nwio_route_t *route_ent ));
59048
59049    #endif /* IPR_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/sr.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

59100    /*
59101    sr.h
59102    */
59103
59104    #ifndef SR_H
59105    #define SR_H
59106
59107    #define MAX_IOCTL_S     512
59108
59109    #define ETH_DEV         ETH_DEV0
59110    #define IP_DEV          IP_DEV0
59111
59112    #define ETH_DEV0        1
59113    #define IP_DEV0         2
59114    #define TCP_DEV0        3
59115    #define UDP_DEV0        4
59116
59117    #define SR_CANCEL_IOCTL 1
59118    #define SR_CANCEL_READ  2
59119    #define SR_CANCEL_WRITE 3
59120
59121    /* Forward struct declarations */
59122
59123    struct acc;
59124
59125    /* prototypes */
59126
59127    typedef int  (*sr_open_t) ARGS(( int port, int srfd,
59128            struct acc *(*get_userdata)(int fd, size_t offset,size_t count,
59129                            int for_ioctl),
59130            int (*put_userdata)(int fd, size_t offset, struct acc *data,
59131                            int for_ioctl) ));
59132    typedef void (*sr_close_t) ARGS(( int fd ));
59133    typedef int (*sr_read_t) ARGS(( int fd, size_t count ));
59134    typedef int (*sr_write_t) ARGS(( int fd, size_t count ));
59135    typedef int  (*sr_ioctl_t) ARGS(( int fd, int req ));
59136    typedef int  (*sr_cancel_t) ARGS(( int fd, int which_operation ));
59137
59138    void sr_init ARGS(( void  ));
59139    int sr_add_minor ARGS(( int minor, int port, sr_open_t openf,
59140            sr_close_t closef, sr_read_t sr_read, sr_write_t sr_write,
59141            sr_ioctl_t ioctlf, sr_cancel_t cancelf ));
59142
59143    #endif /* SR_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/tcp.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

59200    /*
59201    tcp.h
59202    */
59203
59204    #ifndef TCP_H
59205    #define TCP_H
59206
59207    #define ISS_INC_FREQ    250000L
59208    #define TCP_MAX_DATAGRAM        8192
59209    #define TCP_MAX_WND_SIZE        (4*1024)
59210    #define SUN_0WND_BUG            1
59211            /* the sun 4.x.y implementation of tcp/ip does not send zero
59212               windows but instead does not acknowledge new data */
59213    #define SUN_TRANS_BUG           0
59214            /* the sun 4.x.y implementation of tcp/ip does not send fast
59215               as posible until a zero window is reached but use the
59216               round trip delay as a measurement. It is then not possible
59217               to delay ACKs */
59218
59219    #define TCP_DEF_URG_WND         1024
59220    #define TCP_DEF_TOS             0
59221    #define TCP_DEF_TTL             1       /* seconds */
59222    #define TCP_DEF_TIME_OUT        HZ      /* 1 second, in clock ticks */
59223    #define TCP_DEF_MAX_NO_RETRANS  10000
59224    #define TCP_DEF_RTT             15      /* initial retransmission time in
59225                                               ticks */
59226    #define TCP_DEF_MSS             1400
59227    #if SUN_TRANS_BUG
59228    #define TCP_ACK_DELAY           1       /* no delay */
59229    #else
59230    #define TCP_ACK_DELAY           (HZ/2)  /* .5 second is clock ticks */
59231    #endif
59232
59233    #define TCP_DEF_OPT             (NWTC_COPY | NWTC_LP_UNSET | NWTC_UNSET_RA | \
59234                                            NWTC_UNSET_RP)
59235
59236    #define TCP0                    0
59237
59238    struct acc;
59239
59240    void tcp_init ARGS(( void ));
59241    int tcp_open ARGS(( int port, int srfd,
59242            struct acc *(*get_userdata) (int fd, size_t offset, size_t count,
59243                    int for_ioctl),
59244            int (*put_userdata) (int fd, size_t offset, struct acc *data,
59245                    int for_ioctl) ));
59246    int tcp_read ARGS(( int fd, size_t count));
59247    int tcp_write ARGS(( int fd, size_t count));
59248    int tcp_ioctl ARGS(( int fd, int req));
59249    int tcp_cancel ARGS(( int fd, int which_operation ));
59250    void tcp_close ARGS(( int fd));
59251
59252    #endif /* TCP_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/tcp_delay.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

59300    /*
59301    tcp_delay.h
59302    */
59303
59304    #ifndef TCP_DELAY__H
59305    #define TCP_DELAY__H
59306
59307    extern int tcp_delay_on;
59308    extern u32_t tcp_delay;
59309
59310    #endif /* TCP_DELAY__H */
59311
59312
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/tcp_int.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

59400    /*
59401    tcp_int.h
59402    */
59403
59404    #ifndef TCP_INT_H
59405    #define TCP_INT_H
59406
59407    typedef struct tcp_port
59408    {
59409            int tp_minor;
59410            int tp_ipdev;
59411            int tp_flags;
59412            int tp_state;
59413            int tp_ipfd;
59414            acc_t *tp_pack;
59415            ipaddr_t tp_ipaddr;
59416    #if DEBUG
59417            timer_t tp_delay_tim;
59418    #endif
59419    } tcp_port_t;
59420
59421    #define TPF_EMPTY       0x0
59422    #define TPF_SUSPEND     0x1
59423    #define TPF_READ_IP     0x2
59424    #define TPF_READ_SP     0x4
59425    #define TPF_WRITE_IP    0x8
59426    #define TPF_WRITE_SP    0x10
59427    #define TPF_MORE2WRITE  0x20
59428    #define TPF_DELAY_TCP   0x40
59429
59430    #define TPS_EMPTY       0
59431    #define TPS_SETPROTO    1
59432    #define TPS_GETCONF     2
59433    #define TPS_MAIN        3
59434    #define TPS_ERROR       4
```

```
59435
59436   typedef struct tcp_fd
59437   {
59438           int tf_flags;
59439           tcp_port_t *tf_port;
59440           int tf_srfd;
59441           int tf_ioreq;
59442           nwio_tcpconf_t tf_tcpconf;
59443           get_userdata_t tf_get_userdata;
59444           put_userdata_t tf_put_userdata;
59445           struct tcp_conn *tf_conn;
59446           size_t tf_write_offset;
59447           size_t tf_write_count;
59448           size_t tf_read_offset;
59449           size_t tf_read_count;
59450   } tcp_fd_t;
59451
59452   #define TFF_EMPTY       0x0
59453   #define TFF_INUSE       0x1
59454   #define TFF_IOCTL_IP    0x2
59455   #define TFF_OPTSET      0x4
59456   #define TFF_IOC_INIT_SP 0x8
59457   #define TFF_CONNECT     0x20
59458   #define TFF_WRITE_IP    0x80
59459   #define TFF_WR_URG      0x100
59460   #define TFF_PUSH_DATA   0x200
59461   #define TFF_READ_IP     0x400
59462   #define TFF_RECV_URG    0x800
59463   #define TFF_CONNECTED   0x1000
59464
59465   typedef struct tcp_conn
59466   {
59467           int tc_flags;
59468           tcpport_t tc_locport;
59469           ipaddr_t tc_locaddr;
59470           tcpport_t tc_remport;
59471           ipaddr_t tc_remaddr;
59472           int tc_state;
59473           tcp_fd_t *tc_mainuser;
59474           tcp_fd_t *tc_readuser;
59475           tcp_fd_t *tc_writeuser;
59476           tcp_fd_t *tc_connuser;
59477           int tc_orglisten;
59478           time_t tc_senddis;
59479           u32_t tc_SND_UNA;       /* least unacknowledged sequence number */
59480           u32_t tc_SND_TRM;       /* next sequence number to be transmitted */
59481           u32_t tc_SND_NXT;       /* next sequence number for new data */
59482           u32_t tc_SND_UP;        /* urgent pointer, first sequence number not
59483                                    * urgent */
59484           u32_t tc_SND_PSH;       /* push pointer, data should be pushed until the
59485                                    * push pointer is reached */
59486           u32_t tc_SND_WL1;
59487           u32_t tc_SND_WL2;
59488           u32_t tc_ISS;           /* initial sequence number */
59489           u32_t tc_RCV_LO;
59490           u32_t tc_RCV_NXT;
59491           u32_t tc_RCV_HI;
59492           u32_t tc_RCV_UP;
59493           u32_t tc_IRS;
59494           tcp_port_t *tc_port;
```

```
59495           acc_t *tc_rcvd_data;
59496           acc_t *tc_rcv_queue;
59497           acc_t *tc_send_data;
59498           acc_t *tc_remipopt;
59499           acc_t *tc_remtcpopt;
59500           acc_t *tc_frag2send;
59501           u8_t tc_tos;
59502           u8_t tc_ttl;
59503           u16_t tc_rcv_wnd;
59504           u16_t tc_urg_wnd;
59505           int tc_no_retrans;
59506           int tc_max_no_retrans;
59507           time_t tc_0wnd_to;
59508           time_t tc_rtt;
59509           time_t tc_ett;
59510           struct timer tc_major_timer;
59511           struct timer tc_minor_timer;
59512           struct timer tc_ack_timer;
59513           struct timer tc_time_wait_timer;
59514           u16_t tc_mss;
59515           u32_t tc_snd_cwnd;      /* highest sequence number to be sent */
59516           u32_t tc_snd_cthresh;   /* threshold for send window */
59517           u32_t tc_snd_cinc;      /* increment for send window threshold */
59518           u16_t tc_snd_wnd;       /* max send queue size */
59519           int tc_error;
59520   } tcp_conn_t;
59521
59522   #define TCF_EMPTY               0x0
59523   #define TCF_INUSE               0x1
59524   #define TCF_FIN_RECV            0x2
59525   #define TCF_RCV_PUSH            0x4
59526   #define TCF_MORE2WRITE          0x8
59527   #define TCF_SEND_ACK            0x10
59528   #define TCF_FIN_SENT            0x20
59529   #define TCF_ACK_TIMER_SET       0x40
59530
59531   #define TCS_CLOSED              0
59532   #define TCS_LISTEN              1
59533   #define TCS_SYN_RECEIVED        2
59534   #define TCS_SYN_SENT            3
59535   #define TCS_ESTABLISHED         4
59536   #define TCS_FIN_WAIT_1          5
59537   #define TCS_FIN_WAIT_2          6
59538   #define TCS_CLOSE_WAIT          7
59539   #define TCS_CLOSING             8
59540   #define TCS_LAST_ACK            9
59541   #define TCS_TIME_WAIT           10
59542
59543   /* tcp_recv.c */
59544   void tcp_frag2conn ARGS(( tcp_conn_t *tcp_conn,
59545           acc_t *ip_pack, acc_t *tcp_pack ));
59546   void tcp_restart_fd_read ARGS(( tcp_conn_t *tcp_conn ));
59547
59548   /* tcp_send.c */
59549   void tcp_restart_write ARGS(( tcp_conn_t *tcp_conn ));
59550   void tcp_set_ack_timer ARGS(( tcp_conn_t *tcp_conn ));
59551   void tcp_release_retrans ARGS(( tcp_conn_t *tcp_conn, u32_t seg_ack,
59552           U16_t new_win ));
59553   void tcp_set_time_wait_timer ARGS(( tcp_conn_t *tcp_conn ));
59554   void tcp_restart_fd_write ARGS(( tcp_conn_t *tcp_conn ));
```

```
59555   void tcp_close_connection ARGS(( tcp_conn_t *tcp_conn,
59556           int error ));
59557   void tcp_restart_write_port ARGS(( tcp_port_t *tcp_port ));
59558   void tcp_zero_wnd_to ARGS(( int conn, struct timer *timer ));
59559   void tcp_shutdown ARGS(( tcp_conn_t *tcp_conn ));
59560
59561   /* tcp_lib.c */
59562   int tcp_LEmod4G ARGS(( u32_t n1, u32_t n2 ));
59563   int tcp_Lmod4G ARGS(( u32_t n1, u32_t n2 ));
59564   int tcp_GEmod4G ARGS(( u32_t n1, u32_t n2 ));
59565   int tcp_Gmod4G ARGS(( u32_t n1, u32_t n2 ));
59566   void tcp_write_state ARGS(( tcp_conn_t *tcp_conn ));
59567
59568   void tcp_extract_ipopt ARGS(( tcp_conn_t *tcp_conn,
59569           ip_hdr_t *ip_hdr ));
59570   void tcp_extract_tcpopt ARGS(( tcp_conn_t *tcp_conn,
59571           tcp_hdr_t *tcp_hdr ));
59572   void tcp_get_ipopt ARGS(( tcp_conn_t *tcp_conn, ip_hdropt_t
59573           *ip_hdropt ));
59574   void tcp_get_tcpopt ARGS(( tcp_conn_t *tcp_conn, tcp_hdropt_t
59575           *tcp_hdropt ));
59576   acc_t *tcp_make_header ARGS(( tcp_conn_t *tcp_conn,
59577           ip_hdr_t **ref_ip_hdr, tcp_hdr_t **ref_tcp_hdr, acc_t *data ));
59578   u16_t tcp_pack_oneCsum ARGS(( acc_t *pack,
59579           size_t pack_length ));
59580   int tcp_check_conn ARGS(( tcp_conn_t *tcp_conn ));
59581   void tcp_print_pack ARGS(( ip_hdr_t *ip_hdr, tcp_hdr_t *tcp_hdr ));
59582   void tcp_print_conn ARGS(( tcp_conn_t *tcp_conn ));
59583
59584   /* tcp.c */
59585   void tcp_restart_connect ARGS(( tcp_fd_t *tcp_fd ));
59586   int tcp_su4listen ARGS(( tcp_fd_t *tcp_fd ));
59587   void tcp_reply_ioctl ARGS(( tcp_fd_t *tcp_fd, int reply ));
59588   void tcp_reply_write ARGS(( tcp_fd_t *tcp_fd, size_t reply ));
59589   void tcp_reply_read ARGS(( tcp_fd_t *tcp_fd, size_t reply ));
59590
59591   #define TCP_PORT_NR     1
59592   #define TCP_FD_NR       20
59593   #define TCP_CONN_NR     20
59594
59595   EXTERN tcp_port_t tcp_port_table[TCP_PORT_NR];
59596   EXTERN tcp_conn_t tcp_conn_table[TCP_CONN_NR];
59597   EXTERN tcp_fd_t tcp_fd_table[TCP_FD_NR];
59598
59599   #endif /* TCP_INT_H */
```

++
src/inet/generic/type.h
++

```
59600    /*
59601    type.h
59602    */
59603
59604    #ifndef INET_TYPE_H
59605    #define INET_TYPE_H
59606
59607    typedef struct acc *(*get_userdata_t) ARGS(( int fd, size_t offset,
59608            size_t count, int for_ioctl ));
59609    typedef int (*put_userdata_t) ARGS(( int fd, size_t offset,
59610            struct acc *data, int for_ioctl ));
59611    typedef void (*arp_req_func_t) ARGS(( int fd, ether_addr_t
59612            *ethaddr ));
59613    typedef void (*rarp_func_t) ARGS(( int fd, ipaddr_t ipaddr ));
59614
59615    #endif /* INET_TYPE_H */
```

++
src/inet/generic/udp.h
++

```
59700    /*
59701    udp.h
59702    */
59703
59704    #ifndef UDP_H
59705    #define UDP_H
59706
59707    #define UDP_DEF_OPT             NWUO_NOFLAGS
59708    #define UDP_MAX_DATAGRAM        40000 /* 8192 */
59709    #define UDP_READ_EXP_TIME       (10L * HZ)
59710    #define UDP_TOS                 0
59711    #define UDP_IP_FLAGS            0
59712    #define UDP_TTL                 30
59713
59714    #define UDP0    0
59715
59716    struct acc;
59717
59718    void udp_init ARGS(( void ));
59719    int udp_open ARGS(( int port, int srfd,
59720            struct acc *(*get_userdata) (int fd, size_t offset, size_t count,
59721                    int for_ioctl),
59722            int (*put_userdata) (int fd, size_t offset, struct acc *data,
59723                    int for_ioctl) ));
59724    int udp_ioctl ARGS(( int fd, int req ));
59725    int udp_read ARGS(( int fd, size_t count ));
59726    int udp_write ARGS(( int fd, size_t count ));
59727    void udp_close ARGS(( int fd ));
59728    int udp_cancel ARGS(( int fd, int which_operation ));
59729
```

```
59730    #endif /* UDP_H */
59731
```

++
src/inet/generic/arp.c
++

```
59800    /*
59801    arp.c
59802    */
59803
59804    #include "inet.h"
59805
59806    #include "arp.h"
59807    #include "assert.h"
59808    #include "buf.h"
59809    #include "clock.h"
59810    #include "eth.h"
59811    #include "io.h"
59812    #include "sr.h"
59813    #include "type.h"
59814
59815    INIT_PANIC();
59816
59817    #define ARP_PORT_NR     1
59818    #define ARP_CACHE1_NR   8
59819    #define ARP_CACHE2_NR   8
59820    #define ARP_CACHE3_NR   16
59821    #define ARP_CACHE_NR    (ARP_CACHE1_NR+ARP_CACHE2_NR+ARP_CACHE3_NR)
59822    #define ARP_TYPE1       1
59823    #define ARP_TYPE2       2
59824    #define ARP_TYPE3       3
59825
59826    #define MAX_RARP_RETRIES        5
59827    #define MAX_ARP_RETRIES         5
59828    #define RARP_TIMEOUT            (1*HZ)
59829    #define ARP_TIMEOUT             (HZ/2+1)        /* .5 seconds */
59830    #define ARP_EXP_TIME            (20L*60L*HZ)    /* 20 minutes */
59831    #define ARP_NOTRCH_EXP_TIME     (2L*60L*HZ)     /* 2 minutes */
59832    #define ARP_INUSE_OFFSET        (60*HZ) /* an entry in the cache can be deleted
59833                                               if its not used for 1 minute */
59834
59835    typedef struct arp46
59836    {
59837            ether_addr_t a46_dstaddr;
59838            ether_addr_t a46_srcaddr;
59839            ether_type_t a46_ethtype;
59840            union
59841            {
59842                    struct
59843                    {
59844                            u16_t a_hdr, a_pro;
59845                            u8_t a_hln, a_pln;
59846                            u16_t a_op;
59847                            ether_addr_t a_sha;
59848                            u8_t a_spa[4];
59849                            ether_addr_t a_tha;
```

```
59850                           u8_t a_tpa[4];
59851                   } a46_data;
59852                   char    a46_dummy[ETH_MIN_PACK_SIZE-ETH_HDR_SIZE];
59853           } a46_data;
59854   } arp46_t, rarp46_t;
59855
59856   #define a46_hdr a46_data.a46_data.a_hdr
59857   #define a46_pro a46_data.a46_data.a_pro
59858   #define a46_hln a46_data.a46_data.a_hln
59859   #define a46_pln a46_data.a46_data.a_pln
59860   #define a46_op a46_data.a46_data.a_op
59861   #define a46_sha a46_data.a46_data.a_sha
59862   #define a46_spa a46_data.a46_data.a_spa
59863   #define a46_tha a46_data.a46_data.a_tha
59864   #define a46_tpa a46_data.a46_data.a_tpa
59865
59866   typedef struct arp_cache
59867   {
59868           int ac_flags;
59869           int ac_type;
59870           ether_addr_t ac_ethaddr;
59871           ipaddr_t ac_ipaddr;
59872           int ac_eth_port;
59873           time_t ac_expire;
59874           time_t ac_lastuse;
59875   } arp_cache_t;
59876
59877   #define ACF_EMPTY       0
59878   #define ACF_NETREQ      1
59879   #define ACF_NOTRCH      2
59880
59881   typedef struct arp_port
59882   {
59883           int ap_flags;
59884           int ap_state;
59885           int ap_eth_port;
59886           int ap_eth_fd;
59887           int ap_rarp_retries;
59888           ether_addr_t ap_ethaddr;
59889           ipaddr_t ap_ipaddr;
59890           timer_t ap_timer;
59891           ether_addr_t ap_write_ethaddr;
59892           ipaddr_t ap_write_ipaddr;
59893           int ap_write_code;
59894           ipaddr_t ap_req_ipaddr;
59895           arp_req_func_t ap_req_func;
59896           int ap_req_ref;
59897           int ap_req_count;
59898           rarp_func_t ap_rarp_func;
59899           int ap_rarp_ref;
59900   } arp_port_t;
59901
59902   #define APF_EMPTY       0
59903   #define APF_RARP_RD_IP  0x1
59904   #define APF_RARP_RD_SP  0x2
59905   #define APF_ARP_RD_IP   0x4
59906   #define APF_ARP_RD_SP   0x8
59907   #define APF_ARP_WR_IP   0x10
59908   #define APF_ARP_WR_SP   0x20
59909   #define APF_INADDR_SET  0x100
```

```
59910    #define APF_MORE2WRITE  0x200
59911    #define APF_CLIENTREQ   0x400
59912    #define APF_RARPREQ     0x800
59913    #define APF_CLIENTWRITE 0x1000
59914
59915    #define APS_EMPTY       0
59916    #define APS_STATMASK    0xff
59917    #define         APS_GETADDR     0x1
59918    #define         APS_RARPPROTO   0x2
59919    #define         APS_RARPWRITE   0x4
59920    #define         APS_RARPWAIT    0x8
59921    #define         APS_ARPSTART    0x10
59922    #define         APS_ARPPROTO    0x20
59923    #define         APS_ARPMAIN     0x40
59924    #define         APS_ERROR       0x80
59925    #define APS_SUSPEND     0x400
59926
59927    FORWARD acc_t *arp_getdata ARGS(( int fd, size_t offset,
59928            size_t count, int for_ioctl ));
59929    FORWARD int arp_putdata ARGS(( int fd, size_t offset,
59930            acc_t *data, int for_ioctl ));
59931    FORWARD void arp_main ARGS(( arp_port_t *port ));
59932    FORWARD void arp_timeout ARGS(( int fd, timer_t *timer ));
59933    FORWARD void rarp_timeout ARGS(( int fd, timer_t *timer ));
59934    FORWARD void ipaddr_set ARGS(( arp_port_t *port ));
59935    FORWARD void setup_write ARGS(( arp_port_t *port ));
59936    FORWARD void setup_read ARGS(( arp_port_t *port ));
59937    FORWARD void process_arp_req ARGS(( arp_port_t *port, acc_t *data ));
59938    FORWARD void client_reply ARGS(( arp_port_t *port,
59939            ether_addr_t *ethaddr ));
59940    FORWARD arp_cache_t *find_cache_ent ARGS(( int eth_port, ipaddr_t ipaddr,
59941            int level, arp_cache_t **new_ent ));
59942    FORWARD void rarp_read_setup ARGS(( arp_port_t *port ));
59943    FORWARD void print_arp_cache ARGS(( void ));
59944
59945    PRIVATE arp_port_t arp_port_table[ARP_PORT_NR];
59946    PRIVATE arp_port_t *arp_port;
59947    PRIVATE arp_cache_t arp_cache[ARP_CACHE_NR];
59948
59949    PUBLIC void arp_init()
59950    {
59951            int i;
59952
59953            assert (BUF_S >= sizeof(struct nwio_ethstat));
59954            assert (BUF_S >= sizeof(struct nwio_ethopt));
59955            assert (BUF_S >= sizeof(rarp46_t));
59956            assert (BUF_S >= sizeof(arp46_t));
59957            arp_port_table[0].ap_eth_port= ETH0;
59958
59959            for (i=0, arp_port= arp_port_table; i<ARP_PORT_NR; i++, arp_port++)
59960            {
59961                    arp_port->ap_state= APS_EMPTY;
59962                    arp_port->ap_flags= APF_EMPTY;
59963                    arp_main(arp_port);
59964            }
59965    }

59967    PRIVATE void arp_main(port)
59968    arp_port_t *port;
59969    {
```

```
59970           int result;
59971
59972   #if DEBUG & 256
59973    { where(); printf("in arp_main with status: %d\n", port->ap_state &
59974           APS_STATMASK); }
59975   #endif
59976           switch (port->ap_state & APS_STATMASK)
59977           {
59978           case APS_EMPTY:
59979                   port->ap_rarp_retries= 0;
59980                   port->ap_eth_fd= eth_open(port->ap_eth_port,
59981                           port-arp_port_table, arp_getdata, arp_putdata);
59982
59983                   if (port->ap_eth_fd<0)
59984                   {
59985           printf("arp.c: unable to open ethernet\n");
59986                           return;
59987                   }
59988
59989                   port->ap_state= (port->ap_state &
59990                            ~(APS_STATMASK|APS_SUSPEND)) | APS_GETADDR;
59991
59992                   result= eth_ioctl (port->ap_eth_fd, NWIOGETHSTAT);
59993
59994                   if (result==NW_SUSPEND)
59995                           port->ap_state |= APS_SUSPEND;
59996
59997                   if (result<0)
59998                   {
59999   #if DEBUG
60000    if (result != NW_SUSPEND)
60001    { where(); printf("arp.c: eth_ioctl(..,NWIOGETHSTAT)=%d\n", result); }
60002   #endif
60003                           return;
60004                   }
60005                   if ((port->ap_state & APS_STATMASK) != APS_GETADDR)
60006                           return;
60007                   /* drop through */
60008           case APS_GETADDR:
60009   #if DEBUG & 256
60010    { where(); printf("in arp_main with status: %d\n", port->ap_state & APS_STATMASK); }
60011   #endif
60012                   port->ap_state= (port->ap_state &
60013                           ~(APS_STATMASK|APS_SUSPEND)) | APS_RARPPROTO;
60014
60015                   result= eth_ioctl (port->ap_eth_fd, NWIOSETHOPT);
60016
60017                   if (result==NW_SUSPEND)
60018                           port->ap_state |= APS_SUSPEND;
60019
60020                   if (result<0)
60021                   {
60022   #if DEBUG
60023    if (result != NW_SUSPEND)
60024    { printf("arp.c: eth_ioctl(..,NWIOSETHOPT)=%d\n", result); }
60025   #endif
60026                           return;
60027                   }
60028                   if ((port->ap_state & APS_STATMASK) != APS_RARPPROTO)
60029                           return;
```

```
60030                   /* drop through */
60031           case APS_RARPWAIT:
60032   #if DEBUG & 256
60033    { where(); printf("in arp_main with status: %d\n",
60034           port->ap_state & APS_STATMASK); }
60035   #endif
60036                   if (port->ap_flags & APF_INADDR_SET)
60037                   {
60038                           port->ap_state= (port->ap_state &
60039                                   ~(APS_STATMASK|APS_SUSPEND)) | APS_ARPSTART;
60040                           arp_main(port);
60041                           return;
60042                   }
60043                   /* drop through */
60044           case APS_RARPPROTO:
60045   #if DEBUG & 256
60046    { where(); printf("in arp_main with status: %d\n",
60047           port->ap_state & APS_STATMASK); }
60048   #endif
60049                   rarp_read_setup(port);
60050                   port->ap_state= (port->ap_state &
60051                           ~(APS_STATMASK|APS_SUSPEND)) | APS_RARPWRITE;
60052   #if DEBUG & 256
60053    { where(); printf("doing eth_write\n"); }
60054   #endif
60055                   result= eth_write (port->ap_eth_fd, sizeof(rarp46_t));
60056                   if (result == NW_SUSPEND)
60057                           port->ap_state |= APS_SUSPEND;
60058                   if (result<0)
60059                   {
60060   #if DEBUG & 256
60061    if (result != NW_SUSPEND)
60062    { where();  printf("arp.c: eth_write(..,%d)=%d\n", sizeof(rarp46_t), result); }
60063   #endif
60064                           return;
60065                   }
60066                   if ((port->ap_state & APS_STATMASK) != APS_RARPWRITE)
60067                           return;
60068                   /* drop through */
60069           case APS_RARPWRITE:
60070   #if DEBUG & 256
60071    { where(); printf("in arp_main with status: %d\n",
60072           port->ap_state & APS_STATMASK); }
60073   #endif
60074                   port->ap_state= (port->ap_state &
60075                           ~(APS_STATMASK|APS_SUSPEND)) | APS_RARPWAIT;
60076                   if (port->ap_rarp_retries>=MAX_RARP_RETRIES)
60077                           return;
60078   #if DEBUG & 256
60079    { where(); printf("port->ap_rarp_retries= %d\n",
60080           port->ap_rarp_retries); }
60081   #endif
60082                   port->ap_rarp_retries++;
60083                   clck_timer(&port->ap_timer, get_time() + RARP_TIMEOUT,
60084                           rarp_timeout, port-arp_port_table);
60085                   return;
60086           case APS_ARPSTART:
60087   #if DEBUG & 256
60088    { where(); printf("in arp_main with status: %d\n",
60089           port->ap_state & APS_STATMASK); }
```

```
60090   #endif
60091                   if (port->ap_flags & APF_RARP_RD_IP)
60092                   {
60093                           eth_cancel(port->ap_eth_fd, SR_CANCEL_READ);
60094                           port->ap_flags &= ~(APF_RARP_RD_IP|APF_RARP_RD_SP);
60095                   }
60096                   port->ap_state= (port->ap_state &
60097                           ~(APS_STATMASK|APS_SUSPEND)) | APS_ARPPROTO;
60098
60099                   {
60100                           arp_cache_t *cache;
60101                           int i;
60102
60103                           cache= arp_cache;
60104                           for (i=0; i<ARP_CACHE_NR; i++, cache++)
60105                           {
60106                                   cache->ac_flags= ACF_EMPTY;
60107                                   cache->ac_expire= 0;
60108                                   cache->ac_lastuse= 0;
60109                           }
60110
60111                           cache= arp_cache;
60112                           for (i=0; i<ARP_CACHE1_NR; i++, cache++)
60113                                   cache->ac_type= 1;
60114                           for (i=0; i<ARP_CACHE2_NR; i++, cache++)
60115                                   cache->ac_type= 2;
60116                           for (i=0; i<ARP_CACHE3_NR; i++, cache++)
60117                                   cache->ac_type= 3;
60118                   }
60119                   result= eth_ioctl (port->ap_eth_fd, NWIOSETHOPT);
60120
60121                   if (result==NW_SUSPEND)
60122                           port->ap_state |= APS_SUSPEND;
60123
60124                   if (result<0)
60125                   {
60126   #if DEBUG
60127    { where(); printf("arp.c: /* arp */ eth_ioctl(..,NWIOSETHOPT)=%d\n",
60128           result); }
60129   #endif
60130                           return;
60131                   }
60132                   if ((port->ap_state & APS_STATMASK) != APS_ARPPROTO)
60133                           return;
60134                   /* drop through */
60135           case APS_ARPPROTO:
60136   #if DEBUG & 256
60137    { where(); printf("in arp_main with status: %d\n",
60138           port->ap_state & APS_STATMASK); }
60139   #endif
60140                   port->ap_state= (port->ap_state &
60141                           ~(APS_STATMASK|APS_SUSPEND)) | APS_ARPMAIN;
60142                   if (port->ap_flags & APF_MORE2WRITE)
60143                           setup_write(port);
60144                   setup_read(port);
60145                   return;
60146           default:
60147                   ip_panic((
60148                    "arp_main(&arp_port_table[%d]) called but ap_state=0x%x\n",
60149                           port-arp_port_table, port->ap_state ));
```

```
60150           }
60151   }

60153   PRIVATE acc_t *arp_getdata (fd, offset, count, for_ioctl)
60154   int fd;
60155   size_t offset;
60156   size_t count;
60157   int for_ioctl;
60158   {
60159           arp_port_t *port;
60160           rarp46_t *rarp;
60161           arp46_t *arp;
60162           acc_t *data;
60163           int result;
60164
60165   #if DEBUG & 256
60166    { where(); printf("arp_getdata (fd= %d, offset= %d, count= %d)\n", fd,
60167           offset, count); }
60168   #endif
60169           port= &arp_port_table[fd];
60170
60171           switch (port->ap_state & APS_STATMASK)
60172           {
60173           case APS_RARPPROTO:
60174                   if (!count)
60175                   {
60176                           result= (int)offset;
60177                           if (result<0)
60178                           {
60179                                   port->ap_state= (port->ap_state &
60180                                           ~(APS_STATMASK|APS_SUSPEND))|
60181                                           APS_ERROR;
60182                                   break;
60183                           }
60184                           if (port->ap_state & APS_SUSPEND)
60185                                   arp_main(port);
60186                           return NW_OK;
60187                   }
60188                   assert ((!offset) && (count == sizeof(struct nwio_ethopt)));
60189                   {
60190                           struct nwio_ethopt *ethopt;
60191                           acc_t *acc;
60192
60193                           acc= bf_memreq(sizeof(*ethopt));
60194                           ethopt= (struct nwio_ethopt *)ptr2acc_data(acc);
60195                           ethopt->nweo_flags= NWEO_COPY|NWEO_TYPESPEC;
60196                           ethopt->nweo_type= htons(ETH_RARP_PROTO);
60197                           return acc;
60198                   }
60199           case APS_RARPWRITE:
60200                   if (!count)
60201                   {
60202                           result= (int)offset;
60203                           if (result<0)
60204                           {
60205   #if DEBUG
60206    if (result != NW_SUSPEND)
60207    { where(); printf("arp.c: write error on port %d: error %d\n", fd, result); }
60208   #endif
60209                                   port->ap_state= (port->ap_state &
```

```
60210                                   ~(APS_STATMASK|APS_SUSPEND))|
60211                                   APS_ERROR;
60212                                   break;
60213                           }
60214                           if (port->ap_state & APS_SUSPEND)
60215                                   arp_main(port);
60216                           return NW_OK;
60217                   }
60218                   assert (offset+count <= sizeof(rarp46_t));
60219                   data= bf_memreq(sizeof(rarp46_t));
60220                   rarp= (rarp46_t *)ptr2acc_data(data);
60221                   data->acc_offset += offset;
60222                   data->acc_length= count;
60223                   rarp->a46_dstaddr.ea_addr[0]= 0xff;
60224                   rarp->a46_dstaddr.ea_addr[1]= 0xff;
60225                   rarp->a46_dstaddr.ea_addr[2]= 0xff;
60226                   rarp->a46_dstaddr.ea_addr[3]= 0xff;
60227                   rarp->a46_dstaddr.ea_addr[4]= 0xff;
60228                   rarp->a46_dstaddr.ea_addr[5]= 0xff;
60229                   rarp->a46_hdr= htons(ARP_ETHERNET);
60230                   rarp->a46_pro= htons(ETH_IP_PROTO);
60231                   rarp->a46_hln= 6;
60232                   rarp->a46_pln= 4;
60233                   rarp->a46_op= htons(RARP_REQUEST);
60234                   rarp->a46_sha= port->ap_ethaddr;
60235                   rarp->a46_tha= port->ap_ethaddr;
60236                   return data;
60237           case APS_ARPPROTO:
60238                   if (!count)
60239                   {
60240                           result= (int)offset;
60241                           if (result<0)
60242                           {
60243                                   port->ap_state= (port->ap_state &
60244                                           ~(APS_STATMASK|APS_SUSPEND))|
60245                                           APS_ERROR;
60246                                   break;
60247                           }
60248                           if (port->ap_state & APS_SUSPEND)
60249                                   arp_main(port);
60250                           return NW_OK;
60251                   }
60252                   assert ((!offset) && (count == sizeof(struct nwio_ethopt)));
60253                   {
60254                           struct nwio_ethopt *ethopt;
60255                           acc_t *acc;
60256
60257                           acc= bf_memreq(sizeof(*ethopt));
60258                           ethopt= (struct nwio_ethopt *)ptr2acc_data(acc);
60259                           ethopt->nweo_flags= NWEO_COPY|NWEO_EN_BROAD|
60260                                   NWEO_TYPESPEC;
60261                           ethopt->nweo_type= htons(ETH_ARP_PROTO);
60262                           return acc;
60263                   }
60264           case APS_ARPMAIN:
60265                   assert (port->ap_flags & APF_ARP_WR_IP);
60266                   if (!count)
60267                   {
60268                           result= (int)offset;
60269                           if (result<0)
```

```
60270                           {
60271   #if DEBUG
60272    if (result != NW_SUSPEND)
60273    { where(); printf("arp.c: write error on port %d: error %d\n", fd, result); }
60274   #endif
60275                                   port->ap_state= (port->ap_state &
60276                                           ~(APS_STATMASK|APS_SUSPEND))|
60277                                           APS_ERROR;
60278                                   break;
60279                           }
60280                           port->ap_flags &= ~APF_ARP_WR_IP;
60281                           if (port->ap_flags & APF_ARP_WR_SP)
60282                                   setup_write(port);
60283                           return NW_OK;
60284                   }
60285                   assert (offset+count <= sizeof(arp46_t));
60286                   data= bf_memreq(sizeof(arp46_t));
60287                   arp= (arp46_t *)ptr2acc_data(data);
60288                   data->acc_offset += offset;
60289                   data->acc_length= count;
60290                   if (port->ap_write_code == ARP_REPLY)
60291                           arp->a46_dstaddr= port->ap_write_ethaddr;
60292                   else
60293                   {
60294                           arp->a46_dstaddr.ea_addr[0]= 0xff;
60295                           arp->a46_dstaddr.ea_addr[1]= 0xff;
60296                           arp->a46_dstaddr.ea_addr[2]= 0xff;
60297                           arp->a46_dstaddr.ea_addr[3]= 0xff;
60298                           arp->a46_dstaddr.ea_addr[4]= 0xff;
60299                           arp->a46_dstaddr.ea_addr[5]= 0xff;
60300                   }
60301                   arp->a46_hdr= htons(ARP_ETHERNET);
60302                   arp->a46_pro= htons(ETH_IP_PROTO);
60303                   arp->a46_hln= 6;
60304                   arp->a46_pln= 4;
60305                   arp->a46_op= htons(port->ap_write_code);
60306                   arp->a46_sha= port->ap_ethaddr;
60307                   memcpy (arp->a46_spa, &port->ap_ipaddr, sizeof(ipaddr_t));
60308                   arp->a46_tha= port->ap_write_ethaddr;
60309                   memcpy (arp->a46_tpa, &port->ap_write_ipaddr, sizeof(ipaddr_t));
60310                   return data;
60311           default:
60312                   printf("arp_getdata(%d, 0x%d, 0x%d) called but ap_state=0x%x\n",
60313                           fd, offset, count, port->ap_state);
60314                   break;
60315           }
60316           return 0;
60317   }

60319   PRIVATE int arp_putdata (fd, offset, data, for_ioctl)
60320   int fd;
60321   size_t offset;
60322   acc_t *data;
60323   int for_ioctl;
60324   {
60325           arp_port_t *port;
60326           int result;
60327           struct nwio_ethstat *ethstat;
60328           rarp46_t *rarp;
60329
```

```
60330   #if DEBUG & 256
60331    { where(); printf("arp_putdata (fd= %d, offset= %d, data= 0x%x)\n",
60332           fd, offset, data); }
60333   #endif
60334           port= &arp_port_table[fd];
60335
60336           if (port->ap_flags & APF_ARP_RD_IP)
60337           {
60338                   if (!data)
60339                   {
60340                           result= (int)offset;
60341                           if (result<0)
60342                           {
60343   #if DEBUG
60344    if (result != NW_SUSPEND)
60345    { where(); printf("arp.c: read error on port %d: error %d\n", fd, result); }
60346   #endif
60347                                   return NW_OK;
60348                           }
60349                           if (port->ap_flags & APF_ARP_RD_SP)
60350                           {
60351                                   port->ap_flags &= ~(APF_ARP_RD_IP|
60352                                           APF_ARP_RD_SP);
60353                                   setup_read(port);
60354                           }
60355                           else
60356                                   port->ap_flags &= ~(APF_ARP_RD_IP|
60357                                           APF_ARP_RD_SP);
60358                           return NW_OK;
60359                   }
60360                   assert (!offset);
60361                   /* Warning: the above assertion is illegal; puts and gets of
60362                      data can be brokenup in any piece the server likes. However
60363                      we assume that the server is eth.c and it transfers only
60364                      whole packets. */
60365                   data= bf_packIffLess(data, sizeof(arp46_t));
60366                   if (data->acc_length >= sizeof(arp46_t))
60367                           process_arp_req(port,data);
60368                   bf_afree(data);
60369                   return NW_OK;
60370           }
60371           if (port->ap_flags & APF_RARP_RD_IP)
60372           {
60373                   if (!data)
60374                   {
60375                           result= (int)offset;
60376                           if (result<0)
60377                           {
60378                                   if (result != EINTR)
60379                                           ip_warning((
60380                                   "arp.c: read error on port %d: error %d\n",
60381                                                                   fd, result ));
60382                                   return NW_OK;
60383                           }
60384                           port->ap_flags &= ~APF_RARP_RD_IP;
60385                           if (port->ap_flags & APF_INADDR_SET)
60386                                   ipaddr_set(port);
60387                           else if (port->ap_flags & APF_RARP_RD_SP)
60388                                   rarp_read_setup(port);
60389                           return NW_OK;
```

```
60390                   }
60391                   assert (!offset);
60392                   /* Warning: the above assertion is illegal; puts and gets of
60393                      data can be brokenup in any piece the server likes. However
60394                      we assume that the server is eth.c and it transfers only
60395                      whole packets. */
60396                   data= bf_packIffLess(data, sizeof(rarp46_t));
60397                   rarp= (rarp46_t *)ptr2acc_data(data);
60398                   if ((data->acc_length >= sizeof(rarp46_t)) &&
60399                           (rarp->a46_hdr == htons(ARP_ETHERNET)) &&
60400                           (rarp->a46_pro == htons(ETH_IP_PROTO)) &&
60401                           (rarp->a46_hln == 6) &&
60402                           (rarp->a46_pln == 4) &&
60403                           (rarp->a46_op == htons(RARP_REPLY)) &&
60404                           (rarp->a46_tha.ea_addr[5] ==
60405                                   port->ap_ethaddr.ea_addr[5]) &&
60406                           (rarp->a46_tha.ea_addr[4] ==
60407                                   port->ap_ethaddr.ea_addr[4]) &&
60408                           (rarp->a46_tha.ea_addr[3] ==
60409                                   port->ap_ethaddr.ea_addr[3]) &&
60410                           (rarp->a46_tha.ea_addr[2] ==
60411                                   port->ap_ethaddr.ea_addr[2]) &&
60412                           (rarp->a46_tha.ea_addr[1] ==
60413                                   port->ap_ethaddr.ea_addr[1]) &&
60414                           (rarp->a46_tha.ea_addr[0] ==
60415                                   port->ap_ethaddr.ea_addr[0]) &&
60416                           !(port->ap_flags & APF_INADDR_SET))
60417                   {
60418                           memcpy (&port->ap_ipaddr, rarp->a46_tpa,
60419                                   sizeof(ipaddr_t));
60420                           port->ap_flags |= APF_INADDR_SET;
60421   #if DEBUG & 256
60422    { unsigned char *a; where();  a=(unsigned char *)&port->ap_ipaddr;
60423           printf("arp.c: got ip address: %d.%d.%d.%d\n",
60424           a[0], a[1], a[2], a[3]); }
60425   #endif
60426                   }
60427                   bf_afree(data);
60428                   return NW_OK;
60429           }
60430           switch (port->ap_state & APS_STATMASK)
60431           {
60432           case APS_GETADDR:
60433                   if (!data)
60434                   {
60435                           result= (int)offset;
60436                           if (result<0)
60437                           {
60438                                   port->ap_state= (port->ap_state &
60439                                           ~(APS_STATMASK|APS_SUSPEND))|
60440                                           APS_ERROR;
60441                                   break;
60442                           }
60443                           if (port->ap_state & APS_SUSPEND)
60444                                   arp_main(port);
60445                           return NW_OK;
60446                   }
60447                   compare (bf_bufsize(data), ==, sizeof(*ethstat));
60448                   data= bf_packIffLess(data, sizeof(*ethstat));
60449                   compare (data->acc_length, ==, sizeof(*ethstat));
```

```
60450                   ethstat= (struct nwio_ethstat *)ptr2acc_data(data);
60451                   port->ap_ethaddr= ethstat->nwes_addr;
60452                   bf_afree(data);
60453                   return NW_OK;
60454           default:
60455                   printf("arp_putdata(%d, 0x%d, 0x%lx) called but ap_state=0x%x\n",
60456                           fd, offset, (unsigned long)data, port->ap_state);
60457                   break;
60458           }
60459           return EGENERIC;
60460   }

60462   PRIVATE void rarp_timeout (fd, timer)
60463   int fd;
60464   timer_t *timer;
60465   {
60466           arp_port_t *port;
60467
60468   #if DEBUG & 256
60469    { where(); printf("in rarp_timeout()\n"); }
60470   #endif
60471           port= &arp_port_table[fd];
60472
60473           assert (timer == &port->ap_timer);
60474
60475           arp_main(port);
60476   }

60478   PRIVATE void ipaddr_set (port)
60479   arp_port_t *port;
60480   {
60481           if (port->ap_flags & APF_RARPREQ)
60482           {
60483                   port->ap_flags &= ~APF_RARPREQ;
60484                   (*port->ap_rarp_func)(port->ap_rarp_ref, port->ap_ipaddr);
60485           }
60486           if (port->ap_state & APS_RARPWAIT)
60487           {
60488                   clck_untimer(&port->ap_timer);
60489                   port->ap_state= (port->ap_state &
60490                           ~(APS_STATMASK|APS_SUSPEND)) | APS_ARPSTART;
60491                   arp_main(port);
60492           }
60493   }

60495   PRIVATE void setup_read(port)
60496   arp_port_t *port;
60497   {
60498           int result;
60499
60500           while (!(port->ap_flags & APF_ARP_RD_IP))
60501           {
60502                   port->ap_flags |= APF_ARP_RD_IP;
60503                   result= eth_read (port->ap_eth_fd, ETH_MAX_PACK_SIZE);
60504                   if (result == NW_SUSPEND)
60505                           port->ap_flags |= APF_ARP_RD_SP;
60506                   if (result<0)
60507                   {
60508   #if DEBUG
60509    if (result != NW_SUSPEND)
```

```
60510     { where(); printf("arp.c: eth_read(..,%d)=%d\n", ETH_MAX_PACK_SIZE, result); }
60511     #endif
60512                             return;
60513                     }
60514             }
60515     }

60517     PRIVATE void setup_write(port)
60518     arp_port_t *port;
60519     {
60520             int i, result;
60521
60522             while (port->ap_flags & APF_MORE2WRITE)
60523             {
60524                     if (port->ap_flags & APF_CLIENTWRITE)
60525                     {
60526                             port->ap_flags &= ~APF_CLIENTWRITE;
60527                             port->ap_write_ipaddr= port->ap_req_ipaddr;
60528                             port->ap_write_code= ARP_REQUEST;
60529                             clck_timer(&port->ap_timer, get_time() + ARP_TIMEOUT,
60530                                     arp_timeout, port-arp_port_table);
60531                     }
60532                     else
60533                     {
60534                             arp_cache_t *cache;
60535
60536                             cache= arp_cache;
60537                             for (i=0; i<ARP_CACHE_NR; i++, cache++)
60538                                     if ((cache->ac_flags & ACF_NETREQ) &&
60539                                             cache->ac_eth_port ==
60540                                             port->ap_eth_port)
60541                                     {
60542                                             cache->ac_flags &= ~ACF_NETREQ;
60543                                             port->ap_write_ethaddr= cache->
60544                                                     ac_ethaddr;
60545                                             port->ap_write_ipaddr= cache->
60546                                                     ac_ipaddr;
60547                                             port->ap_write_code= ARP_REPLY;
60548                                             break;
60549                                     }
60550                             if (i>=ARP_CACHE_NR)
60551                             {
60552                                     port->ap_flags &= ~APF_MORE2WRITE;
60553                                     break;
60554                             }
60555                     }
60556                     port->ap_flags= (port->ap_flags & ~APF_ARP_WR_SP) |
60557                             APF_ARP_WR_IP;
60558     #if DEBUG & 256
60559      { where(); printf("doing eth_write\n"); }
60560     #endif
60561                     result= eth_write(port->ap_eth_fd, sizeof(arp46_t));
60562                     if (result == NW_SUSPEND)
60563                             port->ap_flags |= APF_ARP_WR_SP;
60564                     if (result<0)
60565                     {
60566     #if DEBUG
60567      if (result != NW_SUSPEND)
60568      { where(); printf("arp.c: eth_write(..,%d)=%d\n", sizeof(rarp46_t), result); }
60569     #endif
```

```
60570                           return;
60571                   }
60572           }
60573   }

60575   PRIVATE void process_arp_req (port, data)
60576   arp_port_t *port;
60577   acc_t *data;
60578   {
60579           arp46_t *arp;
60580           arp_cache_t *prim, *sec;
60581           int level;
60582           time_t curr_tim;
60583           ipaddr_t spa, tpa;
60584
60585   #if DEBUG & 256
60586    { where(); printf("process_arp_req(...)\n"); }
60587   #endif
60588   #if DEBUG & 256
60589    { where(); print_arp_cache(); }
60590   #endif
60591           arp= (arp46_t *)ptr2acc_data(data);
60592           memcpy(&spa, arp->a46_spa, sizeof(ipaddr_t));
60593           memcpy(&tpa, arp->a46_tpa, sizeof(ipaddr_t));
60594
60595   #if DEBUG & 256
60596    {
60597     if (arp->a46_hdr == htons(ARP_ETHERNET))
60598     { where(); printf("arp.c: a46_hdr OK\n"); }
60599     if (arp->a46_hln == 6)
60600     { where(); printf("arp.c: a46_hln OK\n"); }
60601     if (arp->a46_pro == htons(ETH_IP_PROTO))
60602     { where(); printf("arp.c: a46_pro OK\n"); }
60603     if (arp->a46_pln == 4)
60604     { where(); printf("arp.c: a46_pln OK\n"); }
60605    }
60606   #endif
60607           if (arp->a46_hdr != htons(ARP_ETHERNET) ||
60608                   arp->a46_hln != 6 ||
60609                   arp->a46_pro != htons(ETH_IP_PROTO) ||
60610                   arp->a46_pln != 4)
60611                   return;
60612   #if DEBUG & 256
60613    { where(); printf("arp.c: a46_tpa= 0x%lx, ap_ipaddr= 0x%lx\n",
60614           arp->a46_tpa, port->ap_ipaddr); }
60615   #endif
60616           if ((port->ap_flags & APF_CLIENTREQ) && (spa == port->ap_req_ipaddr))
60617                   level= ARP_TYPE3;
60618           else if (arp->a46_op == htons(ARP_REQUEST) && (tpa ==
60619                   port->ap_ipaddr))
60620                   level= ARP_TYPE2;
60621           else
60622                   level= ARP_TYPE1;
60623
60624   #if DEBUG & 256
60625    { where(); printf("arp.c: level= %d\n", level); }
60626   #endif
60627           prim= find_cache_ent(port->ap_eth_port, spa, level, &sec);
60628           if (!prim)
60629           {
```

```
60630                   prim= sec;
60631                   prim->ac_flags= ACF_EMPTY;
60632                   prim->ac_ipaddr= spa;
60633                   prim->ac_eth_port= port->ap_eth_port;
60634           }
60635           else if (prim->ac_type < level)
60636           {
60637                   sec->ac_type= prim->ac_type;
60638                   prim->ac_type= level;
60639           }
60640           prim->ac_ethaddr= arp->a46_sha;
60641           curr_tim= get_time();
60642           prim->ac_expire= curr_tim+ ARP_EXP_TIME;
60643           if (curr_tim > prim->ac_lastuse)
60644                   prim->ac_lastuse= curr_tim;
60645           prim->ac_flags &= ~ACF_NOTRCH;
60646           if (level== ARP_TYPE2)
60647           {
60648                   prim->ac_flags |= ACF_NETREQ;
60649                   port->ap_flags |= APF_MORE2WRITE;
60650                   if (!(port->ap_flags & APF_ARP_WR_IP))
60651                           setup_write(port);
60652           } else if (level== ARP_TYPE3)
60653           {
60654                   prim->ac_lastuse= curr_tim + ARP_INUSE_OFFSET;
60655                   client_reply(port, &arp->a46_sha);
60656           }
60657   #if DEBUG & 256
60658    { where(); print_arp_cache(); }
60659   #endif
60660   }

60662   PRIVATE void client_reply (port, ethaddr)
60663   arp_port_t *port;
60664   ether_addr_t *ethaddr;
60665   {
60666           port->ap_flags &= ~(APF_CLIENTREQ|APF_CLIENTWRITE);
60667           clck_untimer(&port->ap_timer);
60668           (*port->ap_req_func)(port->ap_req_ref, ethaddr);
60669   }

60671   PRIVATE arp_cache_t *find_cache_ent (eth_port, ipaddr, level, new_ent)
60672   int eth_port;
60673   ipaddr_t ipaddr;
60674   int level;
60675   arp_cache_t **new_ent;
60676   {
60677           arp_cache_t *cache, *prim, *sec;
60678           int i;
60679
60680           cache= arp_cache;
60681           prim= 0;
60682           sec= 0;
60683           for (i=0; i<ARP_CACHE_NR; i++, cache++)
60684           {
60685                   if (cache->ac_eth_port == eth_port &&
60686                           cache->ac_ipaddr == ipaddr)
60687                           prim= cache;
60688                   if (cache->ac_type == level && (!sec || cache->ac_lastuse <
60689                           sec->ac_lastuse))
```

```
60690                           sec= cache;
60691           }
60692           assert(sec);
60693           *new_ent= sec;
60694           return prim;
60695   }

60697   PRIVATE void rarp_read_setup (port)
60698   arp_port_t *port;
60699   {
60700           int result;
60701
60702           while (!(port->ap_flags & (APF_RARP_RD_IP|APF_INADDR_SET)))
60703           {
60704                   port->ap_flags= (port->ap_flags & ~ APF_RARP_RD_SP) |
60705                           APF_RARP_RD_IP;
60706                   result= eth_read (port->ap_eth_fd, ETH_MAX_PACK_SIZE);
60707                   if (result == NW_SUSPEND)
60708                           port->ap_flags |= APF_RARP_RD_SP;
60709                   if (result<0)
60710                   {
60711   #if DEBUG
60712    if (result != NW_SUSPEND)
60713    { where(); printf("arp.c: eth_read(..,%d)=%d\n", ETH_MAX_PACK_SIZE, result); }
60714   #endif
60715                           return;
60716                   }
60717                   if ((port->ap_state & APS_STATMASK) != APS_RARPPROTO)
60718                           return;
60719           }
60720   }

60722   PUBLIC int rarp_req(eth_port, ref, func)
60723   int eth_port;
60724   int ref;
60725   rarp_func_t func;
60726   {
60727           arp_port_t *port;
60728           int i;
60729
60730           port= arp_port_table;
60731           for (i=0; i<ARP_PORT_NR; i++, port++)
60732                   if (port->ap_eth_port == eth_port)
60733                           break;
60734           if (i>=ARP_PORT_NR)
60735                   return EGENERIC;
60736           if (port->ap_flags & APF_INADDR_SET)
60737           {
60738                   (*func)(ref, port->ap_ipaddr);
60739                   return NW_OK;
60740           }
60741           port->ap_flags |= APF_RARPREQ;
60742           port->ap_rarp_ref= ref;
60743           port->ap_rarp_func= func;
60744           return NW_SUSPEND;
60745   }

60747   PUBLIC void set_ipaddr (eth_port, ipaddr)
60748   int eth_port;
60749   ipaddr_t ipaddr;
```

```
60750   {
60751           arp_port_t *port;
60752           int i;
60753
60754           port= arp_port_table;
60755           for (i=0; i<ARP_PORT_NR; i++, port++)
60756                   if (port->ap_eth_port == eth_port)
60757                           break;
60758           assert (i < ARP_PORT_NR);
60759           port->ap_ipaddr= ipaddr;
60760           port->ap_flags |= APF_INADDR_SET;
60761           ipaddr_set(port);
60762   }

60764   PUBLIC int arp_ip_eth (eth_port, ref, ipaddr, func)
60765   int eth_port;
60766   int ref;
60767   ipaddr_t ipaddr;
60768   arp_req_func_t func;
60769   {
60770           arp_port_t *port;
60771           int i;
60772           arp_cache_t *prim, *sec;
60773
60774   #if DEBUG & 256
60775    { where(); printf("sending arp_req for: "); writeIpAddr(ipaddr);
60776           printf("\n"); }
60777   #endif
60778           port= arp_port_table;
60779           for (i=0; i<ARP_PORT_NR; i++, port++)
60780                   if (port->ap_eth_port == eth_port)
60781                           break;
60782           if (i>=ARP_PORT_NR)
60783                   return EGENERIC;
60784           if ((port->ap_state & APS_STATMASK) != APS_ARPMAIN)
60785           {
60786                   port->ap_flags |= APF_CLIENTREQ|APF_MORE2WRITE |
60787                           APF_CLIENTWRITE;
60788                   port->ap_req_func= func;
60789                   port->ap_req_ref= ref;
60790                   port->ap_req_ipaddr= ipaddr;
60791                   port->ap_req_count= 0;
60792                   return NW_SUSPEND;
60793           }
60794
60795           prim= find_cache_ent (eth_port, ipaddr, ARP_TYPE3, &sec);
60796           if (prim)
60797           {
60798                   if (prim->ac_type < ARP_TYPE3)
60799                   {
60800                           sec->ac_type= prim->ac_type;
60801                           prim->ac_type= ARP_TYPE3;
60802                   }
60803                   if (prim->ac_expire < get_time())
60804                           prim= 0;
60805           }
60806           if (!prim)
60807           {
60808                   port->ap_flags |= APF_CLIENTREQ|APF_MORE2WRITE|APF_CLIENTWRITE;
60809                   port->ap_req_func= func;
```

```
60810                   port->ap_req_ref= ref;
60811                   port->ap_req_ipaddr= ipaddr;
60812                   port->ap_req_count= 0;
60813                   if (!(port->ap_flags & APF_ARP_WR_IP))
60814                           setup_write(port);
60815                   return NW_SUSPEND;
60816           }
60817           prim->ac_lastuse= get_time();
60818           if (prim->ac_flags & ACF_NOTRCH)
60819                   return EDSTNOTRCH;
60820           else
60821           {
60822                   client_reply (port, &prim->ac_ethaddr);
60823                   return NW_OK;
60824           }
60825   }

60827   PUBLIC int arp_ip_eth_nonbl (eth_port, ipaddr, ethaddr)
60828   int eth_port;
60829   ipaddr_t ipaddr;
60830   ether_addr_t *ethaddr;
60831   {
60832           arp_port_t *port;
60833           int i;
60834           arp_cache_t *prim, *sec;
60835
60836   #if DEBUG & 256
60837    { where(); printf("got a arp_ip_eth_nonbl(%d, ", eth_port);
60838           writeIpAddr(ipaddr); printf(", ...)\n");  }
60839   #endif
60840           port= arp_port_table;
60841           for (i=0; i<ARP_PORT_NR; i++, port++)
60842                   if (port->ap_eth_port == eth_port)
60843                           break;
60844           if (i>=ARP_PORT_NR)
60845                   return EGENERIC;
60846           if ((port->ap_state & APS_STATMASK) != APS_ARPMAIN)
60847           {
60848   #if DEBUG
60849    { where(); printf("replying NW_SUSPEND\n"); }
60850   #endif
60851                   return NW_SUSPEND;
60852           }
60853
60854           prim= find_cache_ent (eth_port, ipaddr, ARP_TYPE3, &sec);
60855           if (prim)
60856           {
60857                   if (prim->ac_type < ARP_TYPE3)
60858                   {
60859                           sec->ac_type= prim->ac_type;
60860                           prim->ac_type= ARP_TYPE3;
60861                   }
60862                   if (prim->ac_expire < get_time())
60863                           prim= 0;
60864           }
60865           if (!prim)
60866           {
60867   #if DEBUG & 256
60868    { where(); printf("replying NW_SUSPEND\n"); }
60869   #endif
```

```
60870                   return NW_SUSPEND;
60871           }
60872
60873           if (prim->ac_flags & ACF_NOTRCH)
60874           {
60875   #if DEBUG
60876    { where(); printf("replying EDSTNOTRCH\n"); }
60877   #endif
60878                   return EDSTNOTRCH;
60879           }
60880           else
60881           {
60882                   prim->ac_lastuse= get_time();
60883                   if (ethaddr)
60884                           *ethaddr= prim->ac_ethaddr;
60885   #if DEBUG & 256
60886    { where(); printf("replying NW_OK (\n"); writeEtherAddr(&prim->ac_ethaddr);
60887           printf(")\n"); }
60888   #endif
60889                   return NW_OK;
60890           }
60891   }

60893   PRIVATE void arp_timeout (fd, timer)
60894   int fd;
60895   timer_t *timer;
60896   {
60897           arp_port_t *port;
60898           arp_cache_t *prim, *sec;
60899           int level;
60900           time_t curr_tim;
60901
60902           port= &arp_port_table[fd];
60903
60904           assert (timer == &port->ap_timer);
60905
60906           if (++port->ap_req_count < MAX_ARP_RETRIES)
60907           {
60908                   port->ap_flags |= APF_CLIENTWRITE|APF_MORE2WRITE;
60909                   if (!(port->ap_flags & APF_ARP_WR_IP))
60910                           setup_write(port);
60911           }
60912           else
60913           {
60914                   level= ARP_TYPE3;
60915                   prim= find_cache_ent(port->ap_eth_port, port->ap_req_ipaddr,
60916                           level, &sec);
60917                   if (!prim)
60918                   {
60919                           prim= sec;
60920                           prim->ac_flags= ACF_EMPTY;
60921                           prim->ac_ipaddr= port->ap_req_ipaddr;
60922                   }
60923                   else if (prim->ac_type < level)
60924                   {
60925                           sec->ac_type= prim->ac_type;
60926                           prim->ac_type= level;
60927                   }
60928                   curr_tim= get_time();
60929                   prim->ac_expire= curr_tim+ ARP_NOTRCH_EXP_TIME;
```

```
60930                   prim->ac_lastuse= curr_tim + ARP_INUSE_OFFSET;
60931                   prim->ac_flags |= ACF_NOTRCH;
60932
60933                   (*port->ap_req_func)(port->ap_req_ref,
60934                           (ether_addr_t *)0);
60935           }
60936   }

60938   PRIVATE void print_arp_cache()
60939   {
60940           int i;
60941           arp_cache_t *ci;
60942
60943           for (i=0, ci= arp_cache; i< ARP_CACHE_NR; i++, ci++)
60944           {
60945                   if (!ci->ac_expire)
60946                           continue;
60947                   printf("%d %d ", ci->ac_flags, ci->ac_type);
60948                   writeEtherAddr(&ci->ac_ethaddr);
60949                   printf(" ");
60950                   writeIpAddr(ci->ac_ipaddr);
60951                   printf(" %d %ld %ld\n", ci->ac_eth_port, ci->ac_expire,
60952                           ci->ac_lastuse);
60953           }
60954   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/eth.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
61000   /*
61001   eth.c
61002   */
61003
61004   #include "inet.h"
61005   #include "buf.h"
61006   #include "clock.h"
61007   #include "osdep_eth.h"
61008
61009   #include "assert.h"
61010   #include "buf.h"
61011   #include "eth.h"
61012   #include "eth_int.h"
61013   #include "io.h"
61014   #include "sr.h"
61015   #include "type.h"
61016
61017   INIT_PANIC();
61018
61019   #define ETH_FD_NR       32
61020   #define EXPIRE_TIME     60*HZ   /* seconds */
61021
61022   typedef struct eth_fd
61023   {
61024           int ef_flags;
```

```
61025           eth_port_t *ef_port;
61026           int ef_srfd;
61027           get_userdata_t ef_get_userdata;
61028           put_userdata_t ef_put_userdata;
61029           nwio_ethopt_t ef_ethopt;
61030           size_t ef_write_count;
61031           acc_t *ef_rd_buf;
61032           acc_t *ef_rd_tail;
61033           time_t ef_exp_tim;
61034           int ef_pack_stat;
61035   } eth_fd_t;
61036
61037   #define EFF_FLAGS       0xf
61038   #       define EFF_EMPTY        0x0
61039   #       define EFF_INUSE        0x1
61040   #       define EFF_BUSY         0x6
61041   #               define  EFF_READ_IP     0x2
61042   #               define  EFF_WRITE_IP    0x4
61043   #       define EFF_OPTSET       0x8
61044
61045   FORWARD int eth_checkopt ARGS(( eth_fd_t *eth_fd ));
61046   FORWARD void eth_buffree ARGS(( int priority, size_t reqsize ));
61047   FORWARD int ok_for_me ARGS(( eth_fd_t *fd, acc_t *pack ));
61048   FORWARD int packet2user ARGS(( eth_fd_t *fd ));
61049   FORWARD void reply_thr_get ARGS(( eth_fd_t *eth_fd,
61050           size_t result, int for_ioctl ));
61051   FORWARD void reply_thr_put ARGS(( eth_fd_t *eth_fd,
61052           size_t result, int for_ioctl ));
61053   FORWARD void restart_write_fd ARGS(( eth_fd_t *eth_fd ));
61054
61055   PUBLIC eth_port_t eth_port_table[ETH_PORT_NR];
61056
61057   PRIVATE eth_fd_t eth_fd_table[ETH_FD_NR];
61058   /* PRIVATE message mess, repl_mess; */
61059
61060   PUBLIC void eth_init()
61061   {
61062           int i;
61063
61064           assert (BUF_S >= sizeof(nwio_ethopt_t));
61065           assert (BUF_S >= ETH_HDR_SIZE); /* these are in fact static assertions,
61066                                              thus a good compiler doesn't
61067                                              generate any code for this */
61068
61069
61070           for (i=0; i<ETH_FD_NR; i++)
61071                   eth_fd_table[i].ef_flags= EFF_EMPTY;
61072           for (i=0; i<ETH_PORT_NR; i++)
61073                   eth_port_table[i].etp_flags= EFF_EMPTY;
61074
61075           bf_logon(eth_buffree);
61076
61077           eth_init0();
61078           /* eth_init1(); etc */
61079   }

61081   PUBLIC int eth_open(port, srfd, get_userdata, put_userdata)
61082   int port, srfd;
61083   get_userdata_t get_userdata;
61084   put_userdata_t put_userdata;
```

```
61085   {
61086           int i;
61087           eth_port_t *eth_port;
61088           eth_fd_t *eth_fd;
61089
61090   #if DEBUG & 256
61091    { where(); printf("eth_open (%d, ...)\n", port); }
61092   #endif
61093           eth_port= &eth_port_table[port];
61094           if (!(eth_port->etp_flags & EPF_ENABLED))
61095                   return EGENERIC;
61096
61097           for (i=0; i<ETH_FD_NR && (eth_fd_table[i].ef_flags & EFF_INUSE);
61098                   i++);
61099
61100           if (i>=ETH_FD_NR)
61101           {
61102   #if DEBUG
61103    { where(); printf("out of fds\n"); }
61104   #endif
61105                   return EOUTOFBUFS;
61106           }
61107
61108           eth_fd= &eth_fd_table[i];
61109
61110           eth_fd->ef_flags= EFF_INUSE;
61111           eth_fd->ef_ethopt.nweo_flags=NWEO_DEFAULT;
61112           eth_fd->ef_port= eth_port;
61113           eth_fd->ef_srfd= srfd;
61114           eth_fd->ef_rd_buf= 0;
61115           eth_fd->ef_get_userdata= get_userdata;
61116           eth_fd->ef_put_userdata= put_userdata;
61117           return i;
61118   }

61120   PUBLIC int eth_ioctl(fd, req)
61121   int fd;
61122   int req;
61123   {
61124           acc_t *data;
61125           int type;
61126           eth_fd_t *eth_fd;
61127           eth_port_t *eth_port;
61128
61129   #if DEBUG & 256
61130    { where(); printf("eth_ioctl (%d, ...)\n", fd); }
61131   #endif
61132           eth_fd= &eth_fd_table[fd];
61133           eth_port= eth_fd->ef_port;
61134           type= req & IOCTYPE_MASK;
61135
61136           assert (eth_fd->ef_flags & EFF_INUSE);
61137
61138           switch (type)
61139           {
61140           case NWIOSETHOPT & IOCTYPE_MASK:
61141                   {
61142                           nwio_ethopt_t *ethopt;
61143                           nwio_ethopt_t oldopt, newopt;
61144                           int result;
```

```
61145                           u32_t new_en_flags, new_di_flags,
61146                                   old_en_flags, old_di_flags;
61147                           u32_t flags;
61148                           eth_fd_t *loc_fd;
61149                           int i;
61150
61151                           if (req != NWIOSETHOPT)
61152                                   break;
61153
61154   #if DEBUG & 256
61155    { where(); printf("calling *get_userdata\n"); }
61156   #endif
61157                           data= (*eth_fd->ef_get_userdata)(eth_fd->
61158                                   ef_srfd, 0, sizeof(nwio_ethopt_t), TRUE);
61159
61160                           ethopt= (nwio_ethopt_t *)ptr2acc_data(data);
61161                           oldopt= eth_fd->ef_ethopt;
61162                           newopt= *ethopt;
61163
61164   #if DEBUG & 256
61165    { where(); printf("newopt.nweo_flags= 0x%x\n", newopt.nweo_flags); }
61166   #endif
61167                           old_en_flags= oldopt.nweo_flags & 0xffff;
61168                           old_di_flags= (oldopt.nweo_flags >> 16) & 0xffff;
61169                           new_en_flags= newopt.nweo_flags & 0xffff;
61170                           new_di_flags= (newopt.nweo_flags >> 16) & 0xffff;
61171                           if (new_en_flags & new_di_flags)
61172                           {
61173                                   bf_afree(data);
61174                                   reply_thr_get (eth_fd, EBADMODE, TRUE);
61175                                   return NW_OK;
61176                           }
61177
61178                           /* NWEO_ACC_MASK */
61179                           if (new_di_flags & NWEO_ACC_MASK)
61180                           {
61181                                   bf_afree(data);
61182                                   reply_thr_get (eth_fd, EBADMODE, TRUE);
61183                                   return NW_OK;
61184                           }
61185                                           /* you can't disable access modes */
61186
61187                           if (!(new_en_flags & NWEO_ACC_MASK))
61188                                   new_en_flags |= (old_en_flags & NWEO_ACC_MASK);
61189
61190
61191                           /* NWEO_LOC_MASK */
61192                           if (!((new_en_flags | new_di_flags) & NWEO_LOC_MASK))
61193                           {
61194                                   new_en_flags |= (old_en_flags & NWEO_LOC_MASK);
61195                                   new_di_flags |= (old_di_flags & NWEO_LOC_MASK);
61196                           }
61197
61198                           /* NWEO_BROAD_MASK */
61199                           if (!((new_en_flags | new_di_flags) & NWEO_BROAD_MASK))
61200                           {
61201                                   new_en_flags |= (old_en_flags & NWEO_BROAD_MASK);
61202                                   new_di_flags |= (old_di_flags & NWEO_BROAD_MASK);
61203                           }
61204
```

```
61205                   /* NWEO_MULTI_MASK */
61206                   if (!((new_en_flags | new_di_flags) & NWEO_MULTI_MASK))
61207                   {
61208                           new_en_flags |= (old_en_flags & NWEO_MULTI_MASK);
61209                           new_di_flags |= (old_di_flags & NWEO_MULTI_MASK);
61210                           newopt.nweo_multi= oldopt.nweo_multi;
61211                   }
61212
61213                   /* NWEO_PROMISC_MASK */
61214                   if (!((new_en_flags | new_di_flags) & NWEO_PROMISC_MASK))
61215                   {
61216                           new_en_flags |= (old_en_flags & NWEO_PROMISC_MASK);
61217                           new_di_flags |= (old_di_flags & NWEO_PROMISC_MASK);
61218                   }
61219
61220                   /* NWEO_REM_MASK */
61221                   if (!((new_en_flags | new_di_flags) & NWEO_REM_MASK))
61222                   {
61223                           new_en_flags |= (old_en_flags & NWEO_REM_MASK);
61224                           new_di_flags |= (old_di_flags & NWEO_REM_MASK);
61225                           newopt.nweo_rem= oldopt.nweo_rem;
61226                   }
61227
61228                   /* NWEO_TYPE_MASK */
61229                   if (!((new_en_flags | new_di_flags) & NWEO_TYPE_MASK))
61230                   {
61231                           new_en_flags |= (old_en_flags & NWEO_TYPE_MASK);
61232                           new_di_flags |= (old_di_flags & NWEO_TYPE_MASK);
61233                           newopt.nweo_type= oldopt.nweo_type;
61234                   }
61235
61236                   /* NWEO_RW_MASK */
61237                   if (!((new_en_flags | new_di_flags) & NWEO_RW_MASK))
61238                   {
61239                           new_en_flags |= (old_en_flags & NWEO_RW_MASK);
61240                           new_di_flags |= (old_di_flags & NWEO_RW_MASK);
61241                   }
61242
61243                   newopt.nweo_flags= ((unsigned long)new_di_flags << 16) |
61244                           new_en_flags;
61245
61246                   eth_fd->ef_ethopt= newopt;
61247
61248                   result= eth_checkopt(eth_fd);
61249
61250                   if (result<0)
61251                           eth_fd->ef_ethopt= oldopt;
61252                   else
61253                   {
61254                           unsigned long opt_flags;
61255                           unsigned changes;
61256                           opt_flags= oldopt.nweo_flags ^
61257                                   eth_fd->ef_ethopt.nweo_flags;
61258                           changes= ((opt_flags >> 16) | opt_flags) &
61259                                   0xffff;
61260                           if (changes & (NWEO_BROAD_MASK |
61261                                   NWEO_MULTI_MASK | NWEO_PROMISC_MASK))
61262                           {
61263                                   flags= NWEO_NOFLAGS;
61264                                   for (i=0, loc_fd= eth_fd_table;
```

```
61265                                                   i<ETH_FD_NR; i++, loc_fd++)
61266                                           {
61267                                                   if (!(loc_fd->ef_flags |
61268                                                   ~(EFF_INUSE | EFF_OPTSET)))
61269                                                           continue;
61270                                                   if (loc_fd->ef_port
61271                                                           != eth_port)
61272                                                           continue;
61273                                                   flags |= loc_fd->ef_ethopt.
61274                                                           nweo_flags;
61275                                           }
61276                                           eth_set_rec_conf(eth_port, flags);
61277                                   }
61278                           }
61279
61280                           bf_afree(data);
61281                           reply_thr_get (eth_fd, result, TRUE);
61282                           return NW_OK;
61283                   }
61284
61285           case NWIOGETHOPT & IOCTYPE_MASK:
61286                   {
61287                           nwio_ethopt_t *ethopt;
61288                           acc_t *acc;
61289                           int result;
61290
61291                           if (req != NWIOGETHOPT)
61292                                   break;
61293                           acc= bf_memreq(sizeof(nwio_ethopt_t));
61294
61295                           ethopt= (nwio_ethopt_t *)ptr2acc_data(acc);
61296
61297                           *ethopt= eth_fd->ef_ethopt;
61298
61299                           result= (*eth_fd->ef_put_userdata)(eth_fd->
61300                                   ef_srfd, 0, acc, TRUE);
61301                           if (result >= 0)
61302                                   reply_thr_put(eth_fd, NW_OK, TRUE);
61303                           return result;
61304                   }
61305           case NWIOGETHSTAT & IOCTYPE_MASK:
61306                   {
61307                           nwio_ethstat_t *ethstat;
61308                           acc_t *acc;
61309                           int result;
61310
61311   assert (sizeof(nwio_ethstat_t) <= BUF_S);
61312
61313                           if (req != NWIOGETHSTAT)
61314                                   break;
61315
61316                           eth_port= eth_fd->ef_port;
61317                           if (!(eth_port->etp_flags & EPF_ENABLED))
61318                           {
61319                                   reply_thr_put(eth_fd, EGENERIC, TRUE);
61320                                   return NW_OK;
61321                           }
61322
61323                           acc= bf_memreq(sizeof(nwio_ethstat_t));
61324   compare (bf_bufsize(acc), ==, sizeof(*ethstat));
```

```
61325
61326                           ethstat= (nwio_ethstat_t *)ptr2acc_data(acc);
61327
61328                           ethstat->nwes_addr= eth_port->etp_ethaddr;
61329
61330                           result= eth_get_stat(eth_port, &ethstat->nwes_stat);
61331   assert (result == 0);
61332   #if DEBUG & 256
61333    { where(); printf("returning NW_OK\n"); }
61334   #endif
61335   compare (bf_bufsize(acc), ==, sizeof(*ethstat));
61336                           result= (*eth_fd->ef_put_userdata)(eth_fd->
61337                                   ef_srfd, 0, acc, TRUE);
61338                           if (result >= 0)
61339                                   reply_thr_put(eth_fd, NW_OK, TRUE);
61340                           return result;
61341                   }
61342           default:
61343                   break;
61344           }
61345           reply_thr_put(eth_fd, EBADIOCTL, TRUE);
61346           return NW_OK;
61347   }

61349   PUBLIC int eth_write(fd, count)
61350   int fd;
61351   size_t count;
61352   {
61353           eth_fd_t *eth_fd;
61354           eth_port_t *eth_port;
61355
61356   #if DEBUG & 256
61357    { where(); printf("eth_write (%d, ...)\n", fd); }
61358   #endif
61359           eth_fd= &eth_fd_table[fd];
61360           eth_port= eth_fd->ef_port;
61361
61362           if (!(eth_fd->ef_flags & EFF_OPTSET))
61363           {
61364                   reply_thr_get (eth_fd, EBADMODE, FALSE);
61365                   return NW_OK;
61366           }
61367
61368           assert (!(eth_fd->ef_flags & EFF_WRITE_IP));
61369
61370           eth_fd->ef_write_count= count;
61371           if (eth_fd->ef_ethopt.nweo_flags & NWEO_RWDATONLY)
61372                   count += ETH_HDR_SIZE;
61373
61374           if (count<ETH_MIN_PACK_SIZE || count>ETH_MAX_PACK_SIZE)
61375           {
61376   #if DEBUG
61377    { where(); printf("illegal packetsize (%d)\n",count); }
61378   #endif
61379                   reply_thr_get (eth_fd, EPACKSIZE, FALSE);
61380                   return NW_OK;
61381           }
61382           eth_fd->ef_flags |= EFF_WRITE_IP;
61383           restart_write_fd(eth_fd);
61384           if (eth_fd->ef_flags & EFF_WRITE_IP)
```

```
61385                   return NW_SUSPEND;
61386           else
61387                   return NW_OK;
61388   }

61390   PUBLIC int eth_read (fd, count)
61391   int fd;
61392   size_t count;
61393   {
61394           eth_fd_t *eth_fd;
61395           acc_t *acc, *acc2;
61396
61397   #if DEBUG & 256
61398    { where(); printf("eth_read (%d, ...)\n", fd); }
61399   #endif
61400           eth_fd= &eth_fd_table[fd];
61401           if (!(eth_fd->ef_flags & EFF_OPTSET))
61402           {
61403                   reply_thr_put(eth_fd, EBADMODE, FALSE);
61404                   return NW_OK;
61405           }
61406           if (count < ETH_MAX_PACK_SIZE)
61407           {
61408                   reply_thr_put(eth_fd, EPACKSIZE, FALSE);
61409                   return NW_OK;
61410           }
61411           if (eth_fd->ef_rd_buf)
61412           {
61413                   if (get_time() <= eth_fd->ef_exp_tim)
61414                           return packet2user (eth_fd);
61415                   for (acc= eth_fd->ef_rd_buf; acc;)
61416                   {
61417                           acc2= acc->acc_ext_link;
61418                           bf_afree(acc);
61419                           acc= acc2;
61420                   }
61421                   eth_fd->ef_rd_buf= 0;
61422           }
61423           eth_fd->ef_flags |= EFF_READ_IP;
61424   #if DEBUG & 256
61425    { where(); printf("eth_fd_table[%d].ef_flags= 0x%x\n",
61426           eth_fd-eth_fd_table, eth_fd->ef_flags); }
61427   #endif
61428           return NW_SUSPEND;
61429   }

61431   PUBLIC int eth_cancel(fd, which_operation)
61432   int fd;
61433   int which_operation;
61434   {
61435           eth_fd_t *eth_fd;
61436
61437   #if DEBUG & 2
61438    { where(); printf("eth_cancel (%d)\n", fd); }
61439   #endif
61440           eth_fd= &eth_fd_table[fd];
61441
61442           switch (which_operation)
61443           {
61444           case SR_CANCEL_READ:
```

```
61445    assert (eth_fd->ef_flags & EFF_READ_IP);
61446                    eth_fd->ef_flags &= ~EFF_READ_IP;
61447                    reply_thr_put(eth_fd, EINTR, FALSE);
61448                    break;
61449            case SR_CANCEL_WRITE:
61450    assert (eth_fd->ef_flags & EFF_WRITE_IP);
61451                    eth_fd->ef_flags &= ~EFF_WRITE_IP;
61452                    reply_thr_get(eth_fd, EINTR, FALSE);
61453                    break;
61454            default:
61455                    ip_panic(( "got unknown cancel request" ));
61456            }
61457            return NW_OK;
61458    }

61460    PUBLIC void eth_close(fd)
61461    int fd;
61462    {
61463            eth_fd_t *eth_fd;
61464            acc_t *acc, *acc2;
61465
61466    #if DEBUG
61467     { where(); printf("eth_close (%d)\n", fd); }
61468    #endif
61469            eth_fd= &eth_fd_table[fd];
61470
61471            assert (eth_fd->ef_flags & EFF_INUSE);
61472
61473            eth_fd->ef_flags= EFF_EMPTY;
61474            for (acc= eth_fd->ef_rd_buf; acc;)
61475            {
61476                    acc2= acc->acc_ext_link;
61477                    bf_afree(acc);
61478                    acc= acc2;
61479            }
61480            eth_fd->ef_rd_buf= 0;
61481    }

61483    PRIVATE int eth_checkopt (eth_fd)
61484    eth_fd_t *eth_fd;
61485    {
61486    /* bug: we don't check access modes yet */
61487
61488            unsigned long flags;
61489            unsigned int en_di_flags;
61490            eth_port_t *eth_port;
61491            acc_t *acc, *acc2;
61492
61493            eth_port= eth_fd->ef_port;
61494            flags= eth_fd->ef_ethopt.nweo_flags;
61495    #if DEBUG & 256
61496     { where(); printf("eth_fd_table[%d].ef_ethopt.nweo_flags= 0x%x\n",
61497            eth_fd-eth_fd_table, flags); }
61498    #endif
61499            en_di_flags= (flags >>16) | (flags & 0xffff);
61500
61501            if ((en_di_flags & NWEO_ACC_MASK) &&
61502                    (en_di_flags & NWEO_LOC_MASK) &&
61503                    (en_di_flags & NWEO_BROAD_MASK) &&
61504                    (en_di_flags & NWEO_MULTI_MASK) &&
```

```
61505                   (en_di_flags & NWEO_PROMISC_MASK) &&
61506                   (en_di_flags & NWEO_REM_MASK) &&
61507                   (en_di_flags & NWEO_TYPE_MASK) &&
61508                   (en_di_flags & NWEO_RW_MASK))
61509           {
61510                   eth_fd->ef_flags |= EFF_OPTSET;
61511                   eth_fd->ef_pack_stat= EPS_EMPTY;
61512                   if (flags & NWEO_EN_LOC)
61513                           eth_fd->ef_pack_stat |= EPS_LOC;
61514                   if (flags & NWEO_EN_BROAD)
61515                           eth_fd->ef_pack_stat |= EPS_BROAD;
61516                   if (flags & NWEO_EN_MULTI)
61517                           eth_fd->ef_pack_stat |= EPS_MULTI;
61518                   if (flags & NWEO_EN_PROMISC)
61519                           eth_fd->ef_pack_stat |= (EPS_PROMISC|EPS_MULTI|
61520                                   EPS_BROAD);
61521           }
61522           else
61523                   eth_fd->ef_flags &= ~EFF_OPTSET;
61524
61525           for (acc= eth_fd->ef_rd_buf; acc;)
61526           {
61527                   acc2= acc->acc_ext_link;
61528                   bf_afree(acc);
61529                   acc= acc2;
61530           }
61531           eth_fd->ef_rd_buf= 0;
61532
61533           return NW_OK;
61534   }

61536   PUBLIC int eth_get_work(eth_port)
61537   eth_port_t *eth_port;
61538   {
61539           eth_fd_t *eth_fd;
61540           int i;
61541
61542   #if DEBUG & 256
61543    { where(); printf("eth_get_work called\n"); }
61544   #endif
61545           if (eth_port->etp_wr_pack)
61546                   return 0;
61547           if (!(eth_port->etp_flags & EPF_MORE2WRITE))
61548                   return 0;
61549
61550           for (i=0, eth_fd= eth_fd_table; i<ETH_FD_NR; i++, eth_fd++)
61551           {
61552                   if ((eth_fd->ef_flags & (EFF_INUSE|EFF_WRITE_IP)) !=
61553                           (EFF_INUSE|EFF_WRITE_IP))
61554                           continue;
61555                   if (eth_fd->ef_port != eth_port)
61556                           continue;
61557   #if DEBUG & 256
61558    { where(); printf("eth_get_work calling restart_write_fd\n"); }
61559   #endif
61560                   restart_write_fd(eth_fd);
61561                   if (eth_port->etp_wr_pack)
61562                           return 1;
61563           }
61564           eth_port->etp_flags &= ~EPF_MORE2WRITE;
```

```
61565           return 0;
61566   }

61568   PUBLIC void eth_arrive (eth_port, pack)
61569   eth_port_t *eth_port;
61570   acc_t *pack;
61571   {
61572           time_t exp_tim;
61573           eth_hdr_t *eth_hdr;
61574           static ether_addr_t broadcast= {255, 255, 255, 255, 255, 255},
61575                   multi_addr, rem_addr, packaddr;
61576           int pack_stat;
61577           ether_type_t type;
61578           eth_fd_t *eth_fd, *share_fd;
61579           acc_t *acc;
61580           int i;
61581
61582   #if DEBUG & 256
61583    { where(); printf("eth_arrive(0x%x, 0x%x) called\n", eth_port, pack); }
61584   #endif
61585   assert(pack->acc_linkC);
61586           exp_tim= get_time() + EXPIRE_TIME;
61587
61588           pack= bf_packIffLess(pack, ETH_HDR_SIZE);
61589           eth_hdr= (eth_hdr_t*)ptr2acc_data(pack);
61590   #if DEBUG & 256
61591    { where(); printf("src= "); writeEtherAddr(&eth_hdr->eh_src); printf(" dst= ");        writeEtherAddr(&eth_hdr->eh_dst);
61592           printf(" proto= 0x%x\n", ntohs(eth_hdr->eh_proto));
61593           printf(" my addr= "); writeEtherAddr(&eth_port->etp_ethaddr);
61594           printf("\n"); }
61595   #endif
61596
61597           packaddr= eth_hdr->eh_dst;
61598           if (packaddr.ea_addr[0] & 0x01)
61599           {
61600                   /* multi cast or broadcast */
61601                   if (!eth_addrcmp(packaddr, broadcast))
61602                           pack_stat= EPS_BROAD;
61603                   else
61604                   {
61605                           pack_stat= EPS_MULTI;
61606   #if DEBUG
61607    { where(); printf("Got a multicast packet\n"); }
61608   #endif
61609                   }
61610           }
61611           else
61612           {
61613                   if (!eth_addrcmp (packaddr, eth_port->etp_ethaddr))
61614                           pack_stat= EPS_LOC;
61615                   else
61616                           pack_stat= EPS_PROMISC;
61617           }
61618           type= eth_hdr->eh_proto;
61619
61620   #if DEBUG & 256
61621    { where(); printf("pack_stat= 0x%x\n", pack_stat); }
61622   #endif
61623
61624           share_fd= 0;
```

```
61625           for (i=0, eth_fd=eth_fd_table; i<ETH_FD_NR; i++, eth_fd++)
61626           {
61627                   if (!(eth_fd->ef_flags & EFF_OPTSET))
61628                   {
61629   #if DEBUG & 256
61630    { where(); printf("fd %d doesn't have EFF_OPTSET\n", i); }
61631   #endif
61632                           continue;
61633                   }
61634                   if (eth_fd->ef_port != eth_port)
61635                   {
61636   #if DEBUG
61637    { where(); printf("fd %d uses port %d, packet is on port %d\n", i,
61638           eth_fd->ef_port-eth_port_table, eth_port-eth_port_table); }
61639   #endif
61640                           continue;
61641                   }
61642                   if (!(eth_fd->ef_pack_stat & pack_stat))
61643                   {
61644   #if DEBUG & 256
61645    { where(); printf("fd %d has ef_pack_stat 0x%x, expecting 0x%x\n", i,
61646           eth_fd->ef_pack_stat, pack_stat); }
61647   #endif
61648                           continue;
61649                   }
61650                   if ((eth_fd->ef_ethopt.nweo_flags & NWEO_TYPESPEC) &&
61651                           type != eth_fd->ef_ethopt.nweo_type)
61652                   {
61653   #if DEBUG & 256
61654    { where(); printf("fd %d uses type 0x%x, expecting 0x%x\n", i,
61655           eth_fd->ef_ethopt.nweo_type, type); }
61656   #endif
61657                           continue;
61658                   }
61659   #if DEBUG & 256
61660    { where(); printf("multi OK\n"); }
61661   #endif
61662                   if (eth_fd->ef_ethopt.nweo_flags & NWEO_REMSPEC)
61663                   {
61664                           rem_addr= eth_fd->ef_ethopt.nweo_rem;
61665                           if (eth_addrcmp (eth_hdr->eh_src,
61666                                   rem_addr))
61667                                   continue;
61668                   }
61669   #if DEBUG & 256
61670    { where(); printf("rem NW_OK\n"); }
61671   #endif
61672                   if (eth_fd->ef_rd_buf)
61673                   {
61674                           if (eth_fd->ef_ethopt.nweo_flags == NWEO_SHARED)
61675                           {
61676                                   share_fd= eth_fd;
61677                                   continue;
61678                           }
61679                   }
61680                   acc= bf_dupacc(pack);
61681                   acc->acc_ext_link= NULL;
61682                   if (!eth_fd->ef_rd_buf)
61683                   {
61684                           eth_fd->ef_rd_buf= acc;
```

```
61685                           eth_fd->ef_exp_tim= exp_tim;
61686                   }
61687                   else
61688                           eth_fd->ef_rd_tail->acc_ext_link= acc;
61689                   eth_fd->ef_rd_tail= acc;
61690
61691                   if (eth_fd->ef_flags & EFF_READ_IP)
61692                           packet2user(eth_fd);
61693                   if ((eth_fd->ef_ethopt.nweo_flags & NWEO_ACC_MASK) != NWEO_COPY)
61694                   {
61695                           bf_afree(pack);
61696                           pack= 0;
61697                           break;
61698                   }
61699           }
61700           if (share_fd && pack)
61701           {
61702                   acc= bf_dupacc(pack);
61703                   acc->acc_ext_link= NULL;
61704                   if (!share_fd->ef_rd_buf)
61705                   {
61706                           share_fd->ef_rd_buf= acc;
61707                           share_fd->ef_exp_tim= exp_tim;
61708                   }
61709                   else
61710                           share_fd->ef_rd_tail->acc_ext_link= acc;
61711                   share_fd->ef_rd_tail= acc;
61712           }
61713           if (pack)
61714                   bf_afree(pack);
61715   }

61717   PRIVATE int packet2user (eth_fd)
61718   eth_fd_t *eth_fd;
61719   {
61720           acc_t *pack, *header;
61721           int result;
61722           size_t size;
61723
61724   #if DEBUG & 256
61725    { where(); printf("packet2user() called\n"); }
61726   #endif
61727           pack= eth_fd->ef_rd_buf;
61728           eth_fd->ef_rd_buf= pack->acc_ext_link;
61729           if (eth_fd->ef_ethopt.nweo_flags & NWEO_RWDATONLY)
61730           {
61731                   pack= bf_packIffLess (pack, ETH_HDR_SIZE);
61732
61733                   assert (pack->acc_length >= ETH_HDR_SIZE);
61734
61735                   if (pack->acc_linkC >1)
61736                   {
61737                           header= bf_dupacc (pack);
61738                           bf_afree(pack);
61739                           pack= header;
61740                   }
61741
61742                   assert (pack->acc_linkC == 1);
61743
61744                   pack->acc_offset += ETH_HDR_SIZE;
```

```
61745                   pack->acc_length -= ETH_HDR_SIZE;
61746           }
61747
61748           size= bf_bufsize (pack);
61749
61750           eth_fd->ef_flags &= ~EFF_READ_IP;
61751           result= (*eth_fd->ef_put_userdata)(eth_fd->ef_srfd, (size_t)0, pack,
61752                   FALSE);
61753           if (result >=0)
61754                   reply_thr_put(eth_fd, size, FALSE);
61755           return result<0 ? result : NW_OK;
61756   }

61758   PRIVATE int ok_for_me (eth_fd, pack)
61759   eth_fd_t *eth_fd;
61760   acc_t *pack;
61761   {
61762           eth_port_t *eth_port;
61763           eth_hdr_t *eth_hdr;
61764           ether_type_t type;
61765           static ether_addr_t broadcast= {255, 255, 255, 255, 255, 255},
61766                   packaddr, portaddr, multi_addr, rem_addr;
61767           int pack_kind;
61768
61769           assert (pack->acc_length >= ETH_HDR_SIZE);
61770
61771           eth_port= eth_fd->ef_port;
61772
61773           eth_hdr= (eth_hdr_t *)ptr2acc_data(pack);
61774           packaddr= eth_hdr->eh_dst;
61775           if (packaddr.ea_addr[0] & 0x01)
61776                   /* multi cast or broadcast */
61777                   if (!eth_addrcmp (packaddr, broadcast))
61778                           pack_kind= EPS_BROAD;
61779                   else
61780                           pack_kind= EPS_MULTI;
61781           else
61782           {
61783                   portaddr= eth_port->etp_ethaddr;
61784                   if (!eth_addrcmp(packaddr, portaddr))
61785                           pack_kind= EPS_LOC;
61786                   else
61787                           pack_kind= EPS_PROMISC;
61788           }
61789
61790           pack_kind &= eth_fd->ef_pack_stat;
61791
61792           if (!pack_kind)
61793                   return FALSE;
61794
61795           type= eth_hdr->eh_proto;
61796
61797           if ((eth_fd->ef_ethopt.nweo_flags & NWEO_TYPESPEC) &&
61798                   type != eth_fd->ef_ethopt.nweo_type)
61799                   return FALSE;
61800
61801           if (eth_fd->ef_ethopt.nweo_flags & NWEO_REMSPEC)
61802           {
61803                   rem_addr= eth_fd->ef_ethopt.nweo_rem;
61804                   if (eth_addrcmp(eth_hdr->eh_src, rem_addr))
```

```
61805                           return FALSE;
61806           }
61807           return TRUE;
61808   }

61810   PRIVATE void eth_buffree (priority, reqsize)
61811   int priority;
61812   size_t reqsize;
61813   {
61814           int i, once_more;
61815           time_t curr_tim;
61816           acc_t *acc;
61817
61818           if (priority <ETH_PRI_EXP_FDBUFS)
61819                   return;
61820
61821   #if DEBUG & 256
61822    { where(); printf("eth_buffree called\n"); }
61823   #endif
61824
61825           curr_tim= get_time();
61826           for (i=0; i<ETH_FD_NR; i++)
61827           {
61828                   if (!(eth_fd_table[i].ef_flags & EFF_INUSE) )
61829                           continue;
61830                   acc= eth_fd_table[i].ef_rd_buf;
61831                   if (acc && eth_fd_table[i].ef_exp_tim < curr_tim)
61832                   {
61833                           eth_fd_table[i].ef_rd_buf= acc->acc_ext_link;
61834                           bf_afree(acc);
61835                           if (bf_free_buffsize >= reqsize)
61836                                   return;
61837                   }
61838           }
61839
61840           if (priority <ETH_PRI_FDBUFS)
61841                   return;
61842
61843           once_more= 1;
61844           while (once_more)
61845           {
61846                   once_more= 0;
61847                   for (i=0; i<ETH_FD_NR; i++)
61848                   {
61849                           if (!(eth_fd_table[i].ef_flags & EFF_INUSE))
61850                                   continue;
61851                           acc= eth_fd_table[i].ef_rd_buf;
61852                           if (acc)
61853                           {
61854                                   eth_fd_table[i].ef_rd_buf= acc->acc_ext_link;
61855                                   bf_afree(acc);
61856                                   if (bf_free_buffsize >= reqsize)
61857                                           return;
61858                                   once_more= 1;
61859                           }
61860                   }
61861           }
61862   }

61864   PRIVATE void restart_write_fd(eth_fd)
```

```
61865  eth_fd_t *eth_fd;
61866  {
61867          eth_port_t *eth_port;
61868          acc_t *user_data, *header;
61869          int size;
61870          unsigned long nweo_flags;
61871          eth_hdr_t *eth_hdr;
61872
61873          eth_port= eth_fd->ef_port;
61874
61875          if (eth_port->etp_wr_pack)
61876          {
61877                  eth_port->etp_flags |= EPF_MORE2WRITE;
61878                  return;
61879          }
61880
61881  assert (eth_fd->ef_flags & EFF_WRITE_IP);
61882          eth_fd->ef_flags &= ~EFF_WRITE_IP;
61883
61884  assert (!eth_port->etp_wr_pack);
61885
61886  #if DEBUG & 256
61887   { where(); printf("calling *get_userdata\n"); }
61888  #endif
61889          user_data= (*eth_fd->ef_get_userdata)(eth_fd->ef_srfd, 0,
61890                  eth_fd->ef_write_count, FALSE);
61891          if (!user_data)
61892          {
61893                  eth_fd->ef_flags &= ~EFF_WRITE_IP;
61894                  reply_thr_get (eth_fd, EFAULT, FALSE);
61895                  return;
61896          }
61897          size= bf_bufsize (user_data);
61898
61899          nweo_flags= eth_fd->ef_ethopt.nweo_flags;
61900
61901          if (nweo_flags & NWEO_RWDATONLY)
61902          {
61903                  header= bf_memreq(ETH_HDR_SIZE);
61904                  header->acc_next= user_data;
61905                  user_data= header;
61906          }
61907
61908          user_data= bf_packIffLess (user_data, ETH_HDR_SIZE);
61909
61910          eth_hdr= (eth_hdr_t *)ptr2acc_data(user_data);
61911
61912          if (nweo_flags & NWEO_REMSPEC)
61913                  eth_hdr->eh_dst= eth_fd->ef_ethopt.nweo_rem;
61914
61915          eth_hdr->eh_src= eth_port->etp_ethaddr;
61916
61917          if (nweo_flags & NWEO_TYPESPEC)
61918                  eth_hdr->eh_proto= eth_fd->ef_ethopt.nweo_type;
61919
61920  assert (!eth_port->etp_wr_pack);
61921          eth_port->etp_wr_pack= user_data;
61922
61923          if (!(eth_port->etp_flags & EPF_WRITE_IP))
61924          {
```

```
61925                   eth_write_port(eth_port);
61926           }
61927           reply_thr_get (eth_fd, size, FALSE);
61928   }

61930   PRIVATE void reply_thr_get (eth_fd, result, for_ioctl)
61931   eth_fd_t *eth_fd;
61932   size_t result;
61933   int for_ioctl;
61934   {
61935           acc_t *data;
61936
61937   #if DEBUG & 256
61938    { where(); printf("calling *get_userdata(fd= %d, %d, 0)\n", eth_fd->
61939           ef_srfd, result, 0); }
61940   #endif
61941           data= (*eth_fd->ef_get_userdata)(eth_fd->ef_srfd, result, 0, for_ioctl);
61942   assert (!data);
61943   }

61945   PRIVATE void reply_thr_put (eth_fd, result, for_ioctl)
61946   eth_fd_t *eth_fd;
61947   size_t result;
61948   int for_ioctl;
61949   {
61950           int error;
61951
61952           error= (*eth_fd->ef_put_userdata)(eth_fd->ef_srfd, result, (acc_t *)0,
61953                   for_ioctl);
61954   assert(!error);
61955   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/icmp.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

62000   /*
62001   icmp.c
62002   */
62003
62004   #include "inet.h"
62005   #include "buf.h"
62006   #include "type.h"
62007
62008   #include "assert.h"
62009   #include "clock.h"
62010   #include "icmp.h"
62011   #include "icmp_lib.h"
62012   #include "io.h"
62013   #include "ip.h"
62014   #include "ip_int.h"
62015   #include "ipr.h"
62016
62017   INIT_PANIC();
62018
62019   typedef struct icmp_port
```

```
62020   {
62021           int icp_flags;
62022           int icp_state;
62023           int icp_ipport;
62024           int icp_ipfd;
62025           acc_t *icp_head_queue;
62026           acc_t *icp_tail_queue;
62027           acc_t *icp_write_pack;
62028   } icmp_port_t;
62029
62030   #define ICPF_EMPTY      0x0
62031   #define ICPF_SUSPEND    0x1
62032   #define ICPF_READ_IP    0x2
62033   #define ICPF_READ_SP    0x4
62034   #define ICPF_WRITE_IP   0x8
62035   #define ICPF_WRITE_SP   0x10
62036
62037   #define ICPS_BEGIN      0
62038   #define ICPS_IPOPT      1
62039   #define ICPS_MAIN       2
62040   #define ICPS_ERROR      3
62041
62042   #define ICMP_PORT_NR    IP_PORT_NR
62043
62044   PRIVATE  icmp_port_t icmp_port_table[ICMP_PORT_NR];
62045
62046   FORWARD void icmp_main ARGS(( icmp_port_t *icmp_port ));
62047   FORWARD acc_t *icmp_getdata ARGS(( int port, size_t offset,
62048           size_t count, int for_ioctl ));
62049   FORWARD int icmp_putdata ARGS(( int port, size_t offset,
62050           acc_t *data, int for_ioctl ));
62051   FORWARD void icmp_read ARGS(( icmp_port_t *icmp_port ));
62052   FORWARD void process_data ARGS(( icmp_port_t *icmp_port,
62053           acc_t *data ));
62054   FORWARD u16_t icmp_pack_oneCsum ARGS(( acc_t *ip_pack ));
62055   FORWARD void icmp_echo_request ARGS(( icmp_port_t *icmp_port,
62056           acc_t *ip_pack, int ip_hdr_len, ip_hdr_t *ip_hdr,
62057           acc_t *icmp_pack, int icmp_len, icmp_hdr_t *icmp_hdr ));
62058   FORWARD void icmp_dst_unreach ARGS(( icmp_port_t *icmp_port,
62059           acc_t *ip_pack, int ip_hdr_len, ip_hdr_t *ip_hdr,
62060           acc_t *icmp_pack, int icmp_len, icmp_hdr_t *icmp_hdr ));
62061   FORWARD void icmp_time_exceeded ARGS(( icmp_port_t *icmp_port,
62062           acc_t *ip_pack, int ip_hdr_len, ip_hdr_t *ip_hdr,
62063           acc_t *icmp_pack, int icmp_len, icmp_hdr_t *icmp_hdr ));
62064   FORWARD void icmp_router_advertisement ARGS(( icmp_port_t *icmp_port,
62065           acc_t *icmp_pack, int icmp_len, icmp_hdr_t *icmp_hdr ));
62066   FORWARD void icmp_redirect ARGS(( icmp_port_t *icmp_port,
62067           ip_hdr_t *ip_hdr, acc_t *icmp_pack, int icmp_len,
62068           icmp_hdr_t *icmp_hdr ));
62069   FORWARD acc_t *make_repl_ip ARGS(( ip_hdr_t *ip_hdr,
62070           int ip_len ));
62071   FORWARD void enqueue_pack ARGS(( icmp_port_t *icmp_port,
62072           acc_t *reply_ip_hdr ));
62073   FORWARD void icmp_write ARGS(( icmp_port_t *icmp_port ));
62074   FORWARD void icmp_buffree ARGS(( int priority, size_t reqsize ));
62075
62076   PUBLIC void icmp_init()
62077   {
62078           int i;
62079           icmp_port_t *icmp_port;
```

```
62080
62081           assert (BUF_S >= sizeof (nwio_ipopt_t));
62082
62083           for (i= 0, icmp_port= icmp_port_table; i<ICMP_PORT_NR; i++,
62084                   icmp_port++)
62085           {
62086                   icmp_port->icp_flags= ICPF_EMPTY;
62087                   icmp_port->icp_state= ICPS_BEGIN;
62088                   icmp_port->icp_ipport= i;
62089           }
62090
62091           bf_logon(icmp_buffree);
62092
62093           for (i= 0, icmp_port= icmp_port_table; i<ICMP_PORT_NR; i++,
62094                   icmp_port++)
62095           {
62096                   icmp_main (icmp_port);
62097           }
62098   }

62100   PRIVATE void icmp_main(icmp_port)
62101   icmp_port_t *icmp_port;
62102   {
62103           int result;
62104           switch (icmp_port->icp_state)
62105           {
62106           case ICPS_BEGIN:
62107                   icmp_port->icp_head_queue= 0;
62108                   icmp_port->icp_ipfd= ip_open (icmp_port->icp_ipport,
62109                           icmp_port-icmp_port_table, icmp_getdata,
62110                           icmp_putdata);
62111                   if (icmp_port->icp_ipfd<0)
62112                   {
62113                           where();
62114                           printf("unable to open ip_port %d\n",
62115                                   icmp_port->icp_ipport);
62116                           break;
62117                   }
62118                   icmp_port->icp_state= ICPS_IPOPT;
62119                   icmp_port->icp_flags &= ~ICPF_SUSPEND;
62120                   result= ip_ioctl (icmp_port->icp_ipfd, NWIOSIPOPT);
62121                   if (result == NW_SUSPEND)
62122                   {
62123                           icmp_port->icp_flags |= ICPF_SUSPEND;
62124                           break;
62125                   }
62126                   else if (result<0)
62127                   {
62128                           where();
62129                           printf("ip_ioctl (.., NWIOSIPOPT)= %d\n",
62130                                   result);
62131                           break;
62132                   }
62133                   /* falls through */
62134           case ICPS_IPOPT:
62135                   icmp_port->icp_state= ICPS_MAIN;
62136                   icmp_port->icp_flags &= ~ICPF_SUSPEND;
62137                   icmp_read(icmp_port);
62138                   break;
62139           default:
```

```
62140                   where();
62141                   printf("unknown state %d\n", icmp_port->icp_state);
62142                   break;
62143           }
62144   }

62146   PRIVATE acc_t *icmp_getdata(port, offset, count, for_ioctl)
62147   int port;
62148   size_t offset, count;
62149   int for_ioctl;
62150   {
62151           icmp_port_t *icmp_port;
62152           nwio_ipopt_t *ipopt;
62153           acc_t *data;
62154           int result;
62155
62156           icmp_port= &icmp_port_table[port];
62157
62158           if (icmp_port->icp_flags & ICPF_WRITE_IP)
62159           {
62160                   if (!count)
62161                   {
62162                           bf_afree(icmp_port->icp_write_pack);
62163                           icmp_port->icp_write_pack= 0;
62164
62165                           result= (int)offset;
62166                           if (result<0)
62167                           {
62168                                   where();
62169                                   printf("got write error %d\n", result);
62170                           }
62171   #if DEBUG & 256
62172    { where(); printf("ip_write completed\n"); }
62173   #endif
62174                           if (icmp_port->icp_flags & ICPF_WRITE_SP)
62175                           {
62176                                   icmp_port->icp_flags &=
62177                                           ~(ICPF_WRITE_IP|ICPF_WRITE_SP);
62178                                   icmp_write (icmp_port);
62179                           }
62180                           return NW_OK;
62181                   }
62182                   return bf_cut(icmp_port->icp_write_pack, offset, count);
62183           }
62184           switch (icmp_port->icp_state)
62185           {
62186           case ICPS_IPOPT:
62187                   if (!count)
62188                   {
62189                           result= (int)offset;
62190                           if (result < 0)
62191                           {
62192                                   icmp_port->icp_state= ICPS_ERROR;
62193                                   break;
62194                           }
62195                           if (icmp_port->icp_flags & ICPF_SUSPEND)
62196                                   icmp_main(icmp_port);
62197                           return NW_OK;
62198                   }
62199
```

```
62200   assert (count == sizeof (*ipopt));
62201                   data= bf_memreq (sizeof (*ipopt));
62202   assert (data->acc_length == sizeof(*ipopt));
62203                   ipopt= (nwio_ipopt_t *)ptr2acc_data(data);
62204                   ipopt->nwio_flags= NWIO_COPY | NWIO_EN_LOC |
62205                           NWIO_EN_BROAD | NWIO_REMANY | NWIO_PROTOSPEC |
62206                           NWIO_HDR_O_ANY | NWIO_RWDATALL;
62207                   ipopt->nwio_proto= IPPROTO_ICMP;
62208                   return data;
62209           default:
62210                   where();
62211                   printf("unknown state %d\n", icmp_port->icp_state);
62212                   return 0;
62213           }
62214   }

62216   PRIVATE int icmp_putdata(port, offset, data, for_ioctl)
62217   int port;
62218   size_t offset;
62219   acc_t *data;
62220   int for_ioctl;
62221   {
62222           icmp_port_t *icmp_port;
62223           int result;
62224
62225           icmp_port= &icmp_port_table[port];
62226
62227           if (icmp_port->icp_flags & ICPF_READ_IP)
62228           {
62229   assert (!for_ioctl);
62230                   if (!data)
62231                   {
62232                           result= (int)offset;
62233                           if (result<0)
62234                           {
62235                                   where();
62236                                   printf("got read error %d\n", result);
62237                           }
62238                           if (icmp_port->icp_flags & ICPF_READ_SP)
62239                           {
62240                                   icmp_port->icp_flags &=
62241                                           ~(ICPF_READ_IP|ICPF_READ_SP);
62242                                   icmp_read (icmp_port);
62243                           }
62244                           return NW_OK;
62245                   }
62246                   process_data(icmp_port, data);
62247                   return NW_OK;
62248           }
62249           switch (icmp_port->icp_state)
62250           {
62251           default:
62252                   where();
62253                   printf("unknown state %d\n", icmp_port->icp_state);
62254                   return 0;
62255           }
62256   }

62258   PRIVATE void icmp_read(icmp_port)
62259   icmp_port_t *icmp_port;
```

```
62260   {
62261           int result;
62262
62263   assert (!(icmp_port->icp_flags & (ICPF_READ_IP|ICPF_READ_SP) ||
62264           (icmp_port->icp_flags & (ICPF_READ_IP|ICPF_READ_SP)) ==
62265           (ICPF_READ_IP|ICPF_READ_SP)));
62266
62267           for (;;)
62268           {
62269                   icmp_port->icp_flags |= ICPF_READ_IP;
62270                   icmp_port->icp_flags &= ~ICPF_READ_SP;
62271
62272                   result= ip_read(icmp_port->icp_ipfd, ICMP_MAX_DATAGRAM);
62273                   if (result == NW_SUSPEND)
62274                   {
62275                           icmp_port->icp_flags |= ICPF_READ_SP;
62276                           return;
62277                   }
62278           }
62279   }

62281   PUBLIC void icmp_frag_ass_tim(pack)
62282   acc_t *pack;
62283   {
62284           ip_warning(( "icmp_frag_ass() called" ));
62285           bf_afree(pack);
62286   }

62288   PUBLIC void icmp_getnetmask(ip_port)
62289   int ip_port;
62290   {
62291           ip_port_t *port;
62292
62293   #if DEBUG & 256
62294    { where(); printf("icmp.c: icmp_getnetmask(ip_port= %d)\n", ip_port); }
62295   #endif
62296           port= &ip_port_table[ip_port];
62297   #if DEBUG & 2
62298    { where(); printf ("icmp_getnetmask() NOT implemented\n"); }
62299   #endif
62300
62301           port->ip_netmask= HTONL(0xffffff00L);
62302           port->ip_flags |= IPF_NETMASKSET;
62303   #if DEBUG & 256
62304    { where(); printf("icmp.c: setting netmask to ");
62305      writeIpAddr(port->ip_netmask); printf("\n"); }
62306   #endif
62307   }

62309   PUBLIC void icmp_dont_frag(pack)
62310   acc_t *pack;
62311   {
62312   printf ("icmp_dont_frag() called\n");
62313           bf_afree(pack);
62314   }

62316   PUBLIC void icmp_ttl_exceded(pack)
62317   acc_t *pack;
62318   {
62319   printf ("icmp_ttl_execeded() called\n");
```

```
62320           bf_afree(pack);
62321   }

62323   PRIVATE void process_data(icmp_port, data)
62324   icmp_port_t *icmp_port;
62325   acc_t *data;
62326   {
62327           ip_hdr_t *ip_hdr;
62328           icmp_hdr_t *icmp_hdr;
62329           acc_t *icmp_data;
62330           int ip_hdr_len;
62331           size_t pack_len;
62332
62333   #if DEBUG & 256
62334    { where(); printf("got an icmp packet\n"); }
62335   #endif
62336           data= bf_packIffLess(data, IP_MIN_HDR_SIZE);
62337           ip_hdr= (ip_hdr_t *)ptr2acc_data(data);
62338           ip_hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
62339
62340           if (ip_hdr_len>IP_MIN_HDR_SIZE)
62341           {
62342                   data= bf_packIffLess(data, ip_hdr_len);
62343                   ip_hdr= (ip_hdr_t *)ptr2acc_data(data);
62344           }
62345
62346           pack_len= bf_bufsize(data);
62347           pack_len -= ip_hdr_len;
62348           if (pack_len < ICMP_MIN_HDR_LEN)
62349           {
62350   #if DEBUG
62351    { where(); printf("got an incomplete icmp packet\n"); }
62352   #endif
62353                   bf_afree(data);
62354                   return;
62355           }
62356
62357           icmp_data= bf_cut(data, ip_hdr_len, pack_len);
62358
62359           icmp_data= bf_packIffLess (icmp_data, ICMP_MIN_HDR_LEN);
62360           icmp_hdr= (icmp_hdr_t *)ptr2acc_data(icmp_data);
62361
62362
62363           if ((u16_t)~icmp_pack_oneCsum(icmp_data))
62364           {
62365   #if DEBUG
62366    { where(); printf("got packet with bad checksum (= 0x%x)\n",
62367           (u16_t)~icmp_pack_oneCsum(icmp_data)); }
62368   #endif
62369                   bf_afree(data);
62370                   bf_afree(icmp_data);
62371                   return;
62372           }
62373           switch (icmp_hdr->ih_type)
62374           {
62375           case ICMP_TYPE_ECHO_REPL:
62376   #if DEBUG
62377    { where(); printf("got an icmp echo reply\n"); }
62378   #endif
62379                   break;
```

```
62380           case ICMP_TYPE_DST_UNRCH:
62381                   icmp_dst_unreach (icmp_port, data, ip_hdr_len, ip_hdr,
62382                           icmp_data, pack_len, icmp_hdr);
62383                   break;
62384           case ICMP_TYPE_REDIRECT:
62385                   icmp_redirect (icmp_port, ip_hdr, icmp_data, pack_len,
62386                           icmp_hdr);
62387                   break;
62388           case ICMP_TYPE_ECHO_REQ:
62389                   icmp_echo_request(icmp_port, data, ip_hdr_len, ip_hdr,
62390                           icmp_data, pack_len, icmp_hdr);
62391                   return;
62392           case ICMP_TYPE_ROUTER_ADVER:
62393                   icmp_router_advertisement(icmp_port, icmp_data, pack_len,
62394                           icmp_hdr);
62395                   break;
62396           case ICMP_TYPE_TIME_EXCEEDED:
62397                   icmp_time_exceeded (icmp_port, data, ip_hdr_len, ip_hdr,
62398                           icmp_data, pack_len, icmp_hdr);
62399                   break;
62400           default:
62401   #if DEBUG
62402    { where(); printf("got an unknown icmp (%d) from ", icmp_hdr->ih_type);
62403           writeIpAddr(ip_hdr->ih_src); printf("\n"); }
62404   #endif
62405                   break;
62406           }
62407           bf_afree(data);
62408           bf_afree(icmp_data);
62409   }

62411   PRIVATE void icmp_echo_request(icmp_port, ip_data, ip_len, ip_hdr,
62412           icmp_data, icmp_len, icmp_hdr)
62413   icmp_port_t *icmp_port;
62414   acc_t *ip_data, *icmp_data;
62415   int ip_len, icmp_len;
62416   ip_hdr_t *ip_hdr;
62417   icmp_hdr_t *icmp_hdr;
62418   {
62419           acc_t *repl_ip_hdr, *repl_icmp;
62420           icmp_hdr_t *repl_icmp_hdr;
62421           u32_t tmp_chksum;
62422           u16_t u16;
62423
62424           if (icmp_hdr->ih_code != 0)
62425           {
62426   #if DEBUG
62427    { where(); printf("got an icmp echo request with unknown code (%d)\n",
62428                   icmp_hdr->ih_code); }
62429   #endif
62430                   bf_afree(ip_data);
62431                   bf_afree(icmp_data);
62432                   return;
62433           }
62434           if (icmp_len < ICMP_MIN_HDR_LEN + sizeof(icmp_id_seq_t))
62435           {
62436   #if DEBUG
62437    { where(); printf("got an incomplete icmp echo request\n"); }
62438   #endif
62439                   bf_afree(ip_data);
```

```
62440                   bf_afree(icmp_data);
62441                   return;
62442           }
62443   #if DEBUG & 256
62444    { where(); printf("got an icmp echo request, ident= %u, seq= %u\n",
62445           ntohs(icmp_hdr->ih_hun.ihh_idseq.iis_id),
62446           ntohs(icmp_hdr->ih_hun.ihh_idseq.iis_seq)); }
62447   #endif
62448           repl_ip_hdr= make_repl_ip(ip_hdr, ip_len);
62449           repl_icmp= bf_memreq (ICMP_MIN_HDR_LEN);
62450   assert (repl_icmp->acc_length == ICMP_MIN_HDR_LEN);
62451           repl_icmp_hdr= (icmp_hdr_t *)ptr2acc_data(repl_icmp);
62452           repl_icmp_hdr->ih_type= ICMP_TYPE_ECHO_REPL;
62453           repl_icmp_hdr->ih_code= 0;
62454
62455           tmp_chksum= ~icmp_hdr->ih_chksum - *(u16_t *)&icmp_hdr->ih_type+
62456                   *(u16_t *)&repl_icmp_hdr->ih_type;
62457           tmp_chksum= (tmp_chksum >> 16) + (tmp_chksum & 0xffff);
62458           tmp_chksum= (tmp_chksum >> 16) + (tmp_chksum & 0xffff);
62459           repl_icmp_hdr->ih_chksum= ~tmp_chksum;
62460
62461           repl_ip_hdr->acc_next= repl_icmp;
62462           repl_icmp->acc_next= bf_cut (icmp_data, ICMP_MIN_HDR_LEN,
62463                   icmp_len - ICMP_MIN_HDR_LEN);
62464
62465           bf_afree(ip_data);
62466           bf_afree(icmp_data);
62467
62468           enqueue_pack(icmp_port, repl_ip_hdr);
62469   }

62471   PRIVATE u16_t icmp_pack_oneCsum(icmp_pack)
62472   acc_t *icmp_pack;
62473   {
62474           u16_t prev;
62475           int odd_byte;
62476           char *data_ptr;
62477           int length;
62478           char byte_buf[2];
62479
62480           assert (icmp_pack);
62481
62482           prev= 0;
62483
62484           odd_byte= FALSE;
62485           for (; icmp_pack; icmp_pack= icmp_pack->acc_next)
62486           {
62487
62488                   data_ptr= ptr2acc_data(icmp_pack);
62489                   length= icmp_pack->acc_length;
62490
62491                   if (!length)
62492                           continue;
62493                   if (odd_byte)
62494                   {
62495                           byte_buf[1]= *data_ptr;
62496                           prev= oneC_sum(prev, (u16_t *)byte_buf, 2);
62497                           data_ptr++;
62498                           length--;
62499                           odd_byte= FALSE;
```

```
62500                   }
62501                   if (length & 1)
62502                   {
62503                           odd_byte= TRUE;
62504                           length--;
62505                           byte_buf[0]= data_ptr[length];
62506                   }
62507                   if (!length)
62508                           continue;
62509                   prev= oneC_sum (prev, (u16_t *)data_ptr, length);
62510           }
62511           if (odd_byte)
62512                   prev= oneC_sum (prev, (u16_t *)byte_buf, 1);
62513           return prev;
62514   }

62516   PRIVATE acc_t *make_repl_ip(ip_hdr, ip_len)
62517   ip_hdr_t *ip_hdr;
62518   int ip_len;
62519   {
62520           ip_hdr_t *repl_ip_hdr;
62521           acc_t *repl;
62522           int repl_hdr_len;
62523
62524           if (ip_len>IP_MIN_HDR_SIZE)
62525           {
62526   #if DEBUG
62527    { where(); printf("ip_hdr options NOT supported (yet?)\n"); }
62528   #endif
62529                   ip_len= IP_MIN_HDR_SIZE;
62530           }
62531
62532           repl_hdr_len= IP_MIN_HDR_SIZE;
62533
62534           repl= bf_memreq(repl_hdr_len);
62535   assert (repl->acc_length == repl_hdr_len);
62536
62537           repl_ip_hdr= (ip_hdr_t *)ptr2acc_data(repl);
62538
62539           repl_ip_hdr->ih_vers_ihl= repl_hdr_len >> 2;
62540           repl_ip_hdr->ih_tos= ip_hdr->ih_tos;
62541           repl_ip_hdr->ih_ttl= ICMP_DEF_TTL;
62542           repl_ip_hdr->ih_proto= IPPROTO_ICMP;
62543           repl_ip_hdr->ih_dst= ip_hdr->ih_src;
62544           repl_ip_hdr->ih_flags_fragoff= 0;
62545
62546           return repl;
62547   }

62549   PRIVATE void enqueue_pack(icmp_port, reply_ip_hdr)
62550   icmp_port_t *icmp_port;
62551   acc_t *reply_ip_hdr;
62552   {
62553           reply_ip_hdr->acc_ext_link= 0;
62554
62555           if (icmp_port->icp_head_queue)
62556           {
62557                   icmp_port->icp_tail_queue->acc_ext_link=
62558                           reply_ip_hdr;
62559           }
```

```
62560           else
62561           {
62562                   icmp_port->icp_head_queue= reply_ip_hdr;
62563                   icmp_port->icp_tail_queue= reply_ip_hdr;
62564           }
62565
62566           if (!(icmp_port->icp_flags & ICPF_WRITE_IP))
62567                   icmp_write(icmp_port);
62568   }

62570   PRIVATE void icmp_write(icmp_port)
62571   icmp_port_t *icmp_port;
62572   {
62573           int result;
62574
62575   assert (!(icmp_port->icp_flags & (ICPF_WRITE_IP|ICPF_WRITE_SP) ||
62576           (icmp_port->icp_flags & (ICPF_WRITE_IP|ICPF_WRITE_SP)) ==
62577           (ICPF_WRITE_IP|ICPF_WRITE_SP)));
62578
62579           for (;icmp_port->icp_head_queue;)
62580           {
62581                   icmp_port->icp_write_pack= icmp_port->icp_head_queue;
62582                   icmp_port->icp_head_queue= icmp_port->icp_head_queue->
62583                           acc_ext_link;
62584
62585                   icmp_port->icp_flags |= ICPF_WRITE_IP;
62586                   icmp_port->icp_flags &= ~ICPF_WRITE_SP;
62587
62588   #if DEBUG & 256
62589    { where(); printf("calling ip_write\n"); }
62590   #endif
62591                   result= ip_write(icmp_port->icp_ipfd,
62592                           bf_bufsize(icmp_port->icp_write_pack));
62593                   if (result == NW_SUSPEND)
62594                   {
62595   #if DEBUG & 256
62596    { where(); printf("ip_write replied NW_SUSPEND\n"); }
62597   #endif
62598                           icmp_port->icp_flags |= ICPF_WRITE_SP;
62599                           return;
62600                   }
62601   #if DEBUG & 256
62602    { where(); printf("ip_write done\n"); }
62603   #endif
62604           }
62605           icmp_port->icp_flags &= ~ICPF_WRITE_IP;
62606   }

62608   PRIVATE void icmp_buffree(priority, reqsize)
62609   int priority;
62610   size_t reqsize;
62611   {
62612           acc_t *tmp_acc;
62613           int donesomething,i;
62614           icmp_port_t *icmp_port;
62615
62616           donesomething= 0;
62617
62618           if (priority < ICMP_PRI_QUEUE)
62619                   return;
```

```
62620
62621           while (bf_free_buffsize < reqsize)
62622           {
62623                   for (i=0, icmp_port= icmp_port_table; i<ICMP_PORT_NR;
62624                           i++, icmp_port++)
62625                   {
62626                           if (icmp_port->icp_head_queue)
62627                           {
62628                                   tmp_acc= icmp_port->icp_head_queue;
62629                                   icmp_port->icp_head_queue= tmp_acc->
62630                                           acc_ext_link;
62631                                   bf_afree(tmp_acc);
62632                                   if (bf_free_buffsize >= reqsize)
62633                                           break;
62634                                   donesomething= 1;
62635                           }
62636                   }
62637                   if (!donesomething)
62638                           break;
62639           }
62640   }

62642   static void icmp_dst_unreach(icmp_port, ip_pack, ip_hdr_len, ip_hdr, icmp_pack,
62643           icmp_len, icmp_hdr)
62644   icmp_port_t *icmp_port;
62645   acc_t *ip_pack;
62646   int ip_hdr_len;
62647   ip_hdr_t *ip_hdr;
62648   acc_t *icmp_pack;
62649   int icmp_len;
62650   icmp_hdr_t *icmp_hdr;
62651   {
62652           acc_t *old_ip_pack;
62653           ip_hdr_t *old_ip_hdr;
62654
62655           if (icmp_len < 8 + IP_MIN_HDR_SIZE)
62656           {
62657   #if DEBUG
62658    { where(); printf("dest unrch with wrong size\n"); }
62659   #endif
62660                   return;
62661           }
62662           old_ip_pack= bf_cut (icmp_pack, 8, icmp_len-8);
62663           old_ip_pack= bf_packIffLess(old_ip_pack, IP_MIN_HDR_SIZE);
62664           old_ip_hdr= (ip_hdr_t *)ptr2acc_data(old_ip_pack);
62665
62666           if (old_ip_hdr->ih_src != ip_hdr->ih_dst)
62667           {
62668   #if DEBUG
62669    { where(); printf("dest unrch based on wrong packet\n"); }
62670   #endif
62671                   bf_afree(old_ip_pack);
62672                   return;
62673           }
62674
62675           switch(icmp_hdr->ih_code)
62676           {
62677           case ICMP_NET_UNRCH:
62678                   ipr_destunrch (old_ip_hdr->ih_dst,
62679                           ip_get_netmask(old_ip_hdr->ih_dst), IPR_UNRCH_TIMEOUT);
```

```
62680                   break;
62681           case ICMP_HOST_UNRCH:
62682                   ipr_destunrch (old_ip_hdr->ih_dst, (ipaddr_t)-1,
62683                           IPR_UNRCH_TIMEOUT);
62684                   break;
62685           default:
62686   #if DEBUG
62687    { where(); printf("got strange code: %d\n", icmp_hdr->ih_code); }
62688   #endif
62689                   break;
62690           }
62691           bf_afree(old_ip_pack);
62692   }

62694   static void icmp_time_exceeded(icmp_port, ip_pack, ip_hdr_len, ip_hdr,
62695           icmp_pack, icmp_len, icmp_hdr)
62696   icmp_port_t *icmp_port;
62697   acc_t *ip_pack;
62698   int ip_hdr_len;
62699   ip_hdr_t *ip_hdr;
62700   acc_t *icmp_pack;
62701   int icmp_len;
62702   icmp_hdr_t *icmp_hdr;
62703   {
62704           acc_t *old_ip_pack;
62705           ip_hdr_t *old_ip_hdr;
62706
62707           if (icmp_len < 8 + IP_MIN_HDR_SIZE)
62708           {
62709   #if DEBUG
62710    { where(); printf("time exceeded with wrong size\n"); }
62711   #endif
62712                   return;
62713           }
62714           old_ip_pack= bf_cut (icmp_pack, 8, icmp_len-8);
62715           old_ip_pack= bf_packIffLess(old_ip_pack, IP_MIN_HDR_SIZE);
62716           old_ip_hdr= (ip_hdr_t *)ptr2acc_data(old_ip_pack);
62717
62718           if (old_ip_hdr->ih_src != ip_hdr->ih_dst)
62719           {
62720   #if DEBUG
62721    { where(); printf("time exceeded based on wrong packet\n"); }
62722   #endif
62723                   bf_afree(old_ip_pack);
62724                   return;
62725           }
62726
62727           switch(icmp_hdr->ih_code)
62728           {
62729           case ICMP_TTL_EXC:
62730                   ipr_ttl_exc (old_ip_hdr->ih_dst, (ipaddr_t)-1,
62731                           IPR_TTL_TIMEOUT);
62732                   break;
62733           default:
62734                   where();
62735                   printf("got strange code: %d\n", icmp_hdr->ih_code);
62736                   break;
62737           }
62738           bf_afree(old_ip_pack);
62739   }
```

```
62741  static void icmp_router_advertisement(icmp_port, icmp_pack, icmp_len, icmp_hdr)
62742  icmp_port_t *icmp_port;
62743  acc_t *icmp_pack;
62744  int icmp_len;
62745  icmp_hdr_t *icmp_hdr;
62746  {
62747          int entries;
62748          int entry_size;
62749          u16_t lifetime;
62750          int i;
62751          char *bufp;
62752
62753          if (icmp_len < 8)
62754          {
62755  #if DEBUG
62756           { where(); printf("router advertisement with wrong size (%d)\n", icmp_len); }
62757  #endif
62758                  return;
62759          }
62760          if (icmp_hdr->ih_code != 0)
62761          {
62762  #if DEBUG
62763           { where(); printf("router advertisement with wrong code (%d)\n",
62764                                                           icmp_hdr->ih_code); }
62765  #endif
62766                  return;
62767          }
62768          entries= icmp_hdr->ih_hun.ihh_ram.iram_na;
62769          entry_size= icmp_hdr->ih_hun.ihh_ram.iram_aes * 4;
62770          if (entries < 1)
62771          {
62772  #if DEBUG
62773           { where(); printf("router advertisement with wrong number of entries (%d)\n",
62774                                                           entries); }
62775  #endif
62776                  return;
62777          }
62778          if (entry_size < 8)
62779          {
62780  #if DEBUG
62781           { where(); printf("router advertisement with wrong entry size (%d)\n",
62782                                                           entry_size); }
62783  #endif
62784                  return;
62785          }
62786          if (icmp_len < 8 + entries * entry_size)
62787          {
62788  #if DEBUG
62789           { where(); printf("router advertisement with wrong size\n");
62790                 printf("\t(entries= %d, entry_size= %d, icmp_len= %d)\n", entries,
62791                                                   entry_size, icmp_len); }
62792  #endif
62793                  return;
62794          }
62795          lifetime= ntohs(icmp_hdr->ih_hun.ihh_ram.iram_lt);
62796          if (lifetime > 9000)
62797          {
62798  #if DEBUG
62799           { where(); printf("router advertisement with wrong lifetime (%d)\n",
```

```
62800                                                                   lifetime); }
62801   #endif
62802                   return;
62803           }
62804           for (i= 0, bufp= (char *)&icmp_hdr->ih_dun.uhd_data[0]; i< entries; i++,
62805                   bufp += entry_size)
62806           {
62807                   ipr_add_route(HTONL(0L), HTONL(0L), *(ipaddr_t *)bufp,
62808                           icmp_port->icp_ipport, lifetime * HZ, 1, 0,
62809                           ntohl(*(i32_t *)(bufp+4)));
62810           }
62811   }
62812
62813   static void icmp_redirect(icmp_port, ip_hdr, icmp_pack, icmp_len, icmp_hdr)
62814   icmp_port_t *icmp_port;
62815   ip_hdr_t *ip_hdr;
62816   acc_t *icmp_pack;
62817   int icmp_len;
62818   icmp_hdr_t *icmp_hdr;
62819   {
62820           acc_t *old_ip_pack;
62821           ip_hdr_t *old_ip_hdr;
62822           int port;
62823
62824           if (icmp_len < 8 + IP_MIN_HDR_SIZE)
62825           {
62826   #if DEBUG
62827            { where(); printf("redirect with wrong size\n"); }
62828   #endif
62829                   return;
62830           }
62831           old_ip_pack= bf_cut (icmp_pack, 8, icmp_len-8);
62832           old_ip_pack= bf_packIffLess(old_ip_pack, IP_MIN_HDR_SIZE);
62833           old_ip_hdr= (ip_hdr_t *)ptr2acc_data(old_ip_pack);
62834
62835           port= icmp_port->icp_ipport;
62836
62837           switch(icmp_hdr->ih_code)
62838           {
62839           case ICMP_REDIRECT_NET:
62840                   ipr_redirect (old_ip_hdr->ih_dst,
62841                           ip_get_netmask(old_ip_hdr->ih_dst),
62842                           ip_hdr->ih_src, icmp_hdr->ih_hun.ihh_gateway, port,
62843                           IPR_REDIRECT_TIMEOUT);
62844                   break;
62845           case ICMP_REDIRECT_HOST:
62846                   ipr_redirect (old_ip_hdr->ih_dst, (ipaddr_t)-1,
62847                           ip_hdr->ih_src, icmp_hdr->ih_hun.ihh_gateway, port,
62848                           IPR_REDIRECT_TIMEOUT);
62849                   break;
62850           default:
62851                   where();
62852                   printf("got strange code: %d\n", icmp_hdr->ih_code);
62853                   break;
62854           }
62855           bf_afree(old_ip_pack);
62856   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/io.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

62900    /*
62901    io.c
62902    */
62903
62904    #include <stdlib.h>
62905
62906    #include "inet.h"
62907    #include "io.h"
62908
62909    PUBLIC void writeIpAddr(addr)
62910    ipaddr_t addr;
62911    {
62912            u8_t *addrInBytes;
62913
62914            addrInBytes= (u8_t *)&addr;
62915            printf("%d.%d.%d.%d", addrInBytes[0], addrInBytes[1],
62916                    addrInBytes[2], addrInBytes[3]);
62917    }

62919    PUBLIC void writeEtherAddr(addr)
62920    ether_addr_t *addr;
62921    {
62922            u8_t *addrInBytes;
62923
62924            addrInBytes= (u8_t *)(addr->ea_addr);
62925            printf("%x:%x:%x:%x:%x:%x", addrInBytes[0], addrInBytes[1],
62926                    addrInBytes[2], addrInBytes[3], addrInBytes[4], addrInBytes[5]);
62927    }



++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/ip.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

63000    /*
63001    ip.c
63002    */
63003
63004    #include "inet.h"
63005    #include "buf.h"
63006    #include "type.h"
63007
63008    #include "arp.h"
63009    #include "assert.h"
63010    #include "clock.h"
63011    #include "eth.h"
63012    #include "icmp.h"
63013    #include "icmp_lib.h"
63014    #include "io.h"
63015    #include "ip.h"
63016    #include "ip_int.h"
63017    #include "ipr.h"
63018    #include "sr.h"
63019
```

```
63020   INIT_PANIC();
63021
63022   FORWARD void ip_eth_main ARGS(( ip_port_t *port ));
63023   FORWARD void ip_close ARGS(( int fd ));
63024   FORWARD int ip_cancel ARGS(( int fd, int which_operation ));
63025   FORWARD acc_t *get_eth_data ARGS(( int fd, size_t offset,
63026           size_t count, int for_ioctl ));
63027   FORWARD int put_eth_data ARGS(( int fd, size_t offset,
63028           acc_t *data, int for_ioctl ));
63029   FORWARD void rarp_func ARGS(( int fd, ipaddr_t ipaddr ));
63030   FORWARD void do_eth_read ARGS(( ip_port_t *port ));
63031
63032   #if IP_ROUTER
63033   FORWARD void ip_route ARGS(( ip_port_t *port, acc_t *pack ));
63034   #endif /* IP_ROUTER */
63035
63036   PUBLIC ip_port_t ip_port_table[IP_PORT_NR];
63037   PUBLIC ip_fd_t ip_fd_table[IP_FD_NR];
63038   PUBLIC ip_ass_t ip_ass_table[IP_ASS_NR];
63039
63040
63041   PRIVATE int ip_cancel (fd, which_operation)
63042   int fd;
63043   int which_operation;
63044   {
63045           ip_fd_t *ip_fd;
63046           acc_t *repl_res;
63047           int result;
63048
63049           ip_fd= &ip_fd_table[fd];
63050
63051           switch (which_operation)
63052           {
63053           case SR_CANCEL_IOCTL:
63054   assert (ip_fd->if_flags & IFF_GIPCONF_IP);
63055                   ip_fd->if_flags &= ~IFF_GIPCONF_IP;
63056                   repl_res= (*ip_fd->if_get_userdata)(ip_fd->if_srfd,
63057                           (size_t)EINTR, (size_t)0, TRUE);
63058   assert (!repl_res);
63059                   break;
63060           case SR_CANCEL_READ:
63061   assert (ip_fd->if_flags & IFF_READ_IP);
63062                   ip_fd->if_flags &= ~IFF_READ_IP;
63063                   result= (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
63064                           (size_t)EINTR, (acc_t *)0, FALSE);
63065   assert (!result);
63066                   break;
63067           case SR_CANCEL_WRITE:
63068   assert (ip_fd->if_flags & IFF_WRITE_MASK);
63069                   ip_fd->if_flags &= ~IFF_WRITE_MASK;
63070                   repl_res= (*ip_fd->if_get_userdata)(ip_fd->if_srfd,
63071                           (size_t)EINTR, (size_t)0, FALSE);
63072   assert (!repl_res);
63073                   break;
63074           default:
63075                   ip_panic(( "unknown cancel request" ));
63076                   break;
63077           }
63078           return NW_OK;
63079   }
```

```
63082   PUBLIC void ip_init()
63083   {
63084           int i, result;
63085           ip_ass_t *ip_ass;
63086           ip_fd_t *ip_fd;
63087           ip_port_t *ip_port;
63088
63089           assert (BUF_S >= sizeof(struct nwio_ethopt));
63090           assert (BUF_S >= IP_MAX_HDR_SIZE + ETH_HDR_SIZE);
63091           assert (BUF_S >= sizeof(nwio_ipopt_t));
63092           assert (BUF_S >= sizeof(nwio_route_t));
63093
63094           ip_port_table[0].ip_dl.dl_eth.de_port= ETH0;
63095           ip_port_table[0].ip_dl_type= IPDL_ETH;
63096           ip_port_table[0].ip_minor= IP_DEV0;
63097
63098           for (i=0, ip_ass= ip_ass_table; i<IP_ASS_NR; i++, ip_ass++)
63099           {
63100                   ip_ass->ia_frags= 0;
63101                   ip_ass->ia_first_time= 0;
63102                   ip_ass->ia_port= 0;
63103           }
63104
63105           for (i=0, ip_fd= ip_fd_table; i<IP_FD_NR; i++, ip_fd++)
63106           {
63107                   ip_fd->if_flags= IFF_EMPTY;
63108           }
63109
63110           for (i=0, ip_port= ip_port_table; i<IP_PORT_NR; i++, ip_port++)
63111           {
63112                   ip_port->ip_flags= IPF_EMPTY;
63113                   switch(ip_port->ip_dl_type)
63114                   {
63115                   case IPDL_ETH:
63116                           ip_port->ip_dl.dl_eth.de_state= IES_EMPTY;
63117                           ip_port->ip_dl.dl_eth.de_flags= IEF_EMPTY;
63118                           break;
63119                   default:
63120                           ip_panic(( "unknown ip_dl_type" ));
63121                           break;
63122                   }
63123           }
63124
63125           icmp_init();
63126           ipr_init();
63127
63128           for (i=0, ip_port= ip_port_table; i<IP_PORT_NR; i++, ip_port++)
63129           {
63130                   ip_port->ip_frame_id= (u16_t)get_time();
63131
63132                   result= sr_add_minor(ip_port->ip_minor,
63133                           ip_port-ip_port_table, ip_open, ip_close,
63134                           ip_read, ip_write, ip_ioctl, ip_cancel);
63135                   assert (result>=0);
63136
63137                   switch(ip_port->ip_dl_type)
63138                   {
63139                   case IPDL_ETH:
```

```
63140                           ip_eth_main(ip_port);
63141                           break;
63142                   default:
63143                           ip_panic(( "unknown ip_dl_type" ));
63144                   }
63145           }
63146   }

63148   PRIVATE void ip_eth_main(ip_port)
63149   ip_port_t *ip_port;
63150   {
63151           int result, i;
63152           ip_fd_t *ip_fd;
63153
63154           switch (ip_port->ip_dl.dl_eth.de_state)
63155           {
63156           case IES_EMPTY:
63157                   ip_port->ip_dl.dl_eth.de_wr_ipaddr= (ipaddr_t)0;
63158                   ip_port->ip_dl.dl_eth.de_state= IES_SETPROTO;
63159                   ip_port->ip_dl.dl_eth.de_fd= eth_open(ip_port->
63160                           ip_dl.dl_eth.de_port, ip_port-ip_port_table,
63161                           get_eth_data, put_eth_data);
63162                   if (ip_port->ip_dl.dl_eth.de_fd < 0)
63163                   {
63164                           printf("ip.c: unable to open eth port\n");
63165                           return;
63166                   }
63167
63168                   result= eth_ioctl(ip_port->ip_dl.dl_eth.de_fd,
63169                           NWIOSETHOPT);
63170                   if (result == NW_SUSPEND)
63171                           ip_port->ip_dl.dl_eth.de_flags |= IEF_SUSPEND;
63172                   if (result<0)
63173                   {
63174   #if DEBUG
63175    { where(); printf("eth_ioctl(..,%lx)=%d\n", NWIOSETHOPT, result); }
63176   #endif
63177                           return;
63178                   }
63179                   if (ip_port->ip_dl.dl_eth.de_state != IES_SETPROTO)
63180                           return;
63181                   /* drops through */
63182           case IES_SETPROTO:
63183                   ip_port->ip_dl.dl_eth.de_state= IES_GETIPADDR;
63184
63185                   result= rarp_req (ip_port->ip_dl.dl_eth.de_port,
63186                           ip_port-ip_port_table, rarp_func);
63187
63188                   if (result == NW_SUSPEND)
63189                           ip_port->ip_dl.dl_eth.de_flags |= IEF_SUSPEND;
63190                   if (result<0)
63191                   {
63192   #if DEBUG & 256
63193    { where(); printf("rarp_req(...)=%d\n", result); }
63194   #endif
63195                           return;
63196                   }
63197                   if (ip_port->ip_dl.dl_eth.de_state != IES_GETIPADDR)
63198                           return;
63199                   /* drops through */
```

```
63200           case IES_GETIPADDR:
63201                   ip_port->ip_dl.dl_eth.de_state= IES_MAIN;
63202                   for (i=0, ip_fd= ip_fd_table; i<IP_FD_NR; i++, ip_fd++)
63203                   {
63204                           if (!(ip_fd->if_flags & IFF_INUSE))
63205                           {
63206   #if DEBUG & 256
63207    { where(); printf("%d not inuse\n", i); }
63208   #endif
63209                                   continue;
63210                           }
63211                           if (ip_fd->if_port != ip_port)
63212                           {
63213   #if DEBUG
63214    { where(); printf("%d wrong port\n", i); }
63215   #endif
63216                                   continue;
63217                           }
63218   #if DEBUG & 256
63219    { where(); printf("(ip_fd_t *)0x%x->if_flags= 0x%x\n", ip_fd,
63220    ip_fd->if_flags); }
63221   #endif
63222                           if (ip_fd->if_flags & IFF_WRITE_IP)
63223                           {
63224   #if DEBUG
63225    { where(); printf("%d write ip\n", i); }
63226   #endif
63227                                   ip_fd->if_flags &= ~IFF_WRITE_IP;
63228                                   ip_write (i, ip_fd->if_wr_count);
63229                           }
63230                           if (ip_fd->if_flags & IFF_GIPCONF_IP)
63231                           {
63232   #if DEBUG  & 256
63233    { where(); printf("restarting ip_ioctl (.., NWIOGIPCONF)\n"); }
63234   #endif
63235                                   ip_ioctl (i, NWIOGIPCONF);
63236                           }
63237                   }
63238   #if DEBUG & 256
63239    { where(); printf("ip_port->ip_ipaddr= "); writeIpAddr(ip_port->ip_ipaddr);
63240           printf("\n"); }
63241   #endif
63242                   icmp_getnetmask(ip_port-ip_port_table);
63243                   do_eth_read(ip_port);
63244                   return;
63245           default:
63246                   ip_panic(( "unknown state" ));
63247           }
63248   }

63250   PRIVATE acc_t *get_eth_data (fd, offset, count, for_ioctl)
63251   int fd;
63252   size_t offset;
63253   size_t count;
63254   int for_ioctl;
63255   {
63256           ip_port_t *port;
63257           acc_t *data;
63258           int result;
63259
```

```
63260   #if DEBUG & 256
63261    { where(); printf("get_eth_data(fd= %d, offset= %d, count= %u) called\n",
63262                   fd, offset, count); }
63263   #endif
63264           port= &ip_port_table[fd];
63265
63266           switch (port->ip_dl.dl_eth.de_state)
63267           {
63268           case IES_SETPROTO:
63269                   if (!count)
63270                   {
63271                           result= (int)offset;
63272                           if (result<0)
63273                           {
63274                                   port->ip_dl.dl_eth.de_state= IES_ERROR;
63275                                   break;
63276                           }
63277                           if (port->ip_dl.dl_eth.de_flags & IEF_SUSPEND)
63278                                   ip_eth_main(port);
63279                           return NW_OK;
63280                   }
63281                   assert ((!offset) && (count == sizeof(struct nwio_ethopt)));
63282                   {
63283                           struct nwio_ethopt *ethopt;
63284                           acc_t *acc;
63285
63286                           acc= bf_memreq(sizeof(*ethopt));
63287                           ethopt= (struct nwio_ethopt *)ptr2acc_data(acc);
63288                           ethopt->nweo_flags= NWEO_COPY|NWEO_EN_BROAD|
63289                                   NWEO_TYPESPEC;
63290                           ethopt->nweo_type= htons(ETH_IP_PROTO);
63291                           return acc;
63292                   }
63293
63294           case IES_MAIN:
63295                   assert (port->ip_dl.dl_eth.de_flags & IEF_WRITE_IP);
63296                   if (!count)
63297                   {
63298
63299                           result= (int)offset;
63300   #if DEBUG & 256
63301    { where(); printf("get_eth_data: result= %d\n", result); }
63302   #endif
63303                           if (result<0)
63304                                   printf("ip.c: error on write: %d\n",
63305                                           result);
63306                           bf_afree (port->ip_dl.dl_eth.de_wr_frame);
63307                           port->ip_dl.dl_eth.de_wr_frame= 0;
63308   #if DEBUG & 256
63309    { where(); printf("eth_write completed\n"); }
63310   #endif
63311                           if (port->ip_dl.dl_eth.de_flags & IEF_WRITE_SP)
63312                           {
63313   #if DEBUG & 256
63314    { where(); printf("calling dl_eth_write_frame\n"); }
63315   #endif
63316                                   port->ip_dl.dl_eth.de_flags &=
63317                                           ~(IEF_WRITE_SP|IEF_WRITE_IP);
63318                                   dll_eth_write_frame(port);
63319                           }
```

```
63320	#if DEBUG & 256
63321	 else { where(); printf("not calling dl_eth_write_frame\n"); }
63322	#endif
63323				return NW_OK;
63324			}
63325	#if DEBUG & 256
63326	 { where(); printf("supplying data for eth\n"); }
63327	#endif
63328			data= bf_cut (port->ip_dl.dl_eth.de_wr_frame, offset,
63329				count);
63330			assert (data);
63331			return data;
63332
63333		default:
63334			printf("get_eth_data(%d, 0x%d, 0x%d) called but ip_state=0x%x\n",
63335				fd, offset, count, port->ip_dl.dl_eth.de_state);
63336			break;
63337		}
63338		return 0;
63339	}

63341	PRIVATE void rarp_func (port, ipaddr)
63342	int port;
63343	ipaddr_t ipaddr;
63344	{
63345		ip_port_t *ip_port;
63346
63347	#if DEBUG & 256
63348	 { where(); printf("rarp_func\n"); }
63349	#endif
63350		ip_port= &ip_port_table[port];
63351
63352		if (!(ip_port->ip_flags & IPF_IPADDRSET))
63353		{
63354			ip_port->ip_ipaddr= ipaddr;
63355			ip_port->ip_flags |= IPF_IPADDRSET;
63356		}
63357		switch (ip_port->ip_dl_type)
63358		{
63359		case IPDL_ETH:
63360			if (ip_port->ip_dl.dl_eth.de_flags & IEF_SUSPEND)
63361				ip_eth_main(ip_port);
63362			break;
63363		default:
63364			ip_panic(( "unknown dl_type" ));
63365		}
63366	}

63368	PRIVATE void do_eth_read(port)
63369	ip_port_t *port;
63370	{
63371		int result;
63372
63373		while (!(port->ip_dl.dl_eth.de_flags & IEF_READ_IP))
63374		{
63375			port->ip_dl.dl_eth.de_flags &= ~IEF_READ_SP;
63376			port->ip_dl.dl_eth.de_flags |= IEF_READ_IP;
63377			result= eth_read (port->ip_dl.dl_eth.de_fd,
63378				ETH_MAX_PACK_SIZE);
63379			if (result == NW_SUSPEND)
```

```
63380                           port->ip_dl.dl_eth.de_flags |= IEF_READ_SP;
63381                   if (result<0)
63382                   {
63383   #if DEBUG & 256
63384    { where(); printf("eth_read(%d, ...)= %d\n", port->ip_dl.dl_eth.de_fd,
63385           result); }
63386   #endif
63387                           return;
63388                   }
63389                   port->ip_dl.dl_eth.de_flags &= ~IEF_READ_IP;
63390           }
63391   }

63393   PRIVATE int put_eth_data (port, offset, data, for_ioctl)
63394   int port;
63395   size_t offset;
63396   acc_t *data;
63397   int for_ioctl;
63398   {
63399           ip_port_t *ip_port;
63400           acc_t *pack;
63401           int result;
63402
63403   #if DEBUG & 256
63404    { where(); printf("put_eth_data() called\n"); }
63405   #endif
63406           ip_port= &ip_port_table[port];
63407
63408           if (ip_port->ip_dl.dl_eth.de_flags & IEF_READ_IP)
63409           {
63410                   if (!data)
63411                   {
63412                           result= (int)offset;
63413                           if (result<0)
63414                           {
63415   #if DEBUG
63416    { where(); printf("ip.c: put_eth_data(..,%d,..)\n", result); }
63417   #endif
63418                                   return NW_OK;
63419                           }
63420                           ip_port->ip_dl.dl_eth.de_flags &= ~IEF_READ_IP;
63421                           if (ip_port->ip_dl.dl_eth.de_flags &
63422                                   IEF_READ_SP)
63423                           {
63424                                   do_eth_read(ip_port);
63425                           }
63426                           return NW_OK;
63427                   }
63428                   assert (!offset);
63429                   /* Warning: the above assertion is illegal; puts and
63430                      gets of data can be brokenup in any piece the server
63431                      likes. However we assume that the server is eth.c
63432                      and it transfers only whole packets. */
63433                   ip_eth_arrived(ip_port, data);
63434                   return NW_OK;
63435           }
63436           printf("ip_port->ip_dl.dl_eth.de_state= 0x%x",
63437                   ip_port->ip_dl.dl_eth.de_state);
63438           ip_panic (( "strange status" ));
63439   }
```

```
63441   PUBLIC int ip_open (port, srfd, get_userdata, put_userdata)
63442   int port;
63443   int srfd;
63444   get_userdata_t get_userdata;
63445   put_userdata_t put_userdata;
63446   {
63447           int i;
63448           ip_fd_t *ip_fd;
63449
63450           for (i=0; i<IP_FD_NR && (ip_fd_table[i].if_flags & IFF_INUSE);
63451                   i++);
63452
63453           if (i>=IP_FD_NR)
63454           {
63455   #if DEBUG
63456    { where(); printf("out of fds\n"); }
63457   #endif
63458                   return EOUTOFBUFS;
63459           }
63460
63461           ip_fd= &ip_fd_table[i];
63462
63463           ip_fd->if_flags= IFF_INUSE;
63464
63465           ip_fd->if_ipopt.nwio_flags= NWIO_DEFAULT;
63466           ip_fd->if_ipopt.nwio_tos= 0;
63467           ip_fd->if_ipopt.nwio_df= FALSE;
63468           ip_fd->if_ipopt.nwio_ttl= 255;
63469           ip_fd->if_ipopt.nwio_hdropt.iho_opt_siz= 0;
63470
63471           ip_fd->if_port= &ip_port_table[port];
63472           ip_fd->if_srfd= srfd;
63473           ip_fd->if_rd_buf= 0;
63474           ip_fd->if_get_userdata= get_userdata;
63475           ip_fd->if_put_userdata= put_userdata;
63476           return i;
63477   }

63479   PRIVATE void ip_close (fd)
63480   int fd;
63481   {
63482           ip_fd_t *ip_fd;
63483
63484           ip_fd= &ip_fd_table[fd];
63485
63486           assert ((ip_fd->if_flags & IFF_INUSE) &&
63487                   !(ip_fd->if_flags & IFF_BUSY));
63488
63489           ip_fd->if_flags= IFF_EMPTY;
63490           if (ip_fd->if_rd_buf)
63491           {
63492                   bf_afree(ip_fd->if_rd_buf);
63493                   ip_fd->if_rd_buf= 0;
63494           }
63495   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/ip_ioctl.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

63500    /*
63501    ip_ioctl.c
63502    */
63503
63504    #include "inet.h"
63505    #include "buf.h"
63506    #include "type.h"
63507
63508    #include "arp.h"
63509    #include "assert.h"
63510    #include "clock.h"
63511    #include "icmp_lib.h"
63512    #include "ip.h"
63513    #include "ip_int.h"
63514    #include "ipr.h"
63515
63516    INIT_PANIC();
63517
63518    FORWARD int ip_checkopt ARGS(( ip_fd_t *ip_fd ));
63519    FORWARD void reply_thr_get ARGS(( ip_fd_t *ip_fd, size_t
63520            reply, int for_ioctl ));
63521
63522    PUBLIC int ip_ioctl (fd, req)
63523    int fd;
63524    int req;
63525    {
63526            ip_fd_t *ip_fd;
63527            int type;
63528
63529    assert (fd>=0 && fd<=IP_FD_NR);
63530            ip_fd= &ip_fd_table[fd];
63531            type= req & IOCTYPE_MASK;
63532
63533            assert (ip_fd->if_flags & IFF_INUSE);
63534
63535            switch (type)
63536            {
63537            case NWIOSIPOPT & IOCTYPE_MASK:
63538                    {
63539                            nwio_ipopt_t *ipopt;
63540                            nwio_ipopt_t oldopt, newopt;
63541                            acc_t *data;
63542                            int result;
63543                            unsigned int new_en_flags, new_di_flags,
63544                                    old_en_flags, old_di_flags;
63545                            unsigned long new_flags;
63546
63547                            if (req != NWIOSIPOPT)
63548                                    break;
63549
63550                            data= (*ip_fd->if_get_userdata)(ip_fd->if_srfd, 0,
63551                                    sizeof(nwio_ipopt_t), TRUE);
63552
63553                            data= bf_packIffLess (data, sizeof(nwio_ipopt_t));
63554                            assert (data->acc_length == sizeof(nwio_ipopt_t));
```

```
63555
63556                   ipopt= (nwio_ipopt_t *)ptr2acc_data(data);
63557                   oldopt= ip_fd->if_ipopt;
63558                   newopt= *ipopt;
63559
63560                   old_en_flags= oldopt.nwio_flags & 0xffff;
63561                   old_di_flags= (oldopt.nwio_flags >> 16) & 0xffff;
63562                   new_en_flags= newopt.nwio_flags & 0xffff;
63563                   new_di_flags= (newopt.nwio_flags >> 16) &
63564                           0xffff;
63565                   if (new_en_flags & new_di_flags)
63566                   {
63567                           reply_thr_get (ip_fd, EBADMODE, TRUE);
63568                           return NW_OK;
63569                   }
63570
63571                   /* NWIO_ACC_MASK */
63572                   if (new_di_flags & NWIO_ACC_MASK)
63573                   {
63574                           reply_thr_get (ip_fd, EBADMODE, TRUE);
63575                           return NW_OK;
63576                           /* you can't disable access modes */
63577                   }
63578
63579                   if (!(new_en_flags & NWIO_ACC_MASK))
63580                           new_en_flags |= (old_en_flags &
63581                                   NWIO_ACC_MASK);
63582
63583                   /* NWIO_LOC_MASK */
63584                   if (!((new_en_flags | new_di_flags) &
63585                           NWIO_LOC_MASK))
63586                   {
63587                           new_en_flags |= (old_en_flags &
63588                                   NWIO_LOC_MASK);
63589                           new_di_flags |= (old_di_flags &
63590                                   NWIO_LOC_MASK);
63591                   }
63592
63593                   /* NWIO_BROAD_MASK */
63594                   if (!((new_en_flags | new_di_flags) &
63595                           NWIO_BROAD_MASK))
63596                   {
63597                           new_en_flags |= (old_en_flags &
63598                                   NWIO_BROAD_MASK);
63599                           new_di_flags |= (old_di_flags &
63600                                   NWIO_BROAD_MASK);
63601                   }
63602
63603                   /* NWIO_REM_MASK */
63604                   if (!((new_en_flags | new_di_flags) &
63605                           NWIO_REM_MASK))
63606                   {
63607                           new_en_flags |= (old_en_flags &
63608                                   NWIO_REM_MASK);
63609                           new_di_flags |= (old_di_flags &
63610                                   NWIO_REM_MASK);
63611                           newopt.nwio_rem= oldopt.nwio_rem;
63612                   }
63613
63614                   /* NWIO_PROTO_MASK */
```

```
63615                           if (!((new_en_flags | new_di_flags) &
63616                                   NWIO_PROTO_MASK))
63617                           {
63618                                   new_en_flags |= (old_en_flags &
63619                                           NWIO_PROTO_MASK);
63620                                   new_di_flags |= (old_di_flags &
63621                                           NWIO_PROTO_MASK);
63622                                   newopt.nwio_proto= oldopt.nwio_proto;
63623                           }
63624
63625                           /* NWIO_HDR_O_MASK */
63626                           if (!((new_en_flags | new_di_flags) &
63627                                   NWIO_HDR_O_MASK))
63628                           {
63629                                   new_en_flags |= (old_en_flags &
63630                                           NWIO_HDR_O_MASK);
63631                                   new_di_flags |= (old_di_flags &
63632                                           NWIO_HDR_O_MASK);
63633                                   newopt.nwio_tos= oldopt.nwio_tos;
63634                                   newopt.nwio_ttl= oldopt.nwio_ttl;
63635                                   newopt.nwio_df= oldopt.nwio_df;
63636                                   newopt.nwio_hdropt= oldopt.nwio_hdropt;
63637                           }
63638
63639                           /* NWIO_RW_MASK */
63640                           if (!((new_en_flags | new_di_flags) &
63641                                   NWIO_RW_MASK))
63642                           {
63643                                   new_en_flags |= (old_en_flags &
63644                                           NWIO_RW_MASK);
63645                                   new_di_flags |= (old_di_flags &
63646                                           NWIO_RW_MASK);
63647                           }
63648
63649                           new_flags= ((unsigned long)new_di_flags << 16) |
63650                                   new_en_flags;
63651
63652                           if ((new_flags & NWIO_RWDATONLY) && (new_flags &
63653                                   (NWIO_REMANY|NWIO_PROTOANY|NWIO_HDR_O_ANY)))
63654                           {
63655                                   reply_thr_get(ip_fd, EBADMODE, TRUE);
63656                                   return NW_OK;
63657                           }
63658
63659                           newopt.nwio_flags= new_flags;
63660                           ip_fd->if_ipopt= newopt;
63661
63662                           result= ip_checkopt(ip_fd);
63663
63664                           if (result<0)
63665                                   ip_fd->if_ipopt= oldopt;
63666
63667                           bf_afree(data);
63668                           reply_thr_get (ip_fd, result, TRUE);
63669                           return NW_OK;
63670                   }
63671
63672           case NWIOGIPOPT & IOCTYPE_MASK:
63673                   {
63674                           nwio_ipopt_t *ipopt;
```

```
63675                           acc_t *acc;
63676                           int result;
63677
63678                           if (req != NWIOGIPOPT)
63679                                   break;
63680                           acc= bf_memreq(sizeof(nwio_ipopt_t));
63681
63682                           ipopt= (nwio_ipopt_t *)ptr2acc_data(acc);
63683
63684                           *ipopt= ip_fd->if_ipopt;
63685
63686                           result= (*ip_fd->if_put_userdata)(ip_fd->
63687                                   if_srfd, 0, acc, TRUE);
63688                           return (*ip_fd->if_put_userdata)(ip_fd->
63689                                   if_srfd, result, (acc_t *)0, TRUE);
63690                   }
63691           case NWIOSIPCONF & IOCTYPE_MASK:
63692                   {
63693                           nwio_ipconf_t *ipconf;
63694                           ip_port_t *ip_port;
63695                           acc_t *data;
63696                           int old_ip_flags;
63697
63698                           ip_port= ip_fd->if_port;
63699
63700                           if (req != NWIOSIPCONF)
63701                                   break;
63702
63703                           data= (*ip_fd->if_get_userdata)(ip_fd->if_srfd,
63704                                   0, sizeof(nwio_ipconf_t), TRUE);
63705
63706                           data= bf_packIffLess (data,
63707                                   sizeof(nwio_ipconf_t));
63708                           assert (data->acc_length == sizeof(nwio_ipconf_t));
63709
63710                           old_ip_flags= ip_port->ip_flags;
63711
63712                           ipconf= (nwio_ipconf_t *)ptr2acc_data(data);
63713
63714                           if (ipconf->nwic_flags & ~NWIC_FLAGS)
63715                                   return (*ip_fd->if_put_userdata)(ip_fd->
63716                                           if_srfd, EBADMODE, (acc_t *)0, TRUE);
63717                           if (ipconf->nwic_flags & NWIC_IPADDR_SET)
63718                           {
63719                                   ip_port->ip_ipaddr= ipconf->nwic_ipaddr;
63720                                   ip_port->ip_flags |= IPF_IPADDRSET;
63721                                   switch (ip_port->ip_dl_type)
63722                                   {
63723                                   case IPDL_ETH:
63724                                           set_ipaddr (ip_port->ip_dl.
63725                                                   dl_eth.de_port,
63726                                                   ip_port->ip_ipaddr);
63727                                           break;
63728                                   default:
63729                                           ip_panic(( "unknown dl_type" ));
63730                                   }
63731                           }
63732                           if (ipconf->nwic_flags & NWIC_NETMASK_SET)
63733                           {
63734                                   ip_port->ip_netmask=
```

```
63735                                           ipconf->nwic_netmask;
63736                                   ip_port->ip_flags |= IPF_NETMASKSET;
63737                           }
63738                           if (!(old_ip_flags & IPF_IPADDRSET) &&
63739                                   (ip_port->ip_flags & IPF_IPADDRSET) &&
63740                                   !(ip_port->ip_flags & IPF_NETMASKSET))
63741                           {
63742                                   icmp_getnetmask(ip_port-ip_port_table);
63743                           }
63744
63745                           bf_afree(data);
63746                           return (*ip_fd->if_put_userdata)(ip_fd->
63747                                   if_srfd, NW_OK, (acc_t *)0, TRUE);
63748                   }
63749           case NWIOGIPCONF & IOCTYPE_MASK:
63750                   {
63751                           nwio_ipconf_t *ipconf;
63752                           ip_port_t *ip_port;
63753                           acc_t *data;
63754                           int result;
63755
63756                           ip_port= ip_fd->if_port;
63757
63758                           if (req != NWIOGIPCONF)
63759                                   break;
63760
63761                           if (!(ip_port->ip_flags & IPF_IPADDRSET))
63762                           {
63763                                   ip_fd->if_flags |= IFF_GIPCONF_IP;
63764   #if DEBUG & 256
63765    { where(); printf("(ip_fd_t *)0x%x->if_flags= 0x%x\n", ip_fd,
63766    ip_fd->if_flags); }
63767   #endif
63768                                   return NW_SUSPEND;
63769                           }
63770                           ip_fd->if_flags &= ~IFF_GIPCONF_IP;
63771                           data= bf_memreq(sizeof(nwio_ipconf_t));
63772                           ipconf= (nwio_ipconf_t *)ptr2acc_data(data);
63773                           ipconf->nwic_flags= NWIC_IPADDR_SET;
63774                           ipconf->nwic_ipaddr= ip_port->ip_ipaddr;
63775                           ipconf->nwic_netmask= ip_port->ip_netmask;
63776                           if (ip_port->ip_flags & IPF_NETMASKSET)
63777                                   ipconf->nwic_flags |= NWIC_NETMASK_SET;
63778
63779                           result= (*ip_fd->if_put_userdata)(ip_fd->
63780                                   if_srfd, 0, data, TRUE);
63781                           return (*ip_fd->if_put_userdata)(ip_fd->
63782                                   if_srfd, result, (acc_t *)0, TRUE);
63783                   }
63784           case NWIOIPGROUTE & IOCTYPE_MASK:
63785                   {
63786                           acc_t *data;
63787                           nwio_route_t *route_ent;
63788                           int result;
63789
63790                           if (req != NWIOIPGROUTE)
63791                                   break;
63792
63793                           data= (*ip_fd->if_get_userdata)(ip_fd->if_srfd,
63794                                   0, sizeof(nwio_route_t), TRUE);
```

```
63795
63796                           data= bf_packIffLess (data, sizeof(nwio_route_t) );
63797                           route_ent= (nwio_route_t *)ptr2acc_data(data);
63798
63799                           result= ipr_get_route(route_ent->nwr_ent_no, route_ent);
63800                           if (result>=0)
63801                                   (*ip_fd->if_put_userdata)(ip_fd->if_srfd, 0,
63802                                           data, TRUE);
63803                           return (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
63804                                   result, (acc_t *)0, TRUE);
63805                   }
63806           case NWIOIPSROUTE & IOCTYPE_MASK:
63807                   {
63808                           acc_t *data;
63809                           nwio_route_t *route_ent;
63810                           route_t *route;
63811                           int result;
63812
63813                           if (req != NWIOIPSROUTE)
63814                                   break;
63815
63816                           data= (*ip_fd->if_get_userdata)(ip_fd->if_srfd,
63817                                   0, sizeof(nwio_route_t), TRUE);
63818
63819                           data= bf_packIffLess (data, sizeof(nwio_route_t) );
63820                           route_ent= (nwio_route_t *)ptr2acc_data(data);
63821                           route= ipr_add_route(route_ent->nwr_dest,
63822                                   route_ent->nwr_netmask, route_ent->nwr_gateway,
63823                                   ip_fd->if_port-ip_port_table, (time_t)0,
63824                                   route_ent->nwr_dist, !!(route_ent->nwr_flags &
63825                                           NWRF_FIXED), route_ent->nwr_pref);
63826                           bf_afree(data);
63827                           if (route)
63828                                   result= NW_OK;
63829                           else
63830                           {
63831   #if DEBUG
63832    { where(); printf("out of routing table entries\n"); }
63833   #endif
63834                                   result= ENOMEM;
63835                           }
63836                           return (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
63837                                   result, (acc_t *)0, TRUE);
63838                   }
63839           default:
63840                   break;
63841           }
63842   #if DEBUG
63843    { where(); printf("replying EBADIOCTL\n"); }
63844   #endif
63845           return (*ip_fd->if_put_userdata)(ip_fd-> if_srfd, EBADIOCTL,
63846                   (acc_t *)0, TRUE);
63847   }

63849   PRIVATE int ip_checkopt (ip_fd)
63850   ip_fd_t *ip_fd;
63851   {
63852   /* bug: we don't check access modes yet */
63853
63854           unsigned long flags;
```

```
63855           unsigned int en_di_flags;
63856           ip_port_t *port;
63857           int result;
63858
63859           flags= ip_fd->if_ipopt.nwio_flags;
63860           en_di_flags= (flags >>16) | (flags & 0xffff);
63861
63862           if (flags & NWIO_HDR_O_SPEC)
63863           {
63864                   result= ip_chk_hdropt (ip_fd->if_ipopt.nwio_hdropt.iho_data,
63865                           ip_fd->if_ipopt.nwio_hdropt.iho_opt_siz);
63866                   if (result<0)
63867                           return result;
63868           }
63869           if ((en_di_flags & NWIO_ACC_MASK) &&
63870                   (en_di_flags & NWIO_LOC_MASK) &&
63871                   (en_di_flags & NWIO_BROAD_MASK) &&
63872                   (en_di_flags & NWIO_REM_MASK) &&
63873                   (en_di_flags & NWIO_PROTO_MASK) &&
63874                   (en_di_flags & NWIO_HDR_O_MASK) &&
63875                   (en_di_flags & NWIO_RW_MASK))
63876           {
63877                   ip_fd->if_flags |= IFF_OPTSET;
63878
63879                   if (ip_fd->if_rd_buf)
63880                           if (get_time() > ip_fd->if_exp_tim ||
63881                                   !ip_ok_for_fd(ip_fd, ip_fd->if_rd_buf))
63882                           {
63883                                   bf_afree(ip_fd->if_rd_buf);
63884                                   ip_fd->if_rd_buf= 0;
63885                           }
63886           }
63887
63888           else
63889           {
63890                   ip_fd->if_flags &= ~IFF_OPTSET;
63891                   if (ip_fd->if_rd_buf)
63892                   {
63893                           bf_afree(ip_fd->if_rd_buf);
63894                           ip_fd->if_rd_buf= 0;
63895                   }
63896           }
63897
63898           return NW_OK;
63899   }

63901   PRIVATE void reply_thr_get(ip_fd, reply, for_ioctl)
63902   ip_fd_t *ip_fd;
63903   size_t reply;
63904   int for_ioctl;
63905   {
63906           acc_t *result;
63907           result= (ip_fd->if_get_userdata)(ip_fd->if_srfd, reply,
63908                   (size_t)0, for_ioctl);
63909           assert (!result);
63910   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/ip_lib.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

64000    /*
64001    ip_lib.c
64002    */
64003
64004    #include "inet.h"
64005    #include "buf.h"
64006    #include "type.h"
64007
64008    #include "assert.h"
64009    #include "io.h"
64010    #include "ip_int.h"
64011
64012    INIT_PANIC();
64013
64014    PUBLIC ipaddr_t ip_get_netmask (hostaddr)
64015    ipaddr_t hostaddr;
64016    {
64017            ipaddr_t host, mask;
64018
64019    #if DEBUG & 256
64020     { where(); printf("ip_lib.c: ip_get_netmask(hostaddr= ");
64021       writeIpAddr(hostaddr); printf(")\n"); }
64022    #endif
64023            host= ntohl(hostaddr);
64024            if (!(host & 0x80000000L))
64025                    mask= 0xff000000L;
64026            else if (!(host & 0x40000000L))
64027                    mask= 0xffff0000L;
64028            else if (!(host & 0x20000000L))
64029                    mask= 0xffffff00L;
64030            else
64031            {
64032    #if DEBUG
64033     { where(); printf("ip.c: marsian address: "); writeIpAddr (hostaddr);
64034       printf("\n"); }
64035    #endif
64036                    mask= 0xffffffffL;
64037            }
64038            return htonl(mask);
64039    }

64041    PUBLIC int ip_chk_hdropt (opt, optlen)
64042    u8_t *opt;
64043    int optlen;
64044    {
64045            int i, security_present= FALSE, lose_source_present= FALSE,
64046                    strict_source_present= FALSE, record_route_present= FALSE,
64047                    timestamp_present= FALSE;
64048
64049    #if DEBUG
64050     { where(); printf("ip_chk_hdropt(..., %d) called\n", optlen); }
64051    #endif
64052
64053    assert (!(optlen & 3));
64054            i= 0;
```

```
64055           while (i<optlen)
64056           {
64057   #if DEBUG
64058    { where(); printf("*opt= %d\n", *opt); }
64059   #endif
64060                   switch (*opt)
64061                   {
64062                   case 0x0:               /* End of Option list */
64063                           return NW_OK;
64064                   case 0x1:               /* No Operation */
64065                           i++;
64066                           opt++;
64067                           break;
64068                   case 0x82:              /* Security */
64069                           if (security_present)
64070                                   return EINVAL;
64071                           security_present= TRUE;
64072                           if (opt[1] != 11)
64073                                   return EINVAL;
64074                           i += opt[1];
64075                           opt += opt[1];
64076                           break;
64077                   case 0x83:              /* Lose Source and Record Route */
64078                           if (lose_source_present)
64079                           {
64080   #if DEBUG
64081    { where(); printf("snd lose soruce route\n"); }
64082   #endif
64083                                   return EINVAL;
64084                           }
64085                           lose_source_present= TRUE;
64086                           if (opt[1]<3)
64087                           {
64088   #if DEBUG
64089    { where(); printf("wrong length in source route\n"); }
64090   #endif
64091                                   return EINVAL;
64092                           }
64093                           i += opt[1];
64094                           opt += opt[1];
64095                           break;
64096                   case 0x89:              /* Strict Source and Record Route */
64097                           if (strict_source_present)
64098                                   return EINVAL;
64099                           strict_source_present= TRUE;
64100                           if (opt[1]<3)
64101                                   return EINVAL;
64102                           i += opt[1];
64103                           opt += opt[1];
64104                           break;
64105                   case 0x7:               /* Record Route */
64106                           if (record_route_present)
64107                                   return EINVAL;
64108                           record_route_present= TRUE;
64109                           if (opt[1]<3)
64110                                   return EINVAL;
64111                           i += opt[1];
64112                           opt += opt[1];
64113                           break;
64114                   case 0x88:
```

```
64115                           if (timestamp_present)
64116                                   return EINVAL;
64117                           timestamp_present= TRUE;
64118                           if (opt[1] != 4)
64119                                   return EINVAL;
64120                           switch (opt[3] & 0xff)
64121                           {
64122                           case 0:
64123                           case 1:
64124                           case 3:
64125                                   break;
64126                           default:
64127                                   return EINVAL;
64128                           }
64129                           i += opt[1];
64130                           opt += opt[1];
64131                           break;
64132                   default:
64133                           return EINVAL;
64134                   }
64135           }
64136           if (i > optlen)
64137           {
64138   #if DEBUG
64139    { where(); printf("option of wrong length\n"); }
64140   #endif
64141                   return EINVAL;
64142           }
64143           return NW_OK;
64144   }

64146   void ip_print_frags(acc)
64147   acc_t *acc;
64148   {
64149           ip_hdr_t *ip_hdr;
64150           int first;
64151
64152           if (!acc)
64153                   printf("(null)");
64154
64155           for (first= 1; acc; acc= acc->acc_ext_link, first= 0)
64156           {
64157   assert (acc->acc_length >= IP_MIN_HDR_SIZE);
64158                   ip_hdr= (ip_hdr_t *)ptr2acc_data(acc);
64159                   if (first)
64160                   {
64161                           writeIpAddr(ip_hdr->ih_src);
64162                           printf(" > ");
64163                           writeIpAddr(ip_hdr->ih_dst);
64164                   }
64165                   printf(" {%x:%d@%d%c}", ip_hdr->ih_id,
64166                           ntohs(ip_hdr->ih_length),
64167                           (ntohs(ip_hdr->ih_flags_fragoff) & IH_FRAGOFF_MASK)*8,
64168                           (ntohs(ip_hdr->ih_flags_fragoff) & IH_MORE_FRAGS) ?
64169                           '+' : '\0');
64170           }
64171   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/ip_read.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

64200    /*
64201    ip_read.c
64202    */
64203
64204    #include "inet.h"
64205    #include "buf.h"
64206    #include "clock.h"
64207    #include "type.h"
64208
64209    #include "assert.h"
64210    #include "icmp_lib.h"
64211    #include "io.h"
64212    #include "ip.h"
64213    #include "ip_int.h"
64214
64215    INIT_PANIC();
64216
64217    FORWARD ip_ass_t *find_ass_ent ARGS(( ip_port_t *port, U16_t id,
64218            int proto, ipaddr_t src, ipaddr_t dst ));
64219    FORWARD acc_t *merge_frags ARGS(( acc_t *first, acc_t *second ));
64220    FORWARD int net_broad ARGS(( ipaddr_t hoaddr, ipaddr_t netaddr,
64221            ipaddr_t netmask ));
64222    FORWARD int ip_frag_chk ARGS(( acc_t *pack ));
64223    FORWARD acc_t *reassemble ARGS(( ip_port_t *port, acc_t *pack,
64224            ip_hdr_t *ip_hdr ));
64225    FORWARD int ok_for_port ARGS(( ip_port_t *port, ipaddr_t ipaddr,
64226            int *ref_broad_all ));
64227
64228    PUBLIC int ip_read (fd, count)
64229    int fd;
64230    size_t count;
64231    {
64232            ip_fd_t *ip_fd;
64233
64234            ip_fd= &ip_fd_table[fd];
64235            if (!(ip_fd->if_flags & IFF_OPTSET))
64236                    return (*ip_fd->if_put_userdata)(ip_fd->if_srfd, EBADMODE,
64237                            (acc_t *)0, FALSE);
64238
64239            ip_fd->if_rd_count= count;
64240
64241            if (ip_fd->if_rd_buf)
64242            {
64243                    if (get_time() <= ip_fd->if_exp_tim)
64244                            return ip_packet2user (ip_fd);
64245             where();
64246                    bf_afree(ip_fd->if_rd_buf);
64247             where();
64248                    ip_fd->if_rd_buf= 0;
64249            }
64250            ip_fd->if_flags |= IFF_READ_IP;
64251    #if DEBUG & 256
64252     { where(); printf("ip_fd_table[%d].if_flags= 0x%x\n",
64253            ip_fd-ip_fd_table, ip_fd->if_flags); }
64254    #endif
```

```
64255           return NW_SUSPEND;
64256   }

64258   PRIVATE acc_t *reassemble (port, pack, pack_hdr)
64259   ip_port_t *port;
64260   acc_t *pack;
64261   ip_hdr_t *pack_hdr;
64262   {
64263           ip_ass_t *ass_ent;
64264           size_t pack_hdr_len, pack_data_len, pack_offset, tmp_offset;
64265           u16_t pack_flags_fragoff;
64266           acc_t *prev_head, *new_head, *new_tail, *tmp_acc;
64267           acc_t swap_acc;
64268           ip_hdr_t *tmp_hdr;
64269           time_t first_time;
64270
64271   #if DEBUG & 256
64272    { where(); printf("in reassemble()\n"); }
64273   #endif
64274           ass_ent= find_ass_ent (port, pack_hdr->ih_id,
64275                   pack_hdr->ih_proto, pack_hdr->ih_src, pack_hdr->ih_dst);
64276   #if DEBUG & 256
64277    { where(); ip_print_frags(ass_ent->ia_frags); printf("\n"); }
64278   #endif
64279
64280           pack_flags_fragoff= ntohs(pack_hdr->ih_flags_fragoff);
64281           pack_hdr_len= (pack_hdr->ih_vers_ihl & IH_IHL_MASK) * 4;
64282           pack_data_len= ntohs(pack_hdr->ih_length)-pack_hdr_len;
64283           pack_offset= (pack_flags_fragoff & IH_FRAGOFF_MASK)*8;
64284           pack->acc_ext_link= NULL;
64285   #if DEBUG & 256
64286    { where(); ip_print_frags(pack); printf("\n"); }
64287   #endif
64288
64289           new_head= 0;
64290
64291           for (prev_head= ass_ent->ia_frags, ass_ent->ia_frags= NULL; prev_head;
64292                   prev_head= prev_head->acc_ext_link)
64293           {
64294                   tmp_hdr= (ip_hdr_t *)ptr2acc_data(prev_head);
64295                   tmp_offset= (ntohs(tmp_hdr->ih_flags_fragoff) &
64296                           IH_FRAGOFF_MASK)*8;
64297   #if DEBUG & 256
64298    { where(); printf("tmp_offset= %d, pack_offset= %d\n", tmp_offset,
64299           pack_offset); }
64300   #endif
64301
64302
64303                   if (tmp_offset >= pack_offset)
64304                           break;
64305
64306                   if (new_head)
64307                           new_tail->acc_ext_link= prev_head;
64308                   else
64309                           new_head= prev_head;
64310                   new_tail= prev_head;
64311           }
64312           if (prev_head)
64313           {
64314   where();
```

```
64315                   pack= merge_frags(pack, prev_head);
64316           }
64317           if (new_head)
64318           {
64319                   pack= merge_frags(new_tail, pack);
64320                   if (pack != new_tail)
64321                   {
64322                           swap_acc= *pack;
64323                           *pack= *new_tail;
64324                           *new_tail= swap_acc;
64325                   }
64326           }
64327           else
64328           {
64329                   new_head= pack;
64330                   new_tail= pack;
64331           }
64332           ass_ent->ia_frags= new_head;
64333   #if DEBUG & 256
64334    { where(); ip_print_frags(ass_ent->ia_frags); printf("\n"); }
64335   #endif
64336
64337           pack= ass_ent->ia_frags;
64338           pack_hdr= (ip_hdr_t *)ptr2acc_data(pack);
64339           pack_flags_fragoff= ntohs(pack_hdr->ih_flags_fragoff);
64340   #if DEBUG & 256
64341    { where(); printf(
64342                   "merge_pack: flags_fragoff= %u, vers_ihl= 0x%x, length= %u\n",
64343           pack_flags_fragoff, pack_hdr->ih_vers_ihl,
64344           ntohs(pack_hdr->ih_length)); }
64345   #endif
64346
64347           if (!(pack_flags_fragoff & (IH_FRAGOFF_MASK|IH_MORE_FRAGS)))
64348                   /* it's now a complete packet */
64349           {
64350   #if DEBUG & 256
64351    { where(); printf("got a complete packet now\n"); }
64352   #endif
64353                   first_time= ass_ent->ia_first_time;
64354
64355                   ass_ent->ia_frags= 0;
64356                   ass_ent->ia_first_time= 0;
64357
64358                   while (pack->acc_ext_link)
64359                   {
64360    { where(); printf("strange\n"); }
64361                           tmp_acc= pack->acc_ext_link;
64362                           pack->acc_ext_link= tmp_acc->acc_ext_link;
64363                           bf_afree(tmp_acc);
64364                   }
64365                   if ((ass_ent->ia_min_ttl) * HZ + first_time <
64366                           get_time())
64367                           icmp_frag_ass_tim(pack);
64368                   else
64369                           return pack;
64370           }
64371           return NULL;
64372   }

64374   PRIVATE acc_t *merge_frags (first, second)
```

```
64375   acc_t *first, *second;
64376   {
64377           ip_hdr_t *first_hdr, *second_hdr;
64378           size_t first_hdr_size, second_hdr_size, first_datasize, second_datasize,
64379                   first_offset, second_offset;
64380           acc_t *cut_second, *tmp_acc;
64381
64382   #if DEBUG & 256
64383    { where(); ip_print_frags(first); printf(" , "); ip_print_frags(second); }
64384   #endif
64385           if (!second)
64386           {
64387                   first->acc_ext_link= NULL;
64388                   return first;
64389           }
64390
64391   assert (first->acc_length >= IP_MIN_HDR_SIZE);
64392   assert (second->acc_length >= IP_MIN_HDR_SIZE);
64393
64394           first_hdr= (ip_hdr_t *)ptr2acc_data(first);
64395           first_offset= (ntohs(first_hdr->ih_flags_fragoff) &
64396                   IH_FRAGOFF_MASK) * 8;
64397           first_hdr_size= (first_hdr->ih_vers_ihl & IH_IHL_MASK) * 4;
64398           first_datasize= ntohs(first_hdr->ih_length) - first_hdr_size;
64399
64400           for (;;)
64401           {
64402                   second_hdr= (ip_hdr_t *)ptr2acc_data(second);
64403                   second_offset= (ntohs(second_hdr->ih_flags_fragoff) &
64404                           IH_FRAGOFF_MASK) * 8;
64405                   second_hdr_size= (second_hdr->ih_vers_ihl & IH_IHL_MASK) * 4;
64406                   second_datasize= ntohs(second_hdr->ih_length) - second_hdr_size;
64407
64408   #if DEBUG
64409    if (second_offset <= first_offset)
64410    { where(); printf ("first_offset= %u, second_offset= %u\n",
64411           first_offset, second_offset);
64412    printf ("first_hdr_size= %u, second_hdr_size= %u\n",
64413           first_hdr_size, second_hdr_size);
64414    printf ("first_datasize= %u, second_datasize= %u\n",
64415           first_datasize, second_datasize); }
64416   #endif
64417   assert (first_hdr_size + first_datasize == bf_bufsize(first));
64418   assert (second_hdr_size + second_datasize == bf_bufsize(second));
64419   assert (second_offset > first_offset);
64420
64421                   if (second_offset > first_offset+first_datasize)
64422                   {
64423                           first->acc_ext_link= second;
64424                           return first;
64425                   }
64426
64427                   if (second_offset + second_datasize <= first_offset +
64428                           first_datasize)
64429                   {
64430                           first->acc_ext_link= second->acc_ext_link;
64431                           bf_afree(second);
64432                           break;
64433                   }
64434
```

```
64435                   if (!(second_hdr->ih_flags_fragoff & HTONS(IH_MORE_FRAGS)))
64436                           first_hdr->ih_flags_fragoff &= ~HTONS(IH_MORE_FRAGS);
64437
64438                   second_datasize= second_offset+second_datasize-(first_offset+
64439                           first_datasize);
64440                   cut_second= bf_cut(second, second_hdr_size + first_offset+
64441                           first_datasize-second_offset, second_datasize);
64442                   tmp_acc= second->acc_ext_link;
64443                   bf_afree(second);
64444                   second= tmp_acc;
64445
64446                   first_datasize += second_datasize;
64447                   first_hdr->ih_length= htons(first_hdr_size + first_datasize);
64448
64449                   first= bf_append (first, cut_second);
64450                   if (!second)
64451                   {
64452                           first->acc_ext_link= NULL;
64453                           break;
64454                   }
64455   assert (first->acc_length >= IP_MIN_HDR_SIZE);
64456                   first_hdr= (ip_hdr_t *)ptr2acc_data(first);
64457           }
64458   assert (first_hdr_size + first_datasize == bf_bufsize(first));
64459           return first;
64460   }

64462   PRIVATE ip_ass_t *find_ass_ent (port, id, proto, src, dst)
64463   ip_port_t *port;
64464   u16_t id;
64465   ipproto_t proto;
64466   ipaddr_t src;
64467   ipaddr_t dst;
64468   {
64469           ip_ass_t *new_ass_ent, *tmp_ass_ent;
64470           int i;
64471           acc_t *tmp_acc, *curr_acc;
64472
64473   #if DEBUG & 256
64474    { where(); printf("find_ass_ent (.., id= %u, proto= %u, src= ",
64475           id, ntohs(proto)); writeIpAddr(src); printf(" dst= ");
64476           writeIpAddr(dst); printf(")\n"); }
64477   #endif
64478           new_ass_ent= 0;
64479
64480           for (i=0, tmp_ass_ent= ip_ass_table; i<IP_ASS_NR; i++,
64481                   tmp_ass_ent++)
64482           {
64483                   if (!tmp_ass_ent->ia_frags && tmp_ass_ent->
64484                           ia_first_time)
64485                   {
64486   #if DEBUG
64487    { where(); printf("ip.c: strange ip_ass entry (can be a race condition)\n"); }
64488   #endif
64489                           continue;
64490                   }
64491
64492                   if ((tmp_ass_ent->ia_srcaddr == src) &&
64493                           (tmp_ass_ent->ia_dstaddr == dst) &&
64494                           (tmp_ass_ent->ia_proto == proto) &&
```

```
64495                           (tmp_ass_ent->ia_id == id) &&
64496                           (tmp_ass_ent->ia_port == port))
64497                   {
64498   #if DEBUG & 256
64499    { where(); printf("found an ass_ent\n"); }
64500   #endif
64501                           return tmp_ass_ent;
64502                   }
64503                   if (!new_ass_ent || tmp_ass_ent->ia_first_time <
64504                           new_ass_ent->ia_first_time)
64505                           new_ass_ent= tmp_ass_ent;
64506           }
64507   #if DEBUG & 256
64508    { where(); printf("made an ass_ent\n"); }
64509   #endif
64510           new_ass_ent->ia_min_ttl= IP_MAX_TTL;
64511           new_ass_ent->ia_port= port;
64512           new_ass_ent->ia_first_time= get_time();
64513           new_ass_ent->ia_srcaddr= src;
64514           new_ass_ent->ia_dstaddr= dst;
64515           new_ass_ent->ia_proto= proto;
64516           new_ass_ent->ia_id= id;
64517
64518           if (new_ass_ent->ia_frags)
64519           {
64520                   curr_acc= new_ass_ent->ia_frags->acc_ext_link;
64521                   while (curr_acc)
64522                   {
64523                           tmp_acc= curr_acc->acc_ext_link;
64524                           bf_afree(curr_acc);
64525                           curr_acc= tmp_acc;
64526                   }
64527                   curr_acc= new_ass_ent->ia_frags;
64528                   new_ass_ent->ia_frags= 0;
64529                   icmp_frag_ass_tim(curr_acc);
64530           }
64531           return new_ass_ent;
64532   }

64534   PUBLIC void ip_eth_arrived(ip_port, pack)
64535   ip_port_t *ip_port;
64536   acc_t *pack;
64537   {
64538           ip_hdr_t *ip_hdr;
64539           int for_this_port, broadcast_allowed, broadcast_pack;
64540           int ip_frag_len, ip_hdr_len;
64541           acc_t *ip_acc, *eth_acc;
64542           ether_addr_t eth_dst, eth_src;
64543           eth_hdr_t *eth_hdr;
64544           size_t pack_size;
64545
64546   #if DEBUG & 256
64547    { where(); printf("ip_eth_arrived(&ip_port_table[%d], packet_length= %d)\n",
64548           ip_port-ip_port_table, bf_bufsize(pack)); }
64549   #endif
64550           pack= bf_packIffLess(pack, ETH_HDR_SIZE);
64551   assert (pack->acc_length >= ETH_HDR_SIZE);
64552
64553           eth_hdr= (eth_hdr_t *)ptr2acc_data(pack);
64554           eth_dst= eth_hdr->eh_dst;
```

```
64555           eth_src= eth_hdr->eh_src;
64556           if (eth_dst.ea_addr[0] & 0x01)
64557                   broadcast_pack= TRUE;
64558           else
64559                   broadcast_pack= FALSE;
64560
64561           pack_size= bf_bufsize(pack);
64562           eth_acc= bf_cut(pack, 0, ETH_HDR_SIZE);
64563           ip_acc= bf_cut(pack, ETH_HDR_SIZE, pack_size-ETH_HDR_SIZE);
64564           pack_size -= ETH_HDR_SIZE;
64565   #if DEBUG & 256
64566    { where(); printf("packet_length= %d\n", bf_bufsize(ip_acc)); }
64567   #endif
64568           bf_afree(pack);
64569
64570           if (pack_size < IP_MIN_HDR_SIZE)
64571           {
64572   #if DEBUG
64573    { where(); printf("wrong acc_length\n"); }
64574   #endif
64575                   bf_afree(ip_acc);
64576                   bf_afree(eth_acc);
64577                   return;
64578           }
64579           ip_acc= bf_packIffLess(ip_acc, IP_MIN_HDR_SIZE);
64580   assert (ip_acc->acc_length >= IP_MIN_HDR_SIZE);
64581
64582           ip_hdr= (ip_hdr_t *)ptr2acc_data(ip_acc);
64583           ip_hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
64584           if (ip_hdr_len>IP_MIN_HDR_SIZE)
64585           {
64586                   ip_acc= bf_packIffLess(ip_acc, ip_hdr_len);
64587                   ip_hdr= (ip_hdr_t *)ptr2acc_data(ip_acc);
64588           }
64589   #if DEBUG & 256
64590    { where(); printf("ih_vers_ihl= 0x%x\n", ip_hdr->ih_vers_ihl); }
64591   #endif
64592           ip_frag_len= ntohs(ip_hdr->ih_length);
64593           if (ip_frag_len<pack_size)
64594           {
64595                   pack= ip_acc;
64596                   ip_acc= bf_cut(pack, 0, ip_frag_len);
64597                   bf_afree(pack);
64598           }
64599
64600   #if DEBUG & 256
64601    { where(); printf("ip_frag_len= %d, packet length= %d\n", ip_frag_len, bf_bufsize(ip_acc)); }
64602   #endif
64603           if (!ip_frag_chk(ip_acc))
64604           {
64605   #if DEBUG
64606    { where(); where(); printf("fragment not allright\n"); }
64607   #endif
64608                   bf_afree(ip_acc);
64609                   bf_afree(eth_acc);
64610                   return;
64611           }
64612           for_this_port= ok_for_port(ip_port, ip_hdr->ih_dst,
64613                   &broadcast_allowed);
64614
```

```
64615           if (!broadcast_allowed && broadcast_pack)
64616           {
64617                   printf("got eth-broadcast pack for ip-nonbroadcast addr, src=");
64618                   writeIpAddr(ip_hdr->ih_src);
64619                   printf(" dst=");
64620                   writeIpAddr(ip_hdr->ih_dst);
64621                   printf("\n");
64622                   bf_afree(ip_acc);
64623                   bf_afree(eth_acc);
64624                   return;
64625           }
64626   #if !IP_ROUTER
64627           if (!for_this_port)
64628           {
64629   #if DEBUG
64630    { where(); printf("ip.c: got strange packet, src=");
64631      writeIpAddr(ip_hdr->ih_src); printf(" dst="); writeIpAddr(ip_hdr->ih_dst);
64632      printf(" src_eth= "); writeEtherAddr(&eth_src); printf(" dst_eth= ");
64633      writeEtherAddr(&eth_dst); printf("\n"); }
64634   #endif
64635                   bf_afree(ip_acc);
64636                   bf_afree(eth_acc);
64637                   return;
64638           }
64639   #else
64640           if (!for_this_port)
64641           {
64642                   bf_afree(eth_acc);
64643                   ip_route(ip_port, ip_acc);
64644                   return;
64645           }
64646   #endif /* !IP_ROUTER */
64647           bf_afree(eth_acc);
64648           if (ntohs(ip_hdr->ih_flags_fragoff) & (IH_FRAGOFF_MASK|IH_MORE_FRAGS))
64649           {
64650   #if DEBUG & 256
64651    { where(); printf("reassembling\n"); }
64652   #endif
64653                   ip_acc= reassemble (ip_port, ip_acc, ip_hdr);
64654                   if (!ip_acc)
64655                           return;
64656   assert (ip_acc->acc_length >= IP_MIN_HDR_SIZE);
64657           ip_hdr= (ip_hdr_t *)ptr2acc_data(ip_acc);
64658   assert (!(ntohs(ip_hdr->ih_flags_fragoff) & (IH_FRAGOFF_MASK|IH_MORE_FRAGS)));
64659           }
64660           ip_port_arrive (ip_port, ip_acc, ip_hdr);
64661   }

64663   PRIVATE int ok_for_port (ip_port, ipaddr, ref_broad_all)
64664   ip_port_t *ip_port;
64665   ipaddr_t ipaddr;
64666   int *ref_broad_all;
64667   {
64668           ipaddr_t netmask;
64669
64670   #if DEBUG & 256
64671    { where(); printf("ok_for_port( .., "); writeIpAddr(ipaddr);
64672      printf(", ..)\nip_port->ip_ipaddr= ");
64673      writeIpAddr(ip_port->ip_ipaddr); printf("\n"); }
64674   #endif
```

```
64675           if (ipaddr == ip_port->ip_ipaddr)
64676                   *ref_broad_all= FALSE;
64677           else if (ipaddr == (ipaddr_t)-1)
64678                   *ref_broad_all= TRUE;
64679           else if (net_broad (ipaddr, ip_port->ip_ipaddr &
64680                   ip_port->ip_netmask, ip_port->ip_netmask))
64681                   *ref_broad_all= TRUE;
64682           else
64683           {
64684                   netmask= ip_get_netmask(ipaddr);
64685                   if (!net_broad (ipaddr, ip_port->ip_ipaddr & netmask,
64686                           netmask))
64687                           return FALSE;
64688                   *ref_broad_all= TRUE;
64689           }
64690           return TRUE;
64691   }

64693   PRIVATE int ip_frag_chk(pack)
64694   acc_t *pack;
64695   {
64696           ip_hdr_t *ip_hdr;
64697           int hdr_len;
64698
64699           if (pack->acc_length < sizeof(ip_hdr_t))
64700           {
64701   #if DEBUG
64702    { where(); printf("wrong length\n"); }
64703   #endif
64704                   return FALSE;
64705           }
64706
64707           ip_hdr= (ip_hdr_t *)ptr2acc_data(pack);
64708
64709           hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) * 4;
64710           if (pack->acc_length < hdr_len)
64711           {
64712   #if DEBUG
64713    { where(); printf("wrong length\n"); }
64714   #endif
64715                   return FALSE;
64716           }
64717
64718           if (((ip_hdr->ih_vers_ihl >> 4) & IH_VERSION_MASK) !=
64719                   IP_VERSION)
64720           {
64721   #if DEBUG
64722    { where(); printf("wrong version (ih_vers_ihl=0x%x)\n",ip_hdr->ih_vers_ihl); }
64723   #endif
64724                   return FALSE;
64725           }
64726           if (ntohs(ip_hdr->ih_length) != bf_bufsize(pack))
64727           {
64728   #if DEBUG
64729    { where(); printf("wrong size\n"); }
64730   #endif
64731                   return FALSE;
64732           }
64733           if ((u16_t)~oneC_sum(0, (u16_t *)ip_hdr, hdr_len))
64734           {
```

```
64735   #if DEBUG
64736    { where(); printf("packet with wrong checksum (= %x)\n",
64737      (u16_t)~oneC_sum(0, (u16_t *)ip_hdr, hdr_len)); }
64738   #endif
64739                   return FALSE;
64740           }
64741           if (hdr_len>IP_MIN_HDR_SIZE && ip_chk_hdropt((u8_t *)
64742                   (ptr2acc_data(pack) + IP_MIN_HDR_SIZE),
64743                   hdr_len-IP_MIN_HDR_SIZE))
64744           {
64745   #if DEBUG
64746    { where(); printf("packet with wrong options\n"); }
64747   #endif
64748                   return FALSE;
64749           }
64750           return TRUE;
64751   }

64753   PRIVATE int net_broad (hostaddr, netaddr, netmask)
64754   ipaddr_t hostaddr, netaddr, netmask;
64755   {
64756           if ((hostaddr & netmask) != (netaddr & netmask))
64757                   return FALSE;
64758           if ((hostaddr & ~netmask) == ~netmask)
64759                   return TRUE;
64760   #if IP_SUN_BROADCAST
64761           if ((hostaddr & ~netmask) == 0)
64762                   return TRUE;
64763   #endif
64764           return FALSE;
64765   }

64767   PUBLIC int ip_packet2user (ip_fd)
64768   ip_fd_t *ip_fd;
64769   {
64770           acc_t *pack, *tmp_pack;
64771           ip_hdr_t *hdr;
64772           int result, hdr_len;
64773           size_t size, transf_size;
64774
64775           pack= ip_fd->if_rd_buf;
64776           ip_fd->if_rd_buf= 0;
64777
64778           size= bf_bufsize (pack);
64779
64780           if (ip_fd->if_ipopt.nwio_flags & NWIO_RWDATONLY)
64781           {
64782
64783                   pack= bf_packIffLess (pack, IP_MIN_HDR_SIZE);
64784                   assert (pack->acc_length >= IP_MIN_HDR_SIZE);
64785
64786                   hdr= (ip_hdr_t *)ptr2acc_data(pack);
64787                   hdr_len= (hdr->ih_vers_ihl & IH_IHL_MASK) * 4;
64788
64789                   assert (size>= hdr_len);
64790                   size -= hdr_len;
64791                   tmp_pack= bf_cut(pack, hdr_len, size);
64792                   bf_afree(pack);
64793                   pack= tmp_pack;
64794           }
```

```
64795
64796           if (size>ip_fd->if_rd_count)
64797           {
64798                   tmp_pack= bf_cut (pack, 0, ip_fd->if_rd_count);
64799                   bf_afree(pack);
64800                   pack= tmp_pack;
64801                   transf_size= ip_fd->if_rd_count;
64802           }
64803           else
64804                   transf_size= size;
64805
64806           result= (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
64807                   (size_t)0, pack, FALSE);
64808           if (result >= 0)
64809                   if (size > transf_size)
64810                           result= EPACKSIZE;
64811                   else
64812                           result= transf_size;
64813
64814   #if DEBUG & 256
64815    { where(); printf("packet2user cleared IFF_READ_IP\n"); }
64816   #endif
64817           ip_fd->if_flags &= ~IFF_READ_IP;
64818           result= (*ip_fd->if_put_userdata)(ip_fd->if_srfd, result,
64819                           (acc_t *)0, FALSE);
64820           assert (result >= 0);
64821
64822           return result;
64823   }

64825   PUBLIC int ip_ok_for_fd (ip_fd, pack)
64826   ip_fd_t *ip_fd;
64827   acc_t *pack;
64828   {
64829           ip_port_t *ip_port;
64830           ip_hdr_t *hdr;
64831           ipaddr_t dst;
64832           unsigned long pack_kind, nwio_flags;
64833
64834
64835           assert (pack->acc_length >= IP_MIN_HDR_SIZE);
64836
64837           ip_port= ip_fd->if_port;
64838
64839           hdr= (ip_hdr_t *)ptr2acc_data(pack);
64840           dst= hdr->ih_dst;
64841           if (dst == ip_port->ip_ipaddr)
64842                   pack_kind= NWIO_EN_LOC;
64843           else
64844                   pack_kind= NWIO_DI_LOC;
64845
64846           nwio_flags= ip_fd->if_ipopt.nwio_flags;
64847           if (!(pack_kind & nwio_flags))
64848                   return FALSE;
64849
64850           if ((nwio_flags & NWIO_PROTOSPEC) &&
64851                   (hdr->ih_proto != ip_fd->if_ipopt.nwio_proto))
64852                   return FALSE;
64853
64854           if ((nwio_flags & NWIO_REMSPEC) &&
```

```
64855                   (hdr->ih_src != ip_fd->if_ipopt.nwio_rem))
64856                   return FALSE;
64857
64858           return TRUE;
64859   }

64861   PUBLIC void ip_port_arrive (ip_port, pack, ip_hdr)
64862   ip_port_t *ip_port;
64863   acc_t *pack;
64864   ip_hdr_t *ip_hdr;
64865   {
64866           ip_fd_t *ip_fd, *share_fd;
64867           ip_hdr_t *hdr;
64868           int port_nr;
64869           unsigned long ip_pack_stat;
64870           int i;
64871           ipproto_t proto;
64872           time_t exp_tim;
64873
64874   assert (pack->acc_linkC>0);
64875
64876   #if DEBUG & 256
64877    { where(); printf ("in ip_port_arrive()\n"); }
64878   #endif
64879   assert (pack->acc_length >= IP_MIN_HDR_SIZE);
64880
64881           exp_tim= get_time() + (ip_hdr->ih_ttl+1) * HZ;
64882
64883           if (ip_hdr->ih_dst == ip_port->ip_ipaddr)
64884                   ip_pack_stat= NWIO_EN_LOC;
64885           else
64886                   ip_pack_stat= NWIO_EN_BROAD;
64887
64888           proto= ip_hdr->ih_proto;
64889
64890   #if DEBUG & 256
64891    { where(); printf("proto= %d\n", proto); }
64892   #endif
64893
64894           share_fd= 0;
64895           for (i=0, ip_fd=ip_fd_table; i<IP_FD_NR; i++, ip_fd++)
64896           {
64897                   if (!(ip_fd->if_flags & IFF_INUSE))
64898                   {
64899                           continue;
64900                   }
64901   #if DEBUG & 256
64902    { where(); printf("ip_fd_table[%d].if_flags= 0x%x\n",
64903           ip_fd-ip_fd_table, ip_fd->if_flags); }
64904   #endif
64905                   if (!(ip_fd->if_flags & IFF_OPTSET))
64906                   {
64907   #if DEBUG & 256
64908    { where(); printf("%d options not set\n", i); }
64909   #endif
64910                           continue;
64911                   }
64912                   if (ip_fd->if_port != ip_port)
64913                   {
64914   #if DEBUG
```

```
64915     { where(); printf("%d wrong port\n", i); }
64916     #endif
64917                             continue;
64918                     }
64919                     if (!(ip_fd->if_ipopt.nwio_flags & ip_pack_stat))
64920                     {
64921     #if DEBUG & 256
64922     { where(); printf("%d wrong ip_pack_stat\n", i); }
64923     #endif
64924                             continue;
64925                     }
64926                     if ((ip_fd->if_ipopt.nwio_flags & NWIO_PROTOSPEC) &&
64927                             proto != ip_fd->if_ipopt.nwio_proto)
64928                     {
64929     #if DEBUG & 256
64930     { where(); printf("%d wrong proto\n", i); }
64931     #endif
64932                             continue;
64933                     }
64934                     if ((ip_fd->if_ipopt.nwio_flags & NWIO_REMSPEC) &&
64935                             ip_hdr->ih_src != ip_fd->if_ipopt.nwio_rem)
64936                     {
64937     #if DEBUG
64938     { where(); printf("%d wrong src addr (REMSPEC)\n", i); }
64939     #endif
64940                             continue;
64941                     }
64942                     if (ip_fd->if_rd_buf)
64943                     {
64944                             if ((ip_fd->if_ipopt.nwio_flags &
64945                                     NWIO_ACC_MASK) == NWIO_SHARED)
64946                             {
64947     #if DEBUG
64948     { where(); printf("%d shared packet\n", i); }
64949     #endif
64950                                     share_fd= ip_fd;
64951                                     continue;
64952                             }
64953     #if DEBUG
64954     { where(); printf("throwing away packet\n"); }
64955     #endif
64956                             bf_afree(ip_fd->if_rd_buf);
64957                     }
64958                     ip_fd->if_rd_buf= pack;
64959                     pack->acc_linkC++;
64960                     ip_fd->if_exp_tim= exp_tim;
64961
64962                     if ((ip_fd->if_ipopt.nwio_flags & NWIO_ACC_MASK) ==
64963                             NWIO_SHARED || (ip_fd->if_ipopt.nwio_flags &
64964                             NWIO_ACC_MASK) ==  NWIO_EXCL)
64965                     {
64966     #if DEBUG
64967     { where(); printf("exclusive packet\n"); }
64968     #endif
64969                             bf_afree(pack);
64970                             pack= 0;
64971                             break;
64972                     }
64973
64974                     if (ip_fd->if_flags & IFF_READ_IP)
```

```
64975                   {
64976   #if DEBUG & 256
64977    { where(); printf("%d calling packet2user\n", i); }
64978   #endif
64979                           ip_packet2user(ip_fd);
64980                   }
64981                   else
64982                   {
64983   #if DEBUG
64984    { where(); printf("%d not READ_IP\n", i); }
64985   #endif
64986                   }
64987           }
64988           if (share_fd && pack)
64989           {
64990   #if DEBUG
64991    { where(); printf("exclusive packet\n"); }
64992   #endif
64993                   bf_afree(share_fd->if_rd_buf);
64994                   share_fd->if_rd_buf= pack;
64995                   share_fd->if_exp_tim= exp_tim;
64996           }
64997           else
64998           {
64999   #if DEBUG & 256
65000    { where(); printf("throwing away packet\n"); }
65001   #endif
65002                   if (pack)
65003                           bf_afree(pack);
65004           }
65005   }
```

++
src/inet/generic/ip_write.c
++

```
65100   /*
65101   ip_write.c
65102   */
65103
65104   #include "inet.h"
65105   #include "buf.h"
65106   #include "type.h"
65107
65108   #include "arp.h"
65109   #include "assert.h"
65110   #include "clock.h"
65111   #include "eth.h"
65112   #include "icmp_lib.h"
65113   #include "io.h"
65114   #include "ip.h"
65115   #include "ip_int.h"
65116   #include "ipr.h"
65117
65118   INIT_PANIC();
65119
```

```
65120   FORWARD acc_t *get_packet ARGS(( ip_fd_t *ip_fd,
65121           U16_t id /* should be: u16_t id */ ));
65122   FORWARD int dll_ready ARGS(( ip_port_t *port, ipaddr_t dst ));
65123   FORWARD void dll_write ARGS(( ip_port_t *port, ipaddr_t dst,
65124           acc_t *pack ));
65125   FORWARD int dll_eth_ready ARGS(( ip_port_t *port, ipaddr_t dst ));
65126   FORWARD void dll_eth_write ARGS(( ip_port_t *port, ipaddr_t dst,
65127           acc_t *pack ));
65128   FORWARD void dll_eth_arp_func ARGS(( int fd,
65129           ether_addr_t *ethaddr ));
65130   FORWARD acc_t *ip_split_pack ARGS(( acc_t **ref_last,
65131           int first_size ));
65132   FORWARD void error_reply ARGS(( ip_fd_t *fd, int error ));
65133   FORWARD int chk_dstaddr ARGS(( ipaddr_t dst ));
65134   FORWARD void restart_netbroadcast ARGS(( void ));
65135   FORWARD int ip_localroute_addr ARGS(( ip_fd_t *ip_fd ));
65136   FORWARD void ip_remroute_addr ARGS(( ip_fd_t *ip_fd, U8_t ttl ));
65137   FORWARD void restart_fd_write ARGS(( ip_fd_t *ip_fd ));
65138   FORWARD void restart_netbroad_fd ARGS(( ip_fd_t *tcp_fd ));
65139   FORWARD void dll_eth_get_work ARGS(( ip_port_t *ip_port ));
65140
65141   #define NF_EMPTY                0
65142   #define NF_INUSE                1
65143   #define NF_SUSPENDED            2
65144
65145   PRIVATE unsigned int netbroad_flags= NF_EMPTY;
65146   PRIVATE acc_t *netbroad_pack;
65147   PRIVATE ipaddr_t netbroad_dst;
65148   PRIVATE ipaddr_t netbroad_netmask;
65149   PRIVATE ip_port_t *netbroad_port;
65150
65151   PUBLIC int ip_write (fd, count)
65152   int fd;
65153   size_t count;
65154   {
65155           ip_fd_t *ip_fd;
65156           acc_t *data;
65157           int result;
65158           int ttl;
65159
65160   #if DEBUG & 256
65161    { where(); printf("ip_write.c: ip_write(fd= %d, count= %d\n", fd,
65162           count); }
65163   #endif
65164           ip_fd= &ip_fd_table[fd];
65165
65166           if (!(ip_fd->if_flags & IFF_OPTSET))
65167           {
65168                   error_reply (ip_fd, EBADMODE);
65169                   return NW_OK;
65170           }
65171           if (ip_fd->if_ipopt.nwio_flags & NWIO_RWDATALL)
65172           {
65173                   if (count < IP_MIN_HDR_SIZE || count > IP_MAX_PACKSIZE)
65174                   {
65175                           error_reply (ip_fd, EPACKSIZE);
65176                           return NW_OK;
65177                   }
65178           }
65179           else
```

```
65180           {
65181   assert (ip_fd->if_ipopt.nwio_flags & NWIO_RWDATONLY);
65182                   if (count < 0 || count > IP_MAX_PACKSIZE-IP_MIN_HDR_SIZE)
65183                   {
65184                           error_reply (ip_fd, EPACKSIZE);
65185                           return NW_OK;
65186                   }
65187           }
65188           ip_fd->if_wr_count= count;
65189
65190           assert (!(ip_fd->if_flags & IFF_WRITE_IP));
65191
65192           ip_fd->if_flags &= ~IFF_WRITE_MASK;
65193           ip_fd->if_flags |= IFF_WRITE_IP;
65194
65195           if (!(ip_fd->if_port->ip_flags & IPF_IPADDRSET))
65196                   return NW_SUSPEND;
65197
65198           if (ip_fd->if_ipopt.nwio_flags & NWIO_REMSPEC)
65199                   ip_fd->if_wr_dstaddr= ip_fd->if_ipopt.nwio_rem;
65200           else
65201           {
65202                   data= (*ip_fd->if_get_userdata)(ip_fd->if_srfd,
65203                           (size_t)(offsetof (struct ip_hdr, ih_dst)),
65204                           sizeof(ipaddr_t), FALSE);
65205                   if (!data)
65206                   {
65207                           ip_fd->if_flags &= ~IFF_WRITE_IP;
65208                           return NW_OK;
65209                   }
65210                   data= bf_packIffLess (data, sizeof(ipaddr_t));
65211                   ip_fd->if_wr_dstaddr= *(ipaddr_t *)ptr2acc_data(data);
65212                   bf_afree(data);
65213                   data= 0;
65214           }
65215           if (ip_fd->if_ipopt.nwio_flags & NWIO_HDR_O_SPEC)
65216                   ttl= 255;                               /* For traceroute */
65217           else
65218           {
65219                   data= (*ip_fd->if_get_userdata)(ip_fd->if_srfd,
65220                           (size_t)(offsetof (struct ip_hdr, ih_ttl)),
65221                           sizeof(u8_t), FALSE);
65222                   if (!data)
65223                   {
65224                           ip_fd->if_flags &= ~IFF_WRITE_IP;
65225                           return NW_OK;
65226                   }
65227                   data= bf_packIffLess (data, sizeof(u8_t));
65228                   ttl= *(u8_t *)ptr2acc_data(data);
65229                   bf_afree(data);
65230                   data= 0;
65231           }
65232           result= ip_localroute_addr(ip_fd);
65233           if (!result)
65234                   ip_remroute_addr(ip_fd, ttl);
65235           if (ip_fd->if_flags & IFF_WRITE_IP)
65236                   return NW_SUSPEND;
65237           else
65238                   return NW_OK;
65239   }
```

```
65241  PRIVATE int ip_localroute_addr (ip_fd)
65242  ip_fd_t *ip_fd;
65243  {
65244          ipaddr_t dstaddr, netmask;
65245          u8_t *addrInBytes;
65246          acc_t *pack;
65247          ip_hdr_t *hdr_ptr;
65248          ip_port_t *ip_port;
65249          int result, i;
65250
65251  #if DEBUG & 256
65252   { where(); printf("ip_write.c: ip_route_addr(...) to ");
65253          writeIpAddr(ip_fd->if_wr_dstaddr); printf("\n"); }
65254  #endif
65255          dstaddr= ip_fd->if_wr_dstaddr;
65256          addrInBytes= (u8_t *)&dstaddr;
65257          ip_port= ip_fd->if_port;
65258
65259          if ((addrInBytes[0] & 0xff) == 0x7f)    /* local loopback */
65260          {
65261                  pack= get_packet(ip_fd, (u16_t)0);
65262                  if (!pack)
65263                  {
65264                          ip_fd->if_flags &= ~IFF_WRITE_IP;
65265                          return TRUE;
65266                  }
65267
65268                  assert (pack->acc_length >= IP_MIN_HDR_SIZE);
65269                  assert (pack->acc_linkC == 1);
65270                  hdr_ptr= (ip_hdr_t *)ptr2acc_data(pack);
65271                  dstaddr= hdr_ptr->ih_dst;       /* src and dst
65272                                                     addresses */
65273                  hdr_ptr->ih_dst= hdr_ptr->ih_src;
65274                  hdr_ptr->ih_src= dstaddr;
65275                  ip_port_arrive (ip_port, pack, hdr_ptr);
65276                  ip_fd->if_flags &= ~IFF_WRITE_IP;
65277                  error_reply (ip_fd, ip_fd->if_wr_count);
65278                  return TRUE;
65279          }
65280          if (dstaddr == (ipaddr_t)-1)
65281          {
65282                  ip_fd->if_flags |= IFF_DLL_WR_IP;
65283                  ip_fd->if_wr_port= ip_port;
65284
65285  #if DEBUG
65286   { where(); printf("calling restart_fd_write\n"); }
65287  #endif
65288                  restart_fd_write(ip_fd);
65289                  return TRUE;
65290          }
65291          netmask= ip_get_netmask(dstaddr);
65292
65293          for (i=0, ip_port= ip_port_table; i<IP_PORT_NR; i++, ip_port++)
65294          {
65295                  if (!(ip_port->ip_flags & IPF_IPADDRSET))
65296                  {
65297  #if DEBUG
65298   { where(); printf("!(ip_port_table[%d].ip_flags & IPF_IPADDRSET)\n",
65299          ip_port-ip_port_table); }
```

```
65300   #endif
65301                           continue;
65302                   }
65303   #if DEBUG & 256
65304    { where(); printf("ip_port_table[%d].ip_ipaddr=  ", ip_port-ip_port_table);
65305           writeIpAddr(ip_port->ip_ipaddr); printf("\n"); }
65306   #endif
65307                   if (ip_port->ip_ipaddr == dstaddr)
65308                   {
65309                           pack= get_packet(ip_fd, (u16_t)0);
65310                           if (!pack)
65311                           {
65312                                   ip_fd->if_flags &= ~IFF_WRITE_IP;
65313                                   return TRUE;
65314                           }
65315
65316                           assert (pack->acc_length >= IP_MIN_HDR_SIZE);
65317                           assert (pack->acc_linkC == 1);
65318                           ip_port_arrive (ip_port, pack,
65319                                   (ip_hdr_t *)ptr2acc_data(pack));
65320                           ip_fd->if_flags &= ~IFF_WRITE_IP;
65321                           error_reply (ip_fd, ip_fd->if_wr_count);
65322                           return TRUE;
65323                   }
65324                   if ((dstaddr & ip_port->ip_netmask) ==
65325                           (ip_port->ip_ipaddr & ip_port->ip_netmask))
65326                   {
65327                           ip_fd->if_wr_port= ip_port;
65328
65329                           if ((dstaddr & ~ip_port->ip_netmask) ==
65330                                   ~ip_port->ip_netmask)
65331                                   ip_fd->if_wr_dstaddr= (ipaddr_t)-1;
65332                           ip_fd->if_flags |= IFF_DLL_WR_IP;
65333
65334   #if DEBUG & 256
65335    { where(); printf("calling restart_fd_write\n"); }
65336   #endif
65337                           restart_fd_write(ip_fd);
65338                           return TRUE;
65339                   }
65340                   if (((dstaddr & netmask) == (ip_port->ip_ipaddr &
65341                           netmask)) && ((dstaddr & ~netmask) == ~netmask))
65342                   {
65343                           if (!(netbroad_flags & NF_INUSE))
65344                                   restart_netbroad_fd(ip_fd);
65345                           else
65346                                   ip_fd->if_flags |= IFF_NETBROAD_IP;
65347                           return TRUE;
65348                   }
65349           }
65350           return FALSE;
65351   }

65353   PRIVATE int dll_ready(port, dst)
65354   ip_port_t *port;
65355   ipaddr_t dst;
65356   {
65357           switch (port->ip_dl_type)
65358           {
65359           case IPDL_ETH:
```

```
65360                   return dll_eth_ready (port, dst);
65361           default:
65362                   ip_panic(( "strange dll_type" ));
65363           }
65364           return NW_OK;
65365   }

65367   PRIVATE int dll_eth_ready (port, dst)
65368   ip_port_t *port;
65369   ipaddr_t dst;
65370   {
65371           int result;
65372
65373           if (port->ip_dl.dl_eth.de_wr_frame || port->ip_dl.dl_eth.
65374                   de_wr_frag)
65375           {
65376   #if DEBUG & 256
65377    { where(); printf("dll_eth_ready: frame or frag present\n"); }
65378   #endif
65379                   return NW_SUSPEND;
65380           }
65381           if (dst == (ipaddr_t)-1)
65382           {
65383   #if DEBUG & 256
65384    { where(); printf("dll_eth_ready: broadcast\n"); }
65385   #endif
65386                   port->ip_dl.dl_eth.de_wr_ipaddr= dst;
65387                   port->ip_dl.dl_eth.de_wr_ethaddr.ea_addr[0]= 0xff;
65388                   port->ip_dl.dl_eth.de_wr_ethaddr.ea_addr[1]= 0xff;
65389                   port->ip_dl.dl_eth.de_wr_ethaddr.ea_addr[2]= 0xff;
65390                   port->ip_dl.dl_eth.de_wr_ethaddr.ea_addr[3]= 0xff;
65391                   port->ip_dl.dl_eth.de_wr_ethaddr.ea_addr[4]= 0xff;
65392                   port->ip_dl.dl_eth.de_wr_ethaddr.ea_addr[5]= 0xff;
65393                   return NW_OK;
65394           }
65395   #if DEBUG & 256
65396    { where(); printf("ip_write.c: calling arp_ip_eth_nonbl(...)\n"); }
65397   #endif
65398           result= arp_ip_eth_nonbl (port->ip_dl.dl_eth.de_port,
65399                   dst, &port->ip_dl.dl_eth.de_wr_ethaddr);
65400   #if DEBUG & 256
65401    { where(); printf("ip_write.c: arp_ip_eth_nonbl(...)= %d\n", result); }
65402   #endif
65403           if (result<0)
65404                   port->ip_dl.dl_eth.de_wr_ipaddr= (ipaddr_t)0;
65405
65406           if (result == EDSTNOTRCH)
65407                   return EDSTNOTRCH;
65408
65409           if (result >= 0)
65410           {
65411                   port->ip_dl.dl_eth.de_wr_ipaddr= dst;
65412                   return NW_OK;
65413           }
65414   assert (result == NW_SUSPEND);
65415
65416           if (!(port->ip_dl.dl_eth.de_flags & IEF_ARP_IP))
65417           {
65418   #if DEBUG & 256
65419    { where(); printf("dll_eth_ready: no ARP_IP\n"); }
```

```
65420   #endif
65421                   return NW_OK;
65422           }
65423   #if DEBUG
65424    { where(); printf("dll_eth_ready: ARP_IP\n"); }
65425   #endif
65426           return NW_SUSPEND;
65427   }

65429   PRIVATE void dll_write (port, dst, pack)
65430   ip_port_t *port;
65431   ipaddr_t dst;
65432   acc_t *pack;
65433   {
65434           switch (port->ip_dl_type)
65435           {
65436           case IPDL_ETH:
65437                   dll_eth_write (port, dst, pack);
65438                   break;
65439           default:
65440                   ip_panic(( "wrong dl_type" ));
65441                   break;
65442           }
65443   }

65445   PRIVATE void dll_eth_write (ip_port, dst, pack)
65446   ip_port_t *ip_port;
65447   ipaddr_t dst;
65448   acc_t *pack;
65449   {
65450           int result;
65451
65452           if (!ip_port->ip_dl.dl_eth.de_wr_frag)
65453           {
65454                   if (ip_port->ip_dl.dl_eth.de_wr_ipaddr == dst)
65455                   {
65456                           ip_port->ip_dl.dl_eth.de_wr_frag= pack;
65457                           if (!(ip_port->ip_dl.dl_eth.de_flags &
65458                                   IEF_WRITE_IP))
65459                           {
65460                                   dll_eth_write_frame(ip_port);
65461                           }
65462                           return;
65463                   }
65464                   ip_port->ip_dl.dl_eth.de_wr_ipaddr= (ipaddr_t)0;
65465   #if DEBUG & 256
65466    { where(); printf("ip_write.c: calling arp_ip_eth_nonbl(...)\n"); }
65467   #endif
65468                   result= arp_ip_eth_nonbl (ip_port->ip_dl.dl_eth.de_port,
65469                           dst, &ip_port->ip_dl.dl_eth.de_wr_ethaddr);
65470   #if DEBUG & 256
65471    { where(); printf("ip_write.c: arp_ip_eth_nonbl(...)= %d\n", result); }
65472   #endif
65473                   if (result == NW_OK)
65474                   {
65475                           ip_port->ip_dl.dl_eth.de_wr_frag= pack;
65476                           if (!(ip_port->ip_dl.dl_eth.de_flags &
65477                                   IEF_WRITE_IP))
65478                                   dll_eth_write_frame(ip_port);
65479                           return;
```

```
65480                   }
65481           }
65482           assert (!(ip_port->ip_dl.dl_eth.de_flags & IEF_ARP_MASK));
65483           ip_port->ip_dl.dl_eth.de_arp_pack= pack;
65484           ip_port->ip_dl.dl_eth.de_flags |= IEF_ARP_IP;
65485   #if DEBUG & 256
65486    { where(); printf("ip_write.c: calling arp_ip_eth(...)\n"); }
65487   #endif
65488           result= arp_ip_eth (ip_port->ip_dl.dl_eth.de_port,
65489                   ip_port-ip_port_table, dst, dll_eth_arp_func);
65490   #if DEBUG & 256
65491    { where(); printf("ip_write.c: arp_ip_eth(...)= %d\n", result); }
65492   #endif
65493           if (result == NW_SUSPEND)
65494                   ip_port->ip_dl.dl_eth.de_flags |= IEF_ARP_SP;
65495           else if (result == EDSTNOTRCH)
65496           {
65497                   if (ip_port->ip_dl.dl_eth.de_arp_pack)
65498                   {
65499                           bf_afree(ip_port->ip_dl.dl_eth.de_arp_pack);
65500                           ip_port->ip_dl.dl_eth.de_arp_pack= 0;
65501                   }
65502                   ip_port->ip_dl.dl_eth.de_flags &= ~IEF_ARP_MASK;
65503           }
65504           else
65505           {
65506                   assert (result == NW_OK);
65507                   assert (ip_port->ip_dl.dl_eth.de_flags & IEF_WRITE_IP);
65508           }
65509   }

65511   PUBLIC void dll_eth_write_frame (ip_port)
65512   ip_port_t *ip_port;
65513   {
65514           acc_t *frag, *frame, *hdr, *tail;
65515           eth_hdr_t *eth_hdr;
65516           size_t pack_size;
65517           int result;
65518
65519   #if DEBUG & 256
65520    { where(); printf("ip_write.c: dll_eth_write_frame(...)\n"); }
65521   #endif
65522
65523   assert (!(ip_port->ip_dl.dl_eth.de_flags & IEF_WRITE_IP));
65524           ip_port->ip_dl.dl_eth.de_flags |= IEF_WRITE_IP;
65525
65526           do
65527           {
65528                   if (!ip_port->ip_dl.dl_eth.de_wr_frag)
65529                   {
65530                           dll_eth_get_work (ip_port);
65531                           if (!ip_port->ip_dl.dl_eth.de_wr_frag)
65532                           {
65533                                   ip_port->ip_dl.dl_eth.de_flags &=
65534                                           ~IEF_WRITE_IP;
65535                                   return;
65536                           }
65537                   }
65538   assert (!ip_port->ip_dl.dl_eth.de_wr_frame);
65539   assert (ip_port->ip_dl.dl_eth.de_wr_frag);
```

```
65540
65541                   frag= ip_port->ip_dl.dl_eth.de_wr_frag;
65542                   ip_port->ip_dl.dl_eth.de_wr_frag= 0;
65543                   frame= ip_split_pack(&frag, ETH_MAX_PACK_SIZE-
65544                           ETH_HDR_SIZE);
65545                   if (!frame)
65546                   {
65547                           assert (!frag);
65548                           continue;
65549                   }
65550                   ip_port->ip_dl.dl_eth.de_wr_frag= frag;
65551                   hdr= bf_memreq(ETH_HDR_SIZE);
65552                   eth_hdr= (eth_hdr_t *)ptr2acc_data(hdr);
65553                   eth_hdr->eh_dst= ip_port->ip_dl.dl_eth.de_wr_ethaddr;
65554                   hdr->acc_next= frame;
65555                   frame= hdr;
65556                   hdr= 0;
65557                   pack_size= bf_bufsize(frame);
65558                   if (pack_size<ETH_MIN_PACK_SIZE)
65559                   {
65560   #if DEBUG & 256
65561    { where(); printf("pack_size= %d\n", pack_size); }
65562   #endif
65563                           tail= bf_memreq(ETH_MIN_PACK_SIZE-pack_size);
65564                           frame= bf_append(frame, tail);
65565                   }
65566   #if DEBUG & 256
65567    { where(); printf("packet size= %d\n", bf_bufsize(ip_port->ip_dl.
65568           dl_eth.de_wr_frame)); }
65569   #endif
65570                   ip_port->ip_dl.dl_eth.de_wr_frame= frame;
65571                   ip_port->ip_dl.dl_eth.de_flags &= ~IEF_WRITE_SP;
65572   #if DEBUG & 256
65573    { where(); printf("ip_write.c: calling eth_write(...)\n"); }
65574   #endif
65575                   result= eth_write (ip_port->ip_dl.dl_eth.de_fd,
65576                           bf_bufsize(ip_port->ip_dl.dl_eth.de_wr_frame));
65577   #if DEBUG & 256
65578    { where(); printf("ip_write.c: eth_write(...)= %d\n", result); }
65579   #endif
65580                   if (result == NW_SUSPEND)
65581                   {
65582                           ip_port->ip_dl.dl_eth.de_flags |= IEF_WRITE_SP;
65583                           return;
65584                   }
65585           } while (!ip_port->ip_dl.dl_eth.de_wr_frame);
65586           ip_port->ip_dl.dl_eth.de_flags &= ~IEF_WRITE_IP;
65587   }

65589   PRIVATE void dll_eth_arp_func (port, ethaddr)
65590   int port;
65591   ether_addr_t *ethaddr;
65592   {
65593           ip_port_t *ip_port;
65594
65595   #if DEBUG & 256
65596    { where(); printf("ip_write.c: dll_eth_arp_func(port= %d, ...)\n",
65597           port); }
65598   #endif
65599           ip_port= &ip_port_table[port];
```

```
65600
65601           if (ethaddr && ip_port->ip_dl.dl_eth.de_arp_pack)
65602           {
65603                   ip_port->ip_dl.dl_eth.de_arp_ethaddr= *ethaddr;
65604                   ip_port->ip_dl.dl_eth.de_flags |= IEF_ARP_COMPL;
65605           }
65606           else
65607           {
65608                   if (ip_port->ip_dl.dl_eth.de_arp_pack)
65609                   {
65610                           bf_afree(ip_port->ip_dl.dl_eth.de_arp_pack);
65611                           ip_port->ip_dl.dl_eth.de_arp_pack= 0;
65612                   }
65613                   ip_port->ip_dl.dl_eth.de_flags &= ~IEF_ARP_MASK;
65614           }
65615           if (!(ip_port->ip_dl.dl_eth.de_flags & IEF_WRITE_IP))
65616                   dll_eth_write_frame(ip_port);
65617   }

65619   PRIVATE void dll_eth_get_work(ip_port)
65620   ip_port_t *ip_port;
65621   {
65622           int i;
65623           ip_fd_t *ip_fd;
65624
65625           if (ip_port->ip_dl.dl_eth.de_wr_frag)
65626                   return;
65627
65628           if ((netbroad_flags & NF_INUSE) && netbroad_port == ip_port)
65629           {
65630                   restart_netbroadcast();
65631                   if (ip_port->ip_dl.dl_eth.de_wr_frag)
65632                           return;
65633           }
65634           if (ip_port->ip_dl.dl_eth.de_flags & IEF_ARP_COMPL)
65635           {
65636   #if DEBUG & 256
65637            { where(); printf("processing arp_pack\n"); }
65638   #endif
65639                   assert (ip_port->ip_dl.dl_eth.de_arp_pack);
65640                   ip_port->ip_dl.dl_eth.de_wr_ipaddr= (ipaddr_t)0;
65641                   ip_port->ip_dl.dl_eth.de_wr_ethaddr= ip_port->ip_dl.
65642                           dl_eth.de_arp_ethaddr;
65643                   ip_port->ip_dl.dl_eth.de_wr_frag= ip_port->ip_dl.dl_eth.
65644                           de_arp_pack;
65645                   ip_port->ip_dl.dl_eth.de_flags &= ~IEF_ARP_MASK;
65646                   return;
65647           }
65648           for (i=0, ip_fd= ip_fd_table; i<IP_FD_NR; i++, ip_fd++)
65649           {
65650                   if (!(ip_fd->if_flags & IFF_INUSE))
65651                           continue;
65652                   if (!(ip_fd->if_flags & IFF_DLL_WR_IP))
65653                           continue;
65654                   if (ip_fd->if_wr_port != ip_port)
65655                           continue;
65656   #if DEBUG & 256
65657            { where(); printf("calling restart_fd_write\n"); }
65658   #endif
65659                   restart_fd_write(ip_fd);
```

```
65660                   if (ip_port->ip_dl.dl_eth.de_wr_frag)
65661                           return;
65662           }
65663   }


65666   PRIVATE void restart_netbroad_fd(ip_fd)
65667   ip_fd_t *ip_fd;
65668   {
65669           assert (!(netbroad_flags & NF_INUSE));
65670           assert (ip_fd->if_flags & IFF_NETBROAD_IP);
65671           ip_fd->if_flags &= ~IFF_NETBROAD_IP;
65672           netbroad_flags |= NF_INUSE;
65673           netbroad_dst= ip_fd->if_wr_dstaddr;
65674           netbroad_netmask= ip_get_netmask(netbroad_dst);
65675           netbroad_pack= get_packet(ip_fd, (int)get_time());
65676           if (!netbroad_pack)
65677           {
65678                   netbroad_flags &= ~NF_INUSE;
65679                   return;
65680           }
65681           netbroad_port= ip_port_table;
65682           restart_netbroadcast();
65683
65684           error_reply(ip_fd, ip_fd->if_wr_count);
65685   }

65687   PRIVATE void restart_fd_write(ip_fd)
65688   ip_fd_t *ip_fd;
65689   {
65690           ip_port_t *ip_port;
65691           ipaddr_t dstaddr;
65692           acc_t *pack;
65693           int result;
65694
65695           assert (ip_fd->if_flags & IFF_DLL_WR_IP);
65696
65697           ip_port= ip_fd->if_wr_port;
65698           dstaddr= ip_fd->if_wr_dstaddr;
65699           result= dll_ready(ip_port, dstaddr);
65700           if (result == NW_SUSPEND)
65701           {
65702                   return;
65703           }
65704           if (result == EDSTNOTRCH)
65705           {
65706   #if DEBUG
65707    { where(); printf("dll_ready returned EDSTNOTRCH, gateway= ");
65708           writeIpAddr(ip_fd->if_wr_dstaddr); printf(", the packet was %s\n",
65709           (ip_fd->if_flags & IFF_ROUTED) ? "routed" : "not routed"); }
65710   #endif
65711                   if (!(ip_fd->if_flags & IFF_ROUTED))
65712                   {
65713                           error_reply (ip_fd, result);
65714                           return;
65715                   }
65716                   else
65717                   {
65718                           ipr_gateway_down (ip_fd->if_wr_dstaddr,
65719                                   IPR_GW_DOWN_TIMEOUT);
```

```
65720                           error_reply(ip_fd, NW_OK);
65721                           return;
65722                   }
65723           }
65724   assert (result == NW_OK);
65725
65726           ip_fd->if_flags &= ~IFF_DLL_WR_IP;
65727
65728           ip_port->ip_frame_id++;
65729           pack= get_packet(ip_fd, ip_port->ip_frame_id);
65730           if (!pack)
65731           {
65732                   return;
65733           }
65734           dll_write(ip_port, dstaddr, pack);
65735           error_reply(ip_fd, ip_fd->if_wr_count);
65736   }

65738   PRIVATE void ip_remroute_addr(ip_fd, ttl)
65739   ip_fd_t *ip_fd;
65740   u8_t ttl;
65741   {
65742           ipaddr_t dstaddr, nexthop;
65743           ip_port_t *ip_port;
65744           int result, port;
65745
65746           dstaddr= ip_fd->if_wr_dstaddr;
65747           result= iproute_frag (dstaddr, ttl, &nexthop, &port);
65748   #if DEBUG & 256
65749    { where(); printf("ip_remroute_addr("); writeIpAddr(dstaddr);
65750           printf(", %d)= %d\n", ttl, result); }
65751   #endif
65752           if (result>0)
65753           {
65754                   ip_port= &ip_port_table[port];
65755                   ip_fd->if_flags |= IFF_DLL_WR_IP|IFF_ROUTED;
65756                   ip_fd->if_wr_dstaddr= nexthop;
65757                   ip_fd->if_wr_port= ip_port;
65758   #if DEBUG & 256
65759    { where(); printf("calling restart_fd_write\n"); }
65760   #endif
65761                   restart_fd_write(ip_fd);
65762                   return;
65763           }
65764           if (result<0)
65765           {
65766                   error_reply (ip_fd, result);
65767                   return;
65768           }
65769   #if IP_ROUTER
65770           ip_panic(( "not implemented" ));
65771   #else
65772           ip_panic(( "shouldn't be here" ));
65773   #endif
65774   }

65776   PRIVATE acc_t *ip_split_pack (ref_last, first_size)
65777   acc_t **ref_last;
65778   int first_size;
65779   {
```

```
65780           int pack_siz;
65781           ip_hdr_t *first_hdr, *second_hdr;
65782           int first_hdr_len, second_hdr_len;
65783           int first_data_len, second_data_len;
65784           int new_first_data_len;
65785           int first_opt_size, second_opt_size;
65786           acc_t *first_pack, *second_pack, *tmp_pack, *tmp_pack1;
65787           u8_t *first_optptr, *second_optptr;
65788           int i, optlen;
65789
65790           first_pack= *ref_last;
65791           *ref_last= 0;
65792           second_pack= 0;
65793
65794           first_pack= bf_packIffLess(first_pack, IP_MIN_HDR_SIZE);
65795           assert (first_pack->acc_length >= IP_MIN_HDR_SIZE);
65796
65797           first_hdr= (ip_hdr_t *)ptr2acc_data(first_pack);
65798   #if DEBUG & 256
65799    { where(); writeIpAddr(first_hdr->ih_dst); printf("\n"); }
65800   #endif
65801           first_hdr_len= (first_hdr->ih_vers_ihl & IH_IHL_MASK) * 4;
65802   #if DEBUG & 256
65803    { where(); printf("fist_hdr_len= %d\n", first_hdr_len); }
65804   #endif
65805           pack_siz= bf_bufsize(first_pack);
65806           if (pack_siz > first_size)
65807           {
65808   #if DEBUG & 256
65809    { where(); printf("splitting pack\n"); }
65810   #endif
65811                   if (first_hdr->ih_flags_fragoff & HTONS(IH_DONT_FRAG))
65812                   {
65813   assert (!(first_hdr->ih_flags_fragoff) & HTONS(IH_FRAGOFF_MASK));
65814                           icmp_dont_frag(first_pack);
65815                           return 0;
65816                   }
65817                   first_data_len= ntohs(first_hdr->ih_length) - first_hdr_len;
65818                   new_first_data_len= (first_size- first_hdr_len) & ~7;
65819                           /* data goes in 8 byte chuncks */
65820                   second_data_len= first_data_len-new_first_data_len;
65821                   second_pack= bf_cut(first_pack, first_hdr_len+
65822                           new_first_data_len, second_data_len);
65823                   tmp_pack= first_pack;
65824                   first_data_len= new_first_data_len;
65825                   first_pack= bf_cut (tmp_pack, 0, first_hdr_len+first_data_len);
65826                   bf_afree(tmp_pack);
65827                   tmp_pack= bf_memreq(first_hdr_len);
65828                   tmp_pack->acc_next= second_pack;
65829                   second_pack= tmp_pack;
65830                   second_hdr= (ip_hdr_t *)ptr2acc_data(second_pack);
65831                   *second_hdr= *first_hdr;
65832                   second_hdr->ih_flags_fragoff= htons(
65833                           ntohs(first_hdr->ih_flags_fragoff)+(first_data_len/8));
65834
65835                   first_opt_size= first_hdr_len-IP_MIN_HDR_SIZE;
65836                   second_opt_size= 0;
65837                   if (first_opt_size)
65838                   {
65839                           first_pack= bf_packIffLess (first_pack,
```

```
65840                                   first_hdr_len);
65841                           first_hdr= (ip_hdr_t *)ptr2acc_data(first_pack);
65842                           assert (first_pack->acc_length>=first_hdr_len);
65843                           first_optptr= (u8_t *)ptr2acc_data(first_pack)+
65844                                   IP_MIN_HDR_SIZE;
65845                           second_optptr= (u8_t *)ptr2acc_data(
65846                                   second_pack)+IP_MIN_HDR_SIZE;
65847                           i= 0;
65848                           while (i<first_opt_size)
65849                           {
65850                                   switch (*first_optptr & IP_OPT_NUMBER)
65851                                   {
65852                                   case 0:
65853                                   case 1:
65854                                           optlen= 1;
65855                                           break;
65856                                   default:
65857                                           optlen= first_optptr[1];
65858                                           break;
65859                                   }
65860                                   assert (i + optlen <= first_opt_size);
65861                                   i += optlen;
65862                                   if (*first_optptr & IP_OPT_COPIED)
65863                                   {
65864                                           second_opt_size += optlen;
65865                                           while (optlen--)
65866                                                   *second_optptr++=
65867                                                           *first_optptr++;
65868                                   }
65869                                   else
65870                                           first_optptr += optlen;
65871                           }
65872                           while (second_opt_size & 3)
65873                           {
65874                                   *second_optptr++= 0;
65875                                   second_opt_size++;
65876                           }
65877                   }
65878                   second_hdr_len= IP_MIN_HDR_SIZE + second_opt_size;
65879   #if DEBUG & 256
65880    { where(); printf("second_opt_size= %d, second_hdr_len= %d\n",
65881           second_opt_size, second_hdr_len); }
65882   #endif
65883                   second_hdr->ih_vers_ihl= second_hdr->ih_vers_ihl & 0xf0
65884                           + (second_hdr_len/4);
65885                   second_hdr->ih_length= htons(second_data_len+
65886                           second_hdr_len);
65887                   second_pack->acc_length= second_hdr_len;
65888                   if (first_pack->acc_buffer->buf_linkC>1)
65889                   {
65890                           tmp_pack= bf_cut(first_pack, 0,
65891                                   IP_MIN_HDR_SIZE);
65892                           tmp_pack1= bf_cut(first_pack, IP_MIN_HDR_SIZE,
65893                                   bf_bufsize(first_pack)-
65894                                   IP_MIN_HDR_SIZE);
65895                           bf_afree(first_pack);
65896   #if DEBUG
65897    { where(); printf("calling bf_pack\n"); }
65898   #endif
65899                           first_pack= bf_pack(tmp_pack);
```

```
65900                           first_pack->acc_next= tmp_pack1;
65901                           first_hdr= (ip_hdr_t *)ptr2acc_data(
65902                                   first_pack);
65903                   }
65904                   assert (first_pack->acc_buffer->buf_linkC == 1);
65905                   first_hdr->ih_flags_fragoff |= HTONS(IH_MORE_FRAGS);
65906                   first_hdr->ih_length= htons(first_data_len+
65907                           first_hdr_len);
65908   assert (!(second_hdr->ih_flags_fragoff & HTONS(IH_DONT_FRAG)));
65909           }
65910           if (first_pack->acc_buffer->buf_linkC>1)
65911           {
65912                   tmp_pack= bf_cut(first_pack, 0, IP_MIN_HDR_SIZE);
65913                   tmp_pack1= bf_cut(first_pack, IP_MIN_HDR_SIZE,
65914                           bf_bufsize(first_pack)-IP_MIN_HDR_SIZE);
65915                   bf_afree(first_pack);
65916   #if DEBUG
65917    { where(); printf("calling bf_pack\n"); }
65918   #endif
65919                   first_pack= bf_pack(tmp_pack);
65920                   first_pack->acc_next= tmp_pack1;
65921                   first_hdr= (ip_hdr_t *)ptr2acc_data(first_pack);
65922           }
65923           assert (first_hdr->ih_ttl);
65924   #if DEBUG & 256
65925    { where(); printf("ip_write.c: ip_split_pack: first_hdr_len= %d\n",
65926           first_hdr_len); }
65927   #endif
65928           first_hdr->ih_hdr_chk= 0;
65929           first_hdr->ih_hdr_chk= ~oneC_sum (0, (u16_t *)first_hdr,
65930                   first_hdr_len);
65931           *ref_last= second_pack;
65932           return first_pack;
65933   }

65935   PRIVATE void restart_netbroadcast()
65936   {
65937           int was_suspended, result, i;
65938           ip_port_t *ip_port, *hi_port;
65939           ip_fd_t *ip_fd;
65940
65941           assert (netbroad_flags & NF_INUSE);
65942           was_suspended= !!(netbroad_flags & NF_SUSPENDED);
65943           hi_port= &ip_port_table[IP_PORT_NR];
65944
65945           for (; netbroad_port < hi_port; netbroad_port++)
65946           {
65947                   if (!(netbroad_port->ip_flags & IPF_IPADDRSET))
65948                           continue;
65949                   if (!((netbroad_dst ^ netbroad_port->ip_ipaddr) &
65950                           netbroad_netmask))
65951                           continue;
65952                   result= dll_ready (netbroad_port, (ipaddr_t)-1);
65953                   if (result == NW_SUSPEND)
65954                   {
65955                           netbroad_flags |= NF_SUSPENDED;
65956                           return;
65957                   }
65958   assert (result >= 0);
65959
```

```
65960                   netbroad_pack->acc_linkC++;
65961   #if DEBUG
65962    { where(); printf("calling dll_write\n"); }
65963   #endif
65964                   dll_write (netbroad_port, (ipaddr_t)(-1),
65965                           netbroad_pack);
65966           }
65967           netbroad_flags &= ~NF_INUSE;
65968           bf_afree(netbroad_pack);
65969           netbroad_pack= 0;
65970           if (!was_suspended)
65971                   return;
65972
65973           for (i=0, ip_fd= ip_fd_table; i<IP_FD_NR; i++, ip_fd++)
65974           {
65975                   if (!(ip_fd->if_flags & IFF_INUSE) ||
65976                           !(ip_fd->if_flags & IFF_NETBROAD_IP))
65977                           continue;
65978                   restart_netbroad_fd(ip_fd);
65979                   if (netbroad_flags & NF_INUSE)
65980                           return;
65981           }
65982   }

65984   PRIVATE void error_reply (ip_fd, error)
65985   ip_fd_t *ip_fd;
65986   int error;
65987   {
65988           ip_fd->if_flags &= ~IFF_WRITE_MASK;
65989           if ((*ip_fd->if_get_userdata)(ip_fd->if_srfd, (size_t)error,
65990                   (size_t)0, FALSE))
65991                   ip_panic(( "can't error_reply" ));
65992   }

65994   PRIVATE acc_t *get_packet (ip_fd, id)
65995   ip_fd_t *ip_fd;
65996   u16_t id;
65997   {
65998           acc_t *pack, *tmp_pack, *tmp_pack1;
65999           ip_hdr_t *hdr, *tmp_hdr;
66000           int pack_len, hdr_len, hdr_opt_len, error;
66001
66002           pack_len= ip_fd->if_wr_count;
66003           pack= (*ip_fd->if_get_userdata)(ip_fd->if_srfd, (size_t)0,
66004                   pack_len, FALSE);
66005           if (!pack)
66006                   return pack;
66007   assert(pack_len == bf_bufsize(pack));
66008           if (ip_fd->if_ipopt.nwio_flags & NWIO_RWDATONLY)
66009           {
66010                   tmp_pack= bf_memreq (IP_MIN_HDR_SIZE);
66011                   tmp_pack->acc_next= pack;
66012                   pack= tmp_pack;
66013                   pack_len += IP_MIN_HDR_SIZE;
66014           }
66015           if (pack_len<IP_MIN_HDR_SIZE)
66016           {
66017                   bf_afree(pack);
66018                   error_reply(ip_fd, EPACKSIZE);
66019                   return 0;
```

```
66020           }
66021           pack= bf_packIffLess(pack, IP_MIN_HDR_SIZE);
66022   assert (pack->acc_length >= IP_MIN_HDR_SIZE);
66023           hdr= (ip_hdr_t *)ptr2acc_data(pack);
66024           if (pack->acc_linkC != 1 || pack->acc_buffer->buf_linkC != 1)
66025           {
66026                   tmp_pack= bf_memreq(IP_MIN_HDR_SIZE);
66027                   tmp_hdr= (ip_hdr_t *)ptr2acc_data(tmp_pack);
66028                   *tmp_hdr= *hdr;
66029                   tmp_pack->acc_next= bf_cut(pack, IP_MIN_HDR_SIZE,
66030                           pack_len-IP_MIN_HDR_SIZE);
66031                   bf_afree(pack);
66032                   hdr= tmp_hdr;
66033   #if DEBUG & 256
66034    { where(); printf("ih_vers_ihl= 0x%x\n", hdr->ih_vers_ihl); }
66035   #endif
66036                   pack= tmp_pack;
66037   assert (pack->acc_length >= IP_MIN_HDR_SIZE);
66038           }
66039   assert (pack->acc_linkC == 1 && pack->acc_buffer->buf_linkC == 1);
66040
66041           if (ip_fd->if_ipopt.nwio_flags & NWIO_HDR_O_SPEC)
66042           {
66043   #if DEBUG & 256
66044    { where(); printf("ih_vers_ihl= 0x%x\n", hdr->ih_vers_ihl); }
66045   #endif
66046                   hdr_opt_len= ip_fd->if_ipopt.nwio_hdropt.iho_opt_siz;
66047                   if (hdr_opt_len)
66048                   {
66049                           tmp_pack= bf_cut(pack, 0, IP_MIN_HDR_SIZE);
66050                           tmp_pack1= bf_cut (pack, IP_MIN_HDR_SIZE,
66051                                   pack_len-IP_MIN_HDR_SIZE);
66052                           bf_afree(pack);
66053                           pack= bf_packIffLess(tmp_pack, IP_MIN_HDR_SIZE);
66054                           hdr= (ip_hdr_t *)ptr2acc_data(pack);
66055    { where(); printf("ih_vers_ihl= 0x%x\n", hdr->ih_vers_ihl); }
66056                           tmp_pack= bf_memreq (hdr_opt_len);
66057                           memcpy (ptr2acc_data(tmp_pack), ip_fd->if_ipopt.
66058                                   nwio_hdropt.iho_data, hdr_opt_len);
66059    { where(); printf("ih_vers_ihl= 0x%x\n", hdr->ih_vers_ihl); }
66060                           pack->acc_next= tmp_pack;
66061                           tmp_pack->acc_next= tmp_pack1;
66062                           hdr_len= IP_MIN_HDR_SIZE+hdr_opt_len;
66063                   }
66064                   else
66065                           hdr_len= IP_MIN_HDR_SIZE;
66066                   hdr->ih_vers_ihl= hdr_len/4;
66067   #if DEBUG & 256
66068    { where(); printf("ih_vers_ihl= 0x%x\n", hdr->ih_vers_ihl); }
66069   #endif
66070                   hdr->ih_tos= ip_fd->if_ipopt.nwio_tos;
66071                   hdr->ih_flags_fragoff= 0;
66072                   if (ip_fd->if_ipopt.nwio_df)
66073                           hdr->ih_flags_fragoff |= HTONS(IH_DONT_FRAG);
66074                   hdr->ih_ttl= ip_fd->if_ipopt.nwio_ttl;
66075   #if DEBUG & 256
66076    { where(); printf("ih_vers_ihl= 0x%x\n", hdr->ih_vers_ihl); }
66077   #endif
66078           }
66079           else
```

```
66080           {
66081   assert (ip_fd->if_ipopt.nwio_flags & NWIO_HDR_O_ANY);
66082                   hdr_len= (hdr->ih_vers_ihl & IH_IHL_MASK)*4;
66083   #if DEBUG & 256
66084    { where(); printf("ih_vers_ihl= 0x%x\n", hdr->ih_vers_ihl); }
66085   #endif
66086                   error= NW_OK;
66087                   if (hdr_len<IP_MIN_HDR_SIZE)
66088                           error= EINVAL;
66089                   else if (hdr_len>pack_len)
66090                           error= EPACKSIZE;
66091                   else if (!hdr->ih_ttl)
66092                           error= EINVAL;
66093                   if (error<0)
66094                   {
66095                           bf_afree(pack);
66096                           error_reply (ip_fd, error);
66097                           return 0;
66098                   }
66099                   pack= bf_packIffLess(pack, hdr_len);
66100                   hdr= (ip_hdr_t *)ptr2acc_data(pack);
66101   #if DEBUG & 256
66102    { where(); printf("ih_vers_ihl= 0x%x\n", hdr->ih_vers_ihl); }
66103   #endif
66104                   if (hdr_len != IP_MIN_HDR_SIZE)
66105                   {
66106                           error= ip_chk_hdropt((u8_t *)(ptr2acc_data(pack) +
66107                                   IP_MIN_HDR_SIZE),
66108                                   hdr_len-IP_MIN_HDR_SIZE);
66109                           if (error<0)
66110                           {
66111                                   bf_afree(pack);
66112                                   error_reply (ip_fd, error);
66113                                   return 0;
66114                           }
66115                   }
66116   #if DEBUG & 256
66117    { where(); printf("ih_vers_ihl= 0x%x\n", hdr->ih_vers_ihl); }
66118   #endif
66119           }
66120   #if DEBUG & 256
66121    if (hdr->ih_flags_fragoff & HTONS(IH_DONT_FRAG))
66122    { where(); printf("proto= %d\n", hdr->ih_proto); }
66123   #endif
66124   assert (!(hdr->ih_flags_fragoff & HTONS(IH_DONT_FRAG)));
66125   #if DEBUG & 256
66126    { where(); printf("ih_vers_ihl= 0x%x\n", hdr->ih_vers_ihl); }
66127   #endif
66128           hdr->ih_vers_ihl= (hdr->ih_vers_ihl & IH_IHL_MASK) |
66129                   (IP_VERSION << 4);
66130   #if DEBUG & 256
66131    { where(); printf("ih_vers_ihl= 0x%x\n", hdr->ih_vers_ihl); }
66132   #endif
66133           hdr->ih_length= htons(pack_len);
66134           hdr->ih_flags_fragoff &= ~HTONS(IH_FRAGOFF_MASK |
66135                   IH_FLAGS_UNUSED | IH_MORE_FRAGS);
66136           if (ip_fd->if_ipopt.nwio_flags & NWIO_PROTOSPEC)
66137                   hdr->ih_proto= ip_fd->if_ipopt.nwio_proto;
66138           hdr->ih_id= htons(id);
66139           hdr->ih_src= ip_fd->if_port->ip_ipaddr;
```

```
66140           if (ip_fd->if_ipopt.nwio_flags & NWIO_REMSPEC)
66141                   hdr->ih_dst= ip_fd->if_ipopt.nwio_rem;
66142           else
66143           {
66144   assert (ip_fd->if_ipopt.nwio_flags & NWIO_REMANY);
66145                   error= chk_dstaddr(hdr->ih_dst);
66146                   if (error<0)
66147                   {
66148                           bf_afree(pack);
66149                           error_reply(ip_fd, error);
66150                           return 0;
66151                   }
66152           }
66153           return pack;
66154   }

66156   PRIVATE chk_dstaddr (dst)
66157   ipaddr_t dst;
66158   {
66159           ipaddr_t hostrep_dst, netmask;
66160
66161           hostrep_dst= ntohl(dst);
66162           if (hostrep_dst == (ipaddr_t)-1)
66163                   return NW_OK;
66164           if ((hostrep_dst & 0xe00000001) == 0xe00000001)
66165                   return EBADDEST;
66166           netmask= ip_get_netmask(dst);
66167           if (!(dst & ~netmask))
66168                   return EBADDEST;
66169           return NW_OK;
66170   }
```

++
src/inet/generic/ipr.c
++

```
66200   /*
66201   ipr.c
66202   */
66203
66204   #include "inet.h"
66205   #include "clock.h"
66206
66207   #include "assert.h"
66208   #include "io.h"
66209   #include "ipr.h"
66210
66211   INIT_PANIC();
66212
66213   #define ROUTE_NR        32
66214   #define DIST_UNREACHABLE        512
66215
66216   PRIVATE route_t route_table[ROUTE_NR];
66217   PRIVATE int fixed_routes;
66218
66219   FORWARD route_t *get_route_ent ARGS(( ipaddr_t dest ));
```

```
66220
66221   PUBLIC void ipr_init()
66222   {
66223           int i;
66224           route_t *route_ind;
66225
66226           for (i= 0, route_ind= route_table; i<ROUTE_NR; i++, route_ind++)
66227                   route_ind->rt_flags= RTF_EMPTY;
66228           fixed_routes= 0;
66229   }

66231   PUBLIC int iproute_frag(dest, ttl, nexthop, port)
66232   ipaddr_t dest;
66233   int ttl;
66234   ipaddr_t *nexthop;
66235   int *port;
66236   {
66237           route_t *route_ent;
66238
66239           route_ent= get_route_ent(dest);
66240           if (!route_ent || route_ent->rt_dist > ttl)
66241                   return EDSTNOTRCH;
66242
66243           *nexthop= route_ent->rt_gateway;
66244           *port= route_ent->rt_port;
66245           return 1;
66246   }

66248   PRIVATE route_t *get_route_ent(dest)
66249   ipaddr_t dest;
66250   {
66251           route_t *route_ind, *route_hi;
66252           route_t *bestroute, *sec_bestroute, tmproute;
66253           time_t currtim;
66254
66255           currtim= get_time();
66256
66257           route_hi= &route_table[ROUTE_NR];
66258           bestroute= 0;
66259           for (route_ind= route_table; route_ind<route_hi; route_ind++)
66260           {
66261                   if (!(route_ind->rt_flags & RTF_INUSE))
66262                           continue;
66263                   if (route_ind->rt_exp_tim && route_ind->rt_exp_tim<currtim)
66264                   {
66265                           route_ind->rt_flags &= ~RTF_INUSE;
66266                           continue;
66267                   }
66268                   if ((dest ^ route_ind->rt_dest) & route_ind->rt_netmask)
66269                           continue;
66270
66271                   if (!bestroute)
66272                   {
66273                           bestroute= route_ind;
66274                           continue;
66275                   }
66276                   if (bestroute->rt_netmask != route_ind->rt_netmask)
66277                   {
66278                           if (route_ind->rt_dist > bestroute->rt_dist ||
66279                                   (route_ind->rt_dist == bestroute->
```

```
66280                                   rt_dist && ntohl(route_ind->rt_netmask)>
66281                                   ntohl(bestroute->rt_netmask)))
66282                                   bestroute= route_ind;
66283                           continue;
66284                   }
66285                   sec_bestroute= bestroute;
66286                   if (bestroute->rt_gateway == route_ind->rt_gateway)
66287                   {
66288                           if (route_ind->rt_timestamp >
66289                                   bestroute->rt_timestamp)
66290                                   bestroute= route_ind;
66291                   }
66292                   else
66293                   {
66294                           if (route_ind->rt_dist < bestroute->rt_dist)
66295                                   bestroute= route_ind;
66296                   }
66297                   if (bestroute->rt_dist > sec_bestroute->rt_dist)
66298                   {
66299                           tmproute= *bestroute;
66300                           *bestroute= *sec_bestroute;
66301                           *sec_bestroute= tmproute;
66302                           route_ind= bestroute= sec_bestroute;
66303                   }
66304           }
66305   #if DEBUG & 256
66306    { where(); if (!bestroute){ printf("no route to "); writeIpAddr(dest);
66307           printf("\n"); } else { printf ("route to ");
66308           writeIpAddr(bestroute->rt_dest); printf(" via ");
66309           writeIpAddr(bestroute->rt_gateway); printf(" at distance %d\n",
66310           bestroute->rt_dist); } }
66311   #endif
66312           return bestroute;
66313   }

66315   PUBLIC route_t *ipr_add_route(dest, netmask, gateway, port, timeout, dist,
66316           fixed, preference)
66317   ipaddr_t dest;
66318   ipaddr_t netmask;
66319   ipaddr_t gateway;
66320   int port;
66321   time_t timeout;
66322   int dist;
66323   int fixed;
66324   i32_t preference;
66325   {
66326           int i;
66327           route_t *route_ind;
66328           route_t *oldest_route;
66329           time_t currtim;
66330
66331   #if DEBUG & 256
66332    { where(); printf("ipr_add_route(dest= "); writeIpAddr(dest);
66333           printf(", netmask= "); writeIpAddr(netmask);
66334           printf(", gateway= "); writeIpAddr(gateway);
66335           printf(", port= %d, timeout= %ld, dist= %d, fixed= %d, pref= %d\n",
66336                   port, timeout, dist, fixed, preference); }
66337   #endif
66338           if (fixed)
66339           {
```

```
66340                   if (fixed_routes >= IPR_MAX_FIXED_ROUTES)
66341                           return 0;
66342                   fixed_routes++;
66343           }
66344           oldest_route= 0;
66345           currtim= get_time();
66346           for (i= 0, route_ind= route_table; i<ROUTE_NR; i++, route_ind++)
66347           {
66348                   if (!(route_ind->rt_flags & RTF_INUSE))
66349                   {
66350                           oldest_route= route_ind;
66351                           break;
66352                   }
66353                   if (route_ind->rt_exp_tim && route_ind->rt_exp_tim < currtim)
66354                   {
66355                           oldest_route= route_ind;
66356                           break;
66357                   }
66358                   if (route_ind->rt_dest == dest &&
66359                           route_ind->rt_netmask == netmask &&
66360                           route_ind->rt_gateway == gateway)
66361                   {
66362                           if (!fixed && (route_ind->rt_flags & RTF_FIXED))
66363                                   continue;
66364                           oldest_route= route_ind;
66365                           break;
66366                   }
66367                   if (route_ind->rt_flags & RTF_FIXED)
66368                           continue;
66369                   if (!oldest_route)
66370                   {
66371                           oldest_route= route_ind;
66372                           continue;
66373                   }
66374                   if (route_ind->rt_timestamp < oldest_route->rt_timestamp)
66375                           oldest_route= route_ind;
66376           }
66377   assert (oldest_route);
66378           oldest_route->rt_dest= dest;
66379           oldest_route->rt_gateway= gateway;
66380           oldest_route->rt_netmask= netmask;
66381           if (timeout)
66382                   oldest_route->rt_exp_tim= currtim + timeout;
66383           else
66384                   oldest_route->rt_exp_tim= 0;
66385           oldest_route->rt_timestamp= currtim;
66386           oldest_route->rt_dist= dist;
66387           oldest_route->rt_port= port;
66388           oldest_route->rt_flags= RTF_INUSE;
66389           oldest_route->rt_pref= preference;
66390           if (fixed)
66391                   oldest_route->rt_flags |= RTF_FIXED;
66392           return oldest_route;
66393   }

66395   PUBLIC void ipr_gateway_down(gateway, timeout)
66396   ipaddr_t gateway;
66397   time_t timeout;
66398   {
66399           route_t *route_ind, *route;
```

```
66400           time_t currtim;
66401           int i;
66402
66403           currtim= get_time();
66404           for (i= 0, route_ind= route_table; i<ROUTE_NR; i++, route_ind++)
66405           {
66406                   if (!(route_ind->rt_flags & RTF_INUSE))
66407                           continue;
66408                   if (route_ind->rt_gateway != gateway)
66409                           continue;
66410                   if (route_ind->rt_exp_tim && route_ind->rt_exp_tim < currtim)
66411                   {
66412                           route_ind->rt_flags &= ~RTF_INUSE;
66413                           continue;
66414                   }
66415                   if (!(route_ind->rt_flags & RTF_FIXED))
66416                   {
66417                           route_ind->rt_timestamp= currtim;
66418                           if (timeout)
66419                                   route_ind->rt_exp_tim= currtim+timeout;
66420                           else
66421                                   route_ind->rt_exp_tim= 0;
66422                           route_ind->rt_dist= DIST_UNREACHABLE;
66423                           continue;
66424                   }
66425   #if DEBUG
66426    { where(); printf("adding route\n"); }
66427   #endif
66428                   route= ipr_add_route(route_ind->rt_dest, route_ind->rt_netmask,
66429                           gateway, route_ind->rt_port, timeout, DIST_UNREACHABLE,
66430                           FALSE, 0);
66431   assert (route);
66432           }
66433   }

66435   PUBLIC void ipr_destunrch(dest, netmask, timeout)
66436   ipaddr_t dest;
66437   ipaddr_t netmask;
66438   time_t timeout;
66439   {
66440           route_t *route;
66441
66442           route= get_route_ent(dest);
66443
66444           if (!route)
66445           {
66446   #if DEBUG
66447    { where(); printf("got a dest unreachable for "); writeIpAddr(dest);
66448           printf("but no route present\n"); }
66449   #endif
66450                   return;
66451           }
66452   #if DEBUG
66453    { where(); printf("adding route\n"); }
66454   #endif
66455           route= ipr_add_route(dest, netmask, route->rt_gateway, route->rt_port,
66456                   timeout, DIST_UNREACHABLE, FALSE, 0);
66457   assert (route);
66458   }
```

```
66460   PUBLIC void ipr_redirect(dest, netmask, old_gateway, new_gateway, new_port,
66461           timeout)
66462   ipaddr_t dest;
66463   ipaddr_t netmask;
66464   ipaddr_t old_gateway;
66465   ipaddr_t new_gateway;
66466   int new_port;
66467   time_t timeout;
66468   {
66469           route_t *route;
66470
66471           route= get_route_ent(dest);
66472
66473           if (!route)
66474           {
66475   #if DEBUG
66476    { where(); printf("got a redirect for "); writeIpAddr(dest);
66477           printf("but no route present\n"); }
66478   #endif
66479                   return;
66480           }
66481           if (route->rt_gateway != old_gateway)
66482           {
66483   #if DEBUG
66484    { where(); printf("got a redirect from "); writeIpAddr(old_gateway);
66485           printf(" for "); writeIpAddr(dest); printf(" but curr gateway is ");
66486           writeIpAddr(route->rt_gateway); printf("\n"); }
66487   #endif
66488                   return;
66489           }
66490           if (route->rt_flags & RTF_FIXED)
66491           {
66492                   if ( route->rt_dest == dest)
66493                   {
66494   #if DEBUG
66495    { where(); printf("got a redirect for "); writeIpAddr(dest);
66496           printf("but route is fixed\n"); }
66497   #endif
66498                           return;
66499                   }
66500           }
66501           else
66502           {
66503   #if DEBUG
66504    { where(); printf("adding route\n"); }
66505   #endif
66506                   route= ipr_add_route(dest, netmask, route->rt_gateway,
66507                           route->rt_port, timeout, DIST_UNREACHABLE, FALSE, 0);
66508   assert(route);
66509           }
66510   #if DEBUG
66511    { where(); printf("adding route\n"); }
66512   #endif
66513           route= ipr_add_route(dest, netmask, new_gateway, new_port,
66514                   timeout, 1, FALSE, 0);
66515   assert (route);
66516   }

66518   PUBLIC void ipr_ttl_exc(dest, netmask, timeout)
66519   ipaddr_t dest;
```

```
66520   ipaddr_t netmask;
66521   time_t timeout;
66522   {
66523           route_t *route;
66524           int new_dist;
66525
66526           route= get_route_ent(dest);
66527
66528           if (!route)
66529           {
66530   #if DEBUG
66531    { where(); printf("got a ttl exceeded for "); writeIpAddr(dest);
66532           printf("but no route present\n"); }
66533   #endif
66534                   return;
66535           }
66536
66537           new_dist= route->rt_dist * 2;
66538           if (new_dist>IP_MAX_TTL)
66539           {
66540                   new_dist= route->rt_dist+1;
66541                   if (new_dist>IP_MAX_TTL)
66542                   {
66543   #if DEBUG
66544    { where(); printf("got a ttl exceeded for "); writeIpAddr(dest);
66545           printf(" but dist is %d\n", route->rt_dist); }
66546   #endif
66547                           return;
66548                   }
66549           }
66550
66551   #if DEBUG
66552    { where(); printf("adding route\n"); }
66553   #endif
66554           route= ipr_add_route(dest, netmask, route->rt_gateway, route->rt_port,
66555                   timeout, new_dist, FALSE, 0);
66556   assert (route);
66557   }

66559   int ipr_get_route(ent_no, route_ent)
66560   int ent_no;
66561   nwio_route_t *route_ent;
66562   {
66563           route_t *route;
66564
66565           if (ent_no<0 || ent_no>= ROUTE_NR)
66566                   return ENOENT;
66567
66568           route= &route_table[ent_no];
66569           if (route->rt_exp_tim && route->rt_exp_tim < get_time())
66570                   route->rt_flags &= ~RTF_INUSE;
66571
66572           route_ent->nwr_ent_count= ROUTE_NR;
66573           route_ent->nwr_dest= route->rt_dest;
66574           route_ent->nwr_netmask= route->rt_netmask;
66575           route_ent->nwr_gateway= route->rt_gateway;
66576           route_ent->nwr_dist= route->rt_dist;
66577           route_ent->nwr_flags= NWRF_EMPTY;
66578           if (route->rt_flags & RTF_INUSE)
66579           {
```

```
66580                   route_ent->nwr_flags |= NWRF_INUSE;
66581                   if (route->rt_flags & RTF_FIXED)
66582                           route_ent->nwr_flags |= NWRF_FIXED;
66583           }
66584           route_ent->nwr_pref= route->rt_pref;
66585           return 0;
66586   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/tcp.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
66600   /*
66601   tcp.c
66602   */
66603
66604   #include "inet.h"
66605   #include "buf.h"
66606   #include "clock.h"
66607   #include "type.h"
66608
66609   #include "io.h"
66610   #include "ip.h"
66611   #include "sr.h"
66612   #include "assert.h"
66613   #include "tcp.h"
66614   #include "tcp_int.h"
66615
66616   INIT_PANIC();
66617
66618   PUBLIC tcp_port_t tcp_port_table[TCP_PORT_NR];
66619   PUBLIC tcp_fd_t tcp_fd_table[TCP_FD_NR];
66620   PUBLIC tcp_conn_t tcp_conn_table[TCP_CONN_NR];
66621
66622   FORWARD void tcp_main ARGS(( tcp_port_t *port ));
66623   FORWARD acc_t *tcp_get_data ARGS(( int fd, size_t offset,
66624           size_t count, int for_ioctl ));
66625   FORWARD int tcp_put_data ARGS(( int fd, size_t offset,
66626           acc_t *data, int for_ioctl ));
66627   FORWARD void read_ip_packets ARGS(( tcp_port_t *port ));
66628   FORWARD int tcp_setconf ARGS(( tcp_fd_t *tcp_fd ));
66629   FORWARD int tcp_connect ARGS(( tcp_fd_t *tcp_fd ));
66630   FORWARD int tcp_listen ARGS(( tcp_fd_t *tcp_fd ));
66631   FORWARD int tcp_attache ARGS(( tcp_fd_t *tcp_fd ));
66632   FORWARD tcpport_t find_unused_port ARGS(( int fd ));
66633   FORWARD int is_unused_port ARGS(( Tcpport_t port ));
66634   FORWARD int reply_thr_put ARGS(( tcp_fd_t *tcp_fd, int reply,
66635           int for_ioctl ));
66636   FORWARD void reply_thr_get ARGS(( tcp_fd_t *tcp_fd, int reply,
66637           int for_ioctl ));
66638   FORWARD tcp_conn_t *find_conn_entry ARGS(( Tcpport_t locport,
66639           ipaddr_t locaddr, Tcpport_t remport, ipaddr_t readaddr ));
66640   FORWARD tcp_conn_t *find_empty_conn ARGS(( void ));
66641   FORWARD void process_inc_fragm ARGS(( tcp_port_t *tcp_port,
66642           acc_t *data ));
66643   FORWARD tcp_conn_t *find_best_conn ARGS(( ip_hdr_t *ip_hdr,
66644           tcp_hdr_t *tcp_hdr ));
```

```
66645   FORWARD void close_mainuser ARGS(( tcp_conn_t *tcp_conn,
66646           tcp_fd_t *tcp_fd ));
66647   FORWARD int conn_right4fd ARGS(( tcp_conn_t *tcp_conn, tcp_fd_t *tcp_fd ));
66648   FORWARD int tcp_su4connect ARGS(( tcp_fd_t *tcp_fd ));
66649   FORWARD void tcp_buffree ARGS(( int priority, size_t reqsize ));
66650   FORWARD void tcp_notreach ARGS(( acc_t *pack ));
66651   FORWARD void tcp_setup_conn ARGS(( tcp_conn_t *tcp_conn ));
66652
66653   PUBLIC void tcp_init()
66654   {
66655           int i, result;
66656           tcp_fd_t *tcp_fd;
66657           tcp_port_t *tcp_port;
66658           tcp_conn_t *tcp_conn;
66659
66660           assert (BUF_S >= sizeof(struct nwio_ipopt));
66661           assert (BUF_S >= sizeof(struct nwio_ipconf));
66662           assert (BUF_S >= sizeof(struct nwio_tcpconf));
66663           assert (BUF_S >= IP_MAX_HDR_SIZE);
66664           assert (BUF_S >= TCP_MAX_HDR_SIZE);
66665
66666           tcp_port_table[0].tp_minor= TCP_DEV0;
66667           tcp_port_table[0].tp_ipdev= IP0;
66668
66669           for (i=0, tcp_fd= tcp_fd_table; i<TCP_FD_NR; i++, tcp_fd++)
66670           {
66671                   tcp_fd->tf_flags= TFF_EMPTY;
66672           }
66673
66674           for (i=0, tcp_conn= tcp_conn_table; i<TCP_CONN_NR; i++,
66675                   tcp_fd++)
66676           {
66677                   tcp_conn->tc_flags= TCF_EMPTY;
66678   #if DEBUG & 256
66679    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
66680           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
66681   #endif
66682           }
66683
66684           bf_logon(tcp_buffree);
66685
66686           for (i=0, tcp_port= tcp_port_table; i<TCP_PORT_NR; i++,
66687                   tcp_port++)
66688           {
66689                   tcp_port->tp_flags= TPF_EMPTY;
66690                   tcp_port->tp_state= TPS_EMPTY;
66691
66692                   result= sr_add_minor (tcp_port->tp_minor,
66693                           tcp_port-tcp_port_table, tcp_open, tcp_close,
66694                           tcp_read, tcp_write, tcp_ioctl, tcp_cancel);
66695                   assert (result>=0);
66696
66697                   tcp_main(tcp_port);
66698           }
66699   }

66701   PRIVATE void tcp_main(tcp_port)
66702   tcp_port_t *tcp_port;
66703   {
66704           int result, i;
```

```
66705           tcp_conn_t *tcp_conn;
66706           tcp_fd_t *tcp_fd;
66707
66708           switch (tcp_port->tp_state)
66709           {
66710           case TPS_EMPTY:
66711                   tcp_port->tp_state= TPS_SETPROTO;
66712   #if DEBUG & 256
66713    { where(); printf("doing ip_open\n"); }
66714   #endif
66715                   tcp_port->tp_ipfd= ip_open(tcp_port->tp_ipdev,
66716                           tcp_port-tcp_port_table, tcp_get_data,
66717                           tcp_put_data);
66718                   if (tcp_port->tp_ipfd < 0)
66719                   {
66720                           tcp_port->tp_state= TPS_ERROR;
66721                           printf("%s, %d: unable to open ip port\n",
66722                                   __FILE__, __LINE__);
66723                           return;
66724                   }
66725
66726   #if DEBUG & 256
66727    { where(); printf("doing ip_ioctl(.., NWIOSIPOPT)\n"); }
66728   #endif
66729                   result= ip_ioctl(tcp_port->tp_ipfd, NWIOSIPOPT);
66730                   if (result == NW_SUSPEND)
66731                           tcp_port->tp_flags |= TPF_SUSPEND;
66732                   if (result < 0)
66733                   {
66734   #if DEBUG
66735    { where(); printf("ip_ioctl(..,%lx)=%d\n", NWIOSIPOPT, result); }
66736   #endif
66737                           return;
66738                   }
66739                   if (tcp_port->tp_state != TPS_GETCONF)
66740                           return;
66741                   /* drops through */
66742           case TPS_GETCONF:
66743                   tcp_port->tp_flags &= ~TPF_SUSPEND;
66744
66745   #if DEBUG & 256
66746    { where(); printf("doing ip_ioctl(.., NWIOGIPCONF)\n"); }
66747   #endif
66748                   result= ip_ioctl(tcp_port->tp_ipfd, NWIOGIPCONF);
66749                   if (result == NW_SUSPEND)
66750                           tcp_port->tp_flags |= TPF_SUSPEND;
66751                   if (result < 0)
66752                   {
66753   #if DEBUG & 256
66754    { where(); printf("ip_ioctl(..,%lx)=%d\n", NWIOGIPCONF, result); }
66755   #endif
66756                           return;
66757                   }
66758                   if (tcp_port->tp_state != TPS_MAIN)
66759                           return;
66760                   /* drops through */
66761           case TPS_MAIN:
66762                   tcp_port->tp_flags &= ~TPF_SUSPEND;
66763                   tcp_port->tp_pack= 0;
66764
```

```
66765                   tcp_conn= &tcp_conn_table[tcp_port-tcp_port_table];
66766                   tcp_conn->tc_flags= TCF_INUSE;
66767   #if DEBUG & 16
66768    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
66769           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
66770   #endif
66771                   tcp_conn->tc_locport= 0;
66772                   tcp_conn->tc_locaddr= tcp_port->tp_ipaddr;
66773                   tcp_conn->tc_remport= 0;
66774                   tcp_conn->tc_remaddr= 0;
66775                   tcp_conn->tc_state= TCS_CLOSED;
66776   #if DEBUG & 2
66777    { where(); tcp_write_state(tcp_conn); }
66778   #endif
66779                   tcp_conn->tc_mainuser= 0;
66780                   tcp_conn->tc_readuser= 0;
66781                   tcp_conn->tc_writeuser= 0;
66782                   tcp_conn->tc_connuser= 0;
66783   #if DEBUG & 256
66784    { where(); printf("tcp_conn_table[%d].tc_connuser= 0x%x\n", tcp_conn-
66785           tcp_conn_table, tcp_conn->tc_connuser); }
66786   #endif
66787                   tcp_conn->tc_orglisten= FALSE;
66788                   tcp_conn->tc_senddis= 0;
66789                   tcp_conn->tc_ISS= 0;
66790                   tcp_conn->tc_SND_UNA= tcp_conn->tc_ISS;
66791                   tcp_conn->tc_SND_TRM= tcp_conn->tc_ISS;
66792                   tcp_conn->tc_SND_NXT= tcp_conn->tc_ISS;
66793                   tcp_conn->tc_SND_UP= tcp_conn->tc_ISS;
66794                   tcp_conn->tc_SND_WL2= tcp_conn->tc_ISS;
66795                   tcp_conn->tc_IRS= 0;
66796                   tcp_conn->tc_SND_WL1= tcp_conn->tc_IRS;
66797                   tcp_conn->tc_RCV_LO= tcp_conn->tc_IRS;
66798                   tcp_conn->tc_RCV_NXT= tcp_conn->tc_IRS;
66799                   tcp_conn->tc_RCV_HI= tcp_conn->tc_IRS;
66800                   tcp_conn->tc_RCV_UP= tcp_conn->tc_IRS;
66801                   tcp_conn->tc_port= tcp_port;
66802                   tcp_conn->tc_rcvd_data= 0;
66803                   tcp_conn->tc_rcv_queue= 0;
66804                   tcp_conn->tc_send_data= 0;
66805                   tcp_conn->tc_remipopt= 0;
66806                   tcp_conn->tc_remtcpopt= 0;
66807                   tcp_conn->tc_frag2send= 0;
66808                   tcp_conn->tc_tos= TCP_DEF_TOS;
66809                   tcp_conn->tc_ttl= IP_MAX_TTL;
66810                   tcp_conn->tc_rcv_wnd= TCP_MAX_WND_SIZE;
66811                   tcp_conn->tc_urg_wnd= TCP_DEF_URG_WND;
66812                   tcp_conn->tc_max_no_retrans= TCP_DEF_MAX_NO_RETRANS;
66813                   tcp_conn->tc_0wnd_to= 0;
66814                   tcp_conn->tc_rtt= TCP_DEF_RTT;
66815                   tcp_conn->tc_ett= 0;
66816                   tcp_conn->tc_mss= TCP_DEF_MSS;
66817                   tcp_conn->tc_error= NW_OK;
66818                   tcp_conn->tc_snd_wnd= TCP_MAX_WND_SIZE;
66819
66820                   for (i=0, tcp_fd= tcp_fd_table; i<TCP_FD_NR; i++,
66821                           tcp_fd++)
66822                   {
66823                           if (!(tcp_fd->tf_flags & TFF_INUSE))
66824                                   continue;
```

```
66825                           if (tcp_fd->tf_port != tcp_port)
66826                                   continue;
66827                           if (tcp_fd->tf_flags & TFF_IOC_INIT_SP)
66828                           {
66829                                   tcp_fd->tf_flags &= ~TFF_IOC_INIT_SP;
66830   #if DEBUG & 256
66831    { where(); printf("restarting tcp_ioctl\n"); }
66832   #endif
66833                                   tcp_ioctl(i, tcp_fd->tf_ioreq);
66834                           }
66835                   }
66836                   read_ip_packets(tcp_port);
66837                   return;
66838
66839           default:
66840                   ip_panic(( "unknown state" ));
66841                   break;
66842           }
66843   }

66845   PRIVATE acc_t *tcp_get_data (port, offset, count, for_ioctl)
66846   int port;
66847   size_t offset;
66848   size_t count;
66849   int for_ioctl;
66850   {
66851           tcp_port_t *tcp_port;
66852           int result;
66853
66854           tcp_port= &tcp_port_table[port];
66855
66856           switch (tcp_port->tp_state)
66857           {
66858           case TPS_SETPROTO:
66859                   if (!count)
66860                   {
66861                           result= (int)offset;
66862                           if (result<0)
66863                           {
66864                                   tcp_port->tp_state= TPS_ERROR;
66865                                   break;
66866                           }
66867                           tcp_port->tp_state= TPS_GETCONF;
66868                           if (tcp_port->tp_flags & TPF_SUSPEND)
66869                                   tcp_main(tcp_port);
66870                           return NW_OK;
66871                   }
66872   assert (!offset);
66873   assert (count == sizeof(struct nwio_ipopt));
66874                   {
66875                           struct nwio_ipopt *ipopt;
66876                           acc_t *acc;
66877
66878                           acc= bf_memreq(sizeof(*ipopt));
66879                           ipopt= (struct nwio_ipopt *)ptr2acc_data(acc);
66880                           ipopt->nwio_flags= NWIO_COPY |
66881                                   NWIO_EN_LOC | NWIO_DI_BROAD |
66882                                   NWIO_REMANY | NWIO_PROTOSPEC |
66883                                   NWIO_HDR_O_ANY | NWIO_RWDATALL;
66884                           ipopt->nwio_proto= IPPROTO_TCP;
```

```
66885                           return acc;
66886                   }
66887           case TPS_MAIN:
66888                   assert(tcp_port->tp_flags & TPF_WRITE_IP);
66889                   if (!count)
66890                   {
66891                           result= (int)offset;
66892   #if DEBUG & 256
66893    { where(); printf("tcp_get_data: got reply: %d\n", result); }
66894   #endif
66895                           if (result<0)
66896                           {
66897                                   if (result == EDSTNOTRCH)
66898                                   {
66899                                           tcp_notreach(tcp_port->tp_pack);
66900                                   }
66901                                   else
66902                                   {
66903                                           ip_warning((
66904                                           "ip_write failed with error: %d\n",
66905                                                                   result ));
66906                                   }
66907                           }
66908                           assert (tcp_port->tp_pack);
66909                           bf_afree (tcp_port->tp_pack);
66910                           tcp_port->tp_pack= 0;
66911
66912                           if (tcp_port->tp_flags & TPF_WRITE_SP)
66913                           {
66914                                   tcp_port->tp_flags &= ~(TPF_WRITE_SP|
66915                                           TPF_WRITE_IP);
66916                                   if (tcp_port->tp_flags & TPF_MORE2WRITE)
66917                                   {
66918   #if DEBUG & 256
66919    { where(); printf("calling tcp_restart_write_port(&tcp_port_table[%d])\n",
66920           tcp_port - tcp_port_table); }
66921   #endif
66922                                           tcp_restart_write_port(
66923                                                   tcp_port);
66924                                   }
66925                           }
66926                           else
66927                                   tcp_port->tp_flags &= ~TPF_WRITE_IP;
66928                   }
66929                   else
66930                   {
66931   #if DEBUG & 256
66932    { where(); printf("suplying data, count= %d, offset= %d, bufsize= %d\n",
66933           count, offset, bf_bufsize(tcp_port->tp_pack)); }
66934   #endif
66935                           return bf_cut (tcp_port->tp_pack, offset,
66936                                   count);
66937                   }
66938                   break;
66939           default:
66940                   printf("tcp_get_data(%d, 0x%x, 0x%x) called but tp_state= 0x%x\n",
66941                           port, offset, count, tcp_port->tp_state);
66942                   break;
66943           }
66944           return NW_OK;
```

```
66945   }

66947   PRIVATE int tcp_put_data (fd, offset, data, for_ioctl)
66948   int fd;
66949   size_t offset;
66950   acc_t *data;
66951   int for_ioctl;
66952   {
66953           tcp_port_t *tcp_port;
66954           int result;
66955
66956           tcp_port= &tcp_port_table[fd];
66957
66958           switch (tcp_port->tp_state)
66959           {
66960           case TPS_GETCONF:
66961                   if (!data)
66962                   {
66963                           result= (int)offset;
66964                           if (result<0)
66965                           {
66966                                   tcp_port->tp_state= TPS_ERROR;
66967                                   return NW_OK;
66968                           }
66969                           tcp_port->tp_state= TPS_MAIN;
66970   #if DEBUG & 256
66971    { where(); printf("get GETCONF reply\n"); }
66972   #endif
66973                           if (tcp_port->tp_flags & TPF_SUSPEND)
66974                                   tcp_main(tcp_port);
66975                   }
66976                   else
66977                   {
66978                           struct nwio_ipconf *ipconf;
66979
66980                           data= bf_packIffLess(data, sizeof(*ipconf));
66981                           ipconf= (struct nwio_ipconf *)ptr2acc_data(data);
66982   assert (ipconf->nwic_flags & NWIC_IPADDR_SET);
66983                           tcp_port->tp_ipaddr= ipconf->nwic_ipaddr;
66984                           bf_afree(data);
66985                   }
66986                   break;
66987           case TPS_MAIN:
66988                   assert(tcp_port->tp_flags & TPF_READ_IP);
66989                   if (!data)
66990                   {
66991                           result= (int)offset;
66992                           if (result<0)
66993   #if DEBUG
66994    { where(); printf("tcp_put_data got error %d (ignored)\n", result); }
66995   #else
66996   ip_panic(( "ip_read() failed" ));
66997   #endif
66998                           if (tcp_port->tp_flags & TPF_READ_SP)
66999                           {
67000                                   tcp_port->tp_flags &= ~(TPF_READ_SP|
67001                                           TPF_READ_IP);
67002                                   read_ip_packets(tcp_port);
67003                           }
67004                           else
```

```
67005                                   tcp_port->tp_flags &= ~TPF_READ_IP;
67006                   }
67007                   else
67008                   {
67009   assert(!offset);          /* this isn't a valid assertion but ip sends
67010                              * only whole datagrams up */
67011   #if DEBUG & 256
67012    { where(); printf("got data from ip\n"); }
67013   #endif
67014                           process_inc_fragm(tcp_port, data);
67015                   }
67016                   break;
67017           default:
67018                   printf("tcp_put_data(%d, 0x%x, 0x%x) called but tp_state= 0x%x\n",
67019           fd, offset, data, tcp_port->tp_state);
67020                   break;
67021           }
67022           return NW_OK;
67023   }

67025   PUBLIC int tcp_open (port, srfd, get_userdata, put_userdata)
67026   int port;
67027   int srfd;
67028   get_userdata_t get_userdata;
67029   put_userdata_t put_userdata;
67030   {
67031           int i;
67032           tcp_fd_t *tcp_fd;
67033
67034           for (i=0; i<TCP_FD_NR && (tcp_fd_table[i].tf_flags & TFF_INUSE);
67035                   i++);
67036           if (i>=TCP_FD_NR)
67037           {
67038   #if DEBUG
67039    { where(); printf("out of fds\n"); }
67040   #endif
67041                   return EOUTOFBUFS;
67042           }
67043
67044           tcp_fd= &tcp_fd_table[i];
67045
67046           tcp_fd->tf_flags= TFF_INUSE;
67047           tcp_fd->tf_flags |= TFF_PUSH_DATA;      /* XXX */
67048
67049           tcp_fd->tf_port= &tcp_port_table[port];
67050           tcp_fd->tf_srfd= srfd;
67051           tcp_fd->tf_tcpconf.nwtc_flags= TCP_DEF_OPT;
67052           tcp_fd->tf_tcpconf.nwtc_remaddr= 0;
67053           tcp_fd->tf_tcpconf.nwtc_remport= 0;
67054           tcp_fd->tf_get_userdata= get_userdata;
67055           tcp_fd->tf_put_userdata= put_userdata;
67056           tcp_fd->tf_conn= 0;
67057           return i;
67058   }

67060   PUBLIC int tcp_ioctl (fd, req)
67061   int fd;
67062   int req;
67063   {
67064           tcp_fd_t *tcp_fd;
```

```
67065           tcp_port_t *tcp_port;
67066           tcp_conn_t *tcp_conn;
67067           nwio_tcpconf_t *tcp_conf;
67068           acc_t *conf_acc;
67069           int type;
67070           int result;
67071
67072   #if DEBUG & 256
67073    { where(); printf("tcp_ioctl called\n"); }
67074   #endif
67075           tcp_fd= &tcp_fd_table[fd];
67076           type= req & IOCTYPE_MASK;
67077
67078   assert (tcp_fd->tf_flags & TFF_INUSE);
67079
67080           tcp_port= tcp_fd->tf_port;
67081           tcp_fd->tf_flags |= TFF_IOCTL_IP;
67082           tcp_fd->tf_ioreq= req;
67083
67084           if (tcp_port->tp_state != TPS_MAIN)
67085           {
67086                   tcp_fd->tf_flags |= TFF_IOC_INIT_SP;
67087                   return NW_SUSPEND;
67088           }
67089
67090           switch (type)
67091           {
67092           case NWIOSTCPCONF & IOCTYPE_MASK:
67093                   if (req != NWIOSTCPCONF)
67094                   {
67095   #if DEBUG
67096    { where(); printf("0x%x: bad ioctl\n", req); }
67097   #endif
67098                           tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67099                           reply_thr_get (tcp_fd, EBADIOCTL, TRUE);
67100                           result= NW_OK;
67101                           break;
67102                   }
67103                   if (tcp_fd->tf_flags & TFF_CONNECTED)
67104                   {
67105                           tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67106                           reply_thr_get (tcp_fd, EISCONN, TRUE);
67107                           result= NW_OK;
67108                           break;
67109                   }
67110                   result= tcp_setconf(tcp_fd);
67111                   break;
67112           case NWIOGTCPCONF & IOCTYPE_MASK:
67113                   if (req != NWIOGTCPCONF)
67114                   {
67115                           tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67116                           reply_thr_put (tcp_fd, EBADIOCTL, TRUE);
67117                           result= NW_OK;
67118                           break;
67119                   }
67120                   conf_acc= bf_memreq(sizeof(*tcp_conf));
67121   assert (conf_acc->acc_length == sizeof(*tcp_conf));
67122                   tcp_conf= (nwio_tcpconf_t *)ptr2acc_data(conf_acc);
67123
67124                   *tcp_conf= tcp_fd->tf_tcpconf;
```

```
67125                   if (tcp_fd->tf_flags & TFF_CONNECTED)
67126                   {
67127                           tcp_conn= tcp_fd->tf_conn;
67128                           tcp_conf->nwtc_locport= tcp_conn->tc_locport;
67129                           tcp_conf->nwtc_remaddr= tcp_conn->tc_remaddr;
67130                           tcp_conf->nwtc_remport= tcp_conn->tc_remport;
67131                   }
67132                   tcp_conf->nwtc_locaddr= tcp_fd->tf_port->tp_ipaddr;
67133                   result= (*tcp_fd->tf_put_userdata)(tcp_fd->tf_srfd,
67134                           0, conf_acc, TRUE);
67135                   tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67136                   reply_thr_put(tcp_fd, result, TRUE);
67137                   result= NW_OK;
67138                   break;
67139           case NWIOTCPCONN & IOCTYPE_MASK:
67140                   if (req != NWIOTCPCONN)
67141                   {
67142                           tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67143                           reply_thr_get (tcp_fd, EBADIOCTL, TRUE);
67144                           result= NW_OK;
67145                           break;
67146                   }
67147                   if (tcp_fd->tf_flags & TFF_CONNECTED)
67148                   {
67149                           tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67150                           reply_thr_get (tcp_fd, EISCONN, TRUE);
67151                           result= NW_OK;
67152                           break;
67153                   }
67154                   result= tcp_connect(tcp_fd);
67155                   break;
67156           case NWIOTCPLISTEN & IOCTYPE_MASK:
67157                   if (req != NWIOTCPLISTEN)
67158                   {
67159                           tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67160                           reply_thr_get (tcp_fd, EBADIOCTL, TRUE);
67161                           result= NW_OK;
67162                           break;
67163                   }
67164                   if (tcp_fd->tf_flags & TFF_CONNECTED)
67165                   {
67166                           tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67167                           reply_thr_get (tcp_fd, EISCONN, TRUE);
67168                           result= NW_OK;
67169                           break;
67170                   }
67171                   result= tcp_listen(tcp_fd);
67172   #if DEBUG & 256
67173    { where(); printf("tcp_listen= %d\n", result); }
67174   #endif
67175                   break;
67176   #if 0
67177           case NWIOTCPATTACH & IOCTYPE_MASK:
67178                   if (req != NWIOTCPATTACH)
67179                   {
67180                           tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67181                           reply_thr_get (tcp_fd, EBADIOCTL, TRUE);
67182                           result= NW_OK;
67183                           break;
67184                   }
```

```
67185                   if (tcp_fd->tf_flags & TFF_CONNECTED)
67186                   {
67187                           tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67188                           reply_thr_get (tcp_fd, EISCONN, TRUE);
67189                           result= NW_OK;
67190                           break;
67191                   }
67192                   result= tcp_attache(tcp_fd);
67193                   break;
67194   #endif
67195           case NWIOTCPSHUTDOWN & IOCTYPE_MASK:
67196                   if (req != NWIOTCPSHUTDOWN)
67197                   {
67198                           tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67199                           reply_thr_get (tcp_fd, EBADIOCTL, TRUE);
67200                           result= NW_OK;
67201                           break;
67202                   }
67203                   if (!(tcp_fd->tf_flags & TFF_CONNECTED))
67204                   {
67205                           tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67206                           reply_thr_get (tcp_fd, ENOTCONN, TRUE);
67207                           result= NW_OK;
67208                           break;
67209                   }
67210                   tcp_fd->tf_flags |= TFF_IOCTL_IP;
67211                   tcp_fd->tf_ioreq= req;
67212                   tcp_conn= tcp_fd->tf_conn;
67213                   if (tcp_conn->tc_writeuser)
67214                           return NW_SUSPEND;
67215
67216                   tcp_conn->tc_writeuser= tcp_fd;
67217                   tcp_restart_fd_write (tcp_conn);
67218                   if (!(tcp_fd->tf_flags & TFF_IOCTL_IP))
67219                           return NW_OK;
67220                   else
67221                           return NW_SUSPEND;
67222                   break;
67223           default:
67224                   tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67225                   reply_thr_get(tcp_fd, EBADIOCTL, TRUE);
67226                   result= NW_OK;
67227                   break;
67228           }
67229           return result;
67230   }

67232   PRIVATE int tcp_setconf(tcp_fd)
67233   tcp_fd_t *tcp_fd;
67234   {
67235           nwio_tcpconf_t *tcpconf;
67236           nwio_tcpconf_t oldconf, newconf;
67237           acc_t *data;
67238           int result;
67239           tcpport_t port;
67240           tcp_fd_t *fd_ptr;
67241           unsigned int new_en_flags, new_di_flags,
67242                   old_en_flags, old_di_flags, all_flags, flags;
67243           int i;
67244
```

```
67245           data= (*tcp_fd->tf_get_userdata)
67246                   (tcp_fd->tf_srfd, 0,
67247                   sizeof(nwio_tcpconf_t), TRUE);
67248
67249           if (!data)
67250                   return EFAULT;
67251
67252           data= bf_packIffLess(data, sizeof(nwio_tcpconf_t));
67253   assert (data->acc_length == sizeof(nwio_tcpconf_t));
67254
67255           tcpconf= (nwio_tcpconf_t *)ptr2acc_data(data);
67256           oldconf= tcp_fd->tf_tcpconf;
67257           newconf= *tcpconf;
67258
67259           old_en_flags= oldconf.nwtc_flags & 0xffff;
67260           old_di_flags= (oldconf.nwtc_flags >> 16) &
67261                   0xffff;
67262           new_en_flags= newconf.nwtc_flags & 0xffff;
67263           new_di_flags= (newconf.nwtc_flags >> 16) &
67264                   0xffff;
67265           if (new_en_flags & new_di_flags)
67266           {
67267   #if DEBUG
67268    { where(); printf("bad ioctl\n"); }
67269   #endif
67270                   tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67271                   reply_thr_get(tcp_fd, EBADMODE, TRUE);
67272                   return NW_OK;
67273           }
67274
67275           /* NWTC_ACC_MASK */
67276           if (new_di_flags & NWTC_ACC_MASK)
67277           {
67278   #if DEBUG
67279    { where(); printf("bad ioctl\n"); }
67280   #endif
67281                   tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67282                   reply_thr_get(tcp_fd, EBADMODE, TRUE);
67283                   return NW_OK;
67284                   /* access modes can't be disabled */
67285           }
67286
67287           if (!(new_en_flags & NWTC_ACC_MASK))
67288                   new_en_flags |= (old_en_flags & NWTC_ACC_MASK);
67289
67290           /* NWTC_LOCPORT_MASK */
67291           if (new_di_flags & NWTC_LOCPORT_MASK)
67292           {
67293   #if DEBUG
67294    { where(); printf("bad ioctl\n"); }
67295   #endif
67296                   tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67297                   reply_thr_get(tcp_fd, EBADMODE, TRUE);
67298                   return NW_OK;
67299                   /* the loc ports can't be disabled */
67300           }
67301           if (!(new_en_flags & NWTC_LOCPORT_MASK))
67302           {
67303                   new_en_flags |= (old_en_flags &
67304                           NWTC_LOCPORT_MASK);
```

```
67305     #if DEBUG
67306      { where(); printf("locport= old locport (=%u)\n",
67307             ntohs(newconf.nwtc_locport)); }
67308     #endif
67309                     newconf.nwtc_locport= oldconf.nwtc_locport;
67310             }
67311             else if ((new_en_flags & NWTC_LOCPORT_MASK) == NWTC_LP_SEL)
67312             {
67313                     newconf.nwtc_locport= find_unused_port(tcp_fd-
67314                             tcp_fd_table);
67315     #if DEBUG & 256
67316      { where(); printf("locport selected (=%u)\n",
67317             ntohs(newconf.nwtc_locport)); }
67318     #endif
67319             }
67320             else if ((new_en_flags & NWTC_LOCPORT_MASK) == NWTC_LP_SET)
67321             {
67322     #if DEBUG & 256
67323      { where(); printf("locport set (=%u)\n",
67324        ntohs(newconf.nwtc_locport)); }
67325     #endif
67326                     if (!newconf.nwtc_locport)
67327                     {
67328     #if DEBUG
67329      { where(); printf("bad ioctl\n"); }
67330     #endif
67331                             tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67332                             reply_thr_get(tcp_fd, EBADMODE, TRUE);
67333                             return NW_OK;
67334                     }
67335             }
67336
67337             /* NWTC_REMADDR_MASK */
67338             if (!((new_en_flags | new_di_flags) &
67339                     NWTC_REMADDR_MASK))
67340             {
67341                     new_en_flags |= (old_en_flags &
67342                             NWTC_REMADDR_MASK);
67343                     new_di_flags |= (old_di_flags &
67344                             NWTC_REMADDR_MASK);
67345                     newconf.nwtc_remaddr= oldconf.nwtc_remaddr;
67346             }
67347             else if (new_en_flags & NWTC_SET_RA)
67348             {
67349                     if (!newconf.nwtc_remaddr)
67350                     {
67351     #if DEBUG
67352      { where(); printf("bad ioctl\n"); }
67353     #endif
67354                             tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67355                             reply_thr_get(tcp_fd, EBADMODE, TRUE);
67356                             return NW_OK;
67357                     }
67358             }
67359             else
67360             {
67361     assert (new_di_flags & NWTC_REMADDR_MASK);
67362                     newconf.nwtc_remaddr= 0;
67363             }
67364
```

```
67365           /* NWTC_REMPORT_MASK */
67366           if (!((new_en_flags | new_di_flags) & NWTC_REMPORT_MASK))
67367           {
67368                   new_en_flags |= (old_en_flags &
67369                           NWTC_REMPORT_MASK);
67370                   new_di_flags |= (old_di_flags &
67371                           NWTC_REMPORT_MASK);
67372                   newconf.nwtc_remport=
67373                           oldconf.nwtc_remport;
67374           }
67375           else if (new_en_flags & NWTC_SET_RP)
67376           {
67377                   if (!newconf.nwtc_remport)
67378                   {
67379   #if DEBUG
67380    { where(); printf("bad ioctl\n"); }
67381   #endif
67382                           tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67383                           reply_thr_get(tcp_fd, EBADMODE, TRUE);
67384                           return NW_OK;
67385                   }
67386           }
67387           else
67388           {
67389   assert (new_di_flags & NWTC_REMPORT_MASK);
67390                   newconf.nwtc_remport= 0;
67391           }
67392
67393           newconf.nwtc_flags= ((unsigned long)new_di_flags
67394                   << 16) | new_en_flags;
67395           all_flags= new_en_flags | new_di_flags;
67396
67397           /* Let's check the access modes */
67398           if ((all_flags & NWTC_LOCPORT_MASK) == NWTC_LP_SEL ||
67399                   (all_flags & NWTC_LOCPORT_MASK) == NWTC_LP_SET)
67400           {
67401                   for (i=0, fd_ptr= tcp_fd_table; i<TCP_FD_NR; i++, fd_ptr++)
67402                   {
67403                           if (fd_ptr == tcp_fd)
67404                                   continue;
67405                           if (!(fd_ptr->tf_flags & TFF_INUSE))
67406                                   continue;
67407                           flags= fd_ptr->tf_tcpconf.nwtc_flags;
67408                           if ((flags & NWTC_LOCPORT_MASK) != NWTC_LP_SEL &&
67409                                   (flags &  NWTC_LOCPORT_MASK) != NWTC_LP_SET)
67410                                   continue;
67411                           if (fd_ptr->tf_tcpconf.nwtc_locport !=
67412                                   newconf.nwtc_locport)
67413                                   continue;
67414                           if ((flags & NWTC_ACC_MASK) != (all_flags  &
67415                                   NWTC_ACC_MASK))
67416                           {
67417                                   tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67418                                   reply_thr_get(tcp_fd, EADDRINUSE, TRUE);
67419                                   return NW_OK;
67420                           }
67421                   }
67422           }
67423
67424           tcp_fd->tf_tcpconf= newconf;
```

```
67425
67426           if ((all_flags & NWTC_ACC_MASK) &&
67427                   ((all_flags & NWTC_LOCPORT_MASK) == NWTC_LP_SET ||
67428                   (all_flags & NWTC_LOCPORT_MASK) == NWTC_LP_SEL) &&
67429                   (all_flags & NWTC_REMADDR_MASK) &&
67430                   (all_flags & NWTC_REMPORT_MASK))
67431                   tcp_fd->tf_flags |= TFF_OPTSET;
67432           else
67433           {
67434   #if DEBUG
67435    { where();
67436      if (!(all_flags & NWTC_ACC_MASK)) printf("NWTC_ACC_MASK not set ");
67437      if (!(all_flags & (NWTC_LP_SET|NWTC_LP_SEL)))
67438       printf("local port not set ");
67439      if (!(all_flags & NWTC_REMADDR_MASK))
67440       printf("NWTC_REMADDR_MASK not set ");
67441      if (!(all_flags & NWTC_REMPORT_MASK))
67442       printf("NWTC_REMPORT_MASK not set "); }
67443   #endif
67444                   tcp_fd->tf_flags &= ~TFF_OPTSET;
67445           }
67446           bf_afree(data);
67447           tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67448           reply_thr_get(tcp_fd, NW_OK, TRUE);
67449           return NW_OK;
67450   }


67453   PRIVATE tcpport_t find_unused_port(fd)
67454   int fd;
67455   {
67456           tcpport_t port, nw_port;
67457
67458           for (port= 0x8000; port < 0xffff-TCP_FD_NR; port+= TCP_FD_NR)
67459           {
67460                   nw_port= htons(port);
67461                   if (is_unused_port(nw_port))
67462                           return nw_port;
67463           }
67464           for (port= 0x8000; port < 0xffff; port++)
67465           {
67466                   nw_port= htons(port);
67467                   if (is_unused_port(nw_port))
67468                           return nw_port;
67469           }
67470           ip_panic(( "unable to find unused port (shouldn't occur)" ));
67471           return 0;
67472   }

67474   PRIVATE int is_unused_port(port)
67475   tcpport_t port;
67476   {
67477           int i;
67478           tcp_fd_t *tcp_fd;
67479           tcp_conn_t *tcp_conn;
67480
67481           for (i= 0, tcp_fd= tcp_fd_table; i<TCP_FD_NR; i++,
67482                   tcp_fd++)
67483           {
67484                   if (!(tcp_fd->tf_flags & TFF_OPTSET))
```

```
67485                           continue;
67486                   if (tcp_fd->tf_tcpconf.nwtc_locport == port)
67487                           return FALSE;
67488           }
67489           for (i= TCP_PORT_NR, tcp_conn= tcp_conn_table+i;
67490                   i<TCP_CONN_NR; i++, tcp_conn++)
67491                   /* the first TCP_PORT_NR ports are special */
67492           {
67493                   if (!(tcp_conn->tc_flags & TCF_INUSE))
67494                           continue;
67495                   if (tcp_conn->tc_locport == port)
67496                           return FALSE;
67497           }
67498           return TRUE;
67499   }

67501   PRIVATE int reply_thr_put(tcp_fd, reply, for_ioctl)
67502   tcp_fd_t *tcp_fd;
67503   int reply;
67504   int for_ioctl;
67505   {
67506   #if DEBUG & 256
67507    { where(); printf("reply_thr_put(..) called\n"); }
67508   #endif
67509   #if DEBUG & 256
67510    { where(); printf("calling 0x%x\n", tcp_fd->tf_put_userdata); }
67511   #endif
67512   assert (tcp_fd);
67513           return (*tcp_fd->tf_put_userdata)(tcp_fd->tf_srfd, reply,
67514                   (acc_t *)0, for_ioctl);
67515   }

67517   PRIVATE void reply_thr_get(tcp_fd, reply, for_ioctl)
67518   tcp_fd_t *tcp_fd;
67519   int reply;
67520   int for_ioctl;
67521   {
67522           acc_t *result;
67523   #if DEBUG & 256
67524    { where(); printf("reply_thr_get(..) called\n"); }
67525   #endif
67526           result= (*tcp_fd->tf_get_userdata)(tcp_fd->tf_srfd, reply,
67527                   (size_t)0, for_ioctl);
67528           assert (!result);
67529   }

67531   PUBLIC int tcp_su4listen(tcp_fd)
67532   tcp_fd_t *tcp_fd;
67533   {
67534           tcp_conn_t *tcp_conn;
67535           acc_t *tmp_acc;
67536
67537           tcp_conn= tcp_fd->tf_conn;
67538
67539           tcp_conn->tc_locport= tcp_fd->tf_tcpconf.nwtc_locport;
67540           tcp_conn->tc_locaddr= tcp_fd->tf_port->tp_ipaddr;
67541           if (tcp_fd->tf_tcpconf.nwtc_flags & NWTC_SET_RP)
67542                   tcp_conn->tc_remport= tcp_fd->tf_tcpconf.nwtc_remport;
67543           else
67544                   tcp_conn->tc_remport= 0;
```

```
67545           if (tcp_fd->tf_tcpconf.nwtc_flags & NWTC_SET_RA)
67546                   tcp_conn->tc_remaddr= tcp_fd->tf_tcpconf.nwtc_remaddr;
67547           else
67548                   tcp_conn->tc_remaddr= 0;
67549
67550           tcp_setup_conn(tcp_conn);
67551           tcp_conn->tc_port= tcp_fd->tf_port;
67552           tcp_conn->tc_mainuser= tcp_fd;
67553           tcp_conn->tc_connuser= tcp_fd;
67554           tcp_conn->tc_orglisten= TRUE;
67555           tcp_conn->tc_state= TCS_LISTEN;
67556   #if DEBUG & 2
67557    { where(); tcp_write_state(tcp_conn); }
67558   #endif
67559           return NW_SUSPEND;
67560   }

67562   /*
67563   find_empty_conn
67564
67565   This function returns a connection that is not inuse.
67566   This includes connections that are never used, and connection without a
67567   user that are not used for a while.
67568   */
67569
67570   PRIVATE tcp_conn_t *find_empty_conn()
67571   {
67572           int i;
67573           tcp_conn_t *tcp_conn;
67574           int state;
67575
67576           for (i=TCP_PORT_NR, tcp_conn= tcp_conn_table+i;
67577                   i<TCP_CONN_NR; i++, tcp_conn++)
67578                   /* the first TCP_PORT_NR connection are reserved for
67579                           RSTs */
67580           {
67581                   if (tcp_conn->tc_flags == TCF_EMPTY)
67582                   {
67583                           tcp_conn->tc_connuser= NULL;
67584                           tcp_conn->tc_mainuser= NULL;
67585                           return tcp_conn;
67586                   }
67587                   if (tcp_conn->tc_mainuser)
67588                           continue;
67589                   if (tcp_conn->tc_senddis > get_time())
67590                           continue;
67591                   if (tcp_conn->tc_state != TCS_CLOSED)
67592                   {
67593   #if DEBUG
67594    { where(); printf("calling tcp_close_connection\n"); }
67595   #endif
67596                            tcp_close_connection (tcp_conn, ENOCONN);
67597                   }
67598                   return tcp_conn;
67599           }
67600           return NULL;
67601   }


67604   /*
```

```
67605   find_conn_entry
67606
67607   This function return a connection matching locport, locaddr, remport, remaddr.
67608   If no such connection exists NULL is returned.
67609   If a connection exists without mainuser it is closed.
67610   */
67611
67612   PRIVATE tcp_conn_t *find_conn_entry(locport, locaddr, remport, remaddr)
67613   tcpport_t locport;
67614   ipaddr_t locaddr;
67615   tcpport_t remport;
67616   ipaddr_t remaddr;
67617   {
67618           tcp_conn_t *tcp_conn;
67619           int i, state;
67620
67621   #if DEBUG & 256
67622    { where(); printf("find_conn_entry(locport= %u, locaddr= ", ntohs(locport));
67623           writeIpAddr(locaddr);
67624           printf("\nremport= %u, remaddr= ", ntohs(remport));
67625           writeIpAddr(remaddr); printf(")\n"); }
67626   #endif
67627   assert(remport);
67628   assert(remaddr);
67629           for (i=TCP_PORT_NR, tcp_conn= tcp_conn_table+i; i<TCP_CONN_NR;
67630                   i++, tcp_conn++)
67631                   /* the first TCP_PORT_NR connection are reserved for
67632                           RSTs */
67633           {
67634                   if (tcp_conn->tc_flags == TCF_EMPTY)
67635                           continue;
67636                   if (tcp_conn->tc_locport != locport ||
67637                           tcp_conn->tc_locaddr != locaddr ||
67638                           tcp_conn->tc_remport != remport ||
67639                           tcp_conn->tc_remaddr != remaddr)
67640                           continue;
67641                   if (tcp_conn->tc_mainuser)
67642                           return tcp_conn;
67643                   state= tcp_conn->tc_state;
67644                   if (state != TCS_CLOSED)
67645                   {
67646                           tcp_close_connection(tcp_conn, ENOCONN);
67647                   }
67648                   return tcp_conn;
67649           }
67650           return NULL;
67651   }

67653   PRIVATE void read_ip_packets(tcp_port)
67654   tcp_port_t *tcp_port;
67655   {
67656           int result;
67657
67658           do
67659           {
67660                   tcp_port->tp_flags |= TPF_READ_IP;
67661   #if DEBUG & 256
67662    { where(); printf("doing ip_read\n"); }
67663   #endif
67664                   result= ip_read(tcp_port->tp_ipfd, TCP_MAX_DATAGRAM);
```

```
67665                   if (result == NW_SUSPEND)
67666                   {
67667                           tcp_port->tp_flags |= TPF_READ_SP;
67668                           return;
67669                   }
67670   assert(result == NW_OK);
67671                   tcp_port->tp_flags &= ~TPF_READ_IP;
67672           } while(!(tcp_port->tp_flags & TPF_READ_IP));
67673   }

67675   /*
67676   process_inc_fragm
67677   */
67678
67679   PRIVATE void process_inc_fragm(tcp_port, data)
67680   tcp_port_t *tcp_port;
67681   acc_t *data;
67682   {
67683           acc_t *ip_pack, *tcp_pack;
67684           ip_hdr_t *ip_hdr;
67685           tcp_hdr_t *tcp_hdr;
67686           tcp_conn_t *tcp_conn;
67687           int pack_len, ip_hdr_len;
67688
67689           pack_len= bf_bufsize(data);
67690           if ((u16_t)~tcp_pack_oneCsum(data, pack_len))
67691           {
67692                   data= bf_packIffLess(data, IP_MIN_HDR_SIZE);
67693                   ip_hdr= (ip_hdr_t *)ptr2acc_data(data);
67694   #if DEBUG
67695    { where(); printf("checksum error in tcp_pack\n");
67696      printf("tcp_pack_oneCsum(...)= 0x%x (%d) length= %d\n",
67697       (u16_t)~tcp_pack_oneCsum(data, pack_len),
67698       (u16_t)~tcp_pack_oneCsum(data, pack_len), pack_len);
67699       printf("src ip_addr= "); writeIpAddr(ip_hdr->ih_src); printf("\n"); }
67700   #endif
67701                   bf_afree(data);
67702                   return;
67703           }
67704
67705           data= bf_packIffLess(data, IP_MIN_HDR_SIZE);
67706           ip_hdr= (ip_hdr_t *)ptr2acc_data(data);
67707           ip_hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
67708           ip_pack= bf_cut(data, 0, ip_hdr_len);
67709
67710           pack_len -= ip_hdr_len;
67711
67712           tcp_pack= bf_cut(data, ip_hdr_len, pack_len);
67713           bf_afree(data);
67714
67715           tcp_pack= bf_packIffLess(tcp_pack, TCP_MIN_HDR_SIZE);
67716           tcp_hdr= (tcp_hdr_t *)ptr2acc_data(tcp_pack);
67717
67718           tcp_conn= find_best_conn(ip_hdr, tcp_hdr);
67719   assert(tcp_conn);
67720   #if DEBUG & 256
67721    { where(); tcp_print_pack(ip_hdr, tcp_hdr); printf("\n");
67722      tcp_print_conn(tcp_conn); printf("\n"); }
67723   #endif
67724
```

```
67725           tcp_hdr->th_chksum= pack_len;   /* tcp_pack size in chksum field */
67726   #if DEBUG & 256
67727    { where(); printf("calling tcp_frag2conn(...)\n"); }
67728   #endif
67729           tcp_frag2conn(tcp_conn, ip_pack, tcp_pack);
67730   }

67732   /*
67733   find_best_conn
67734   */
67735
67736   PRIVATE tcp_conn_t *find_best_conn(ip_hdr, tcp_hdr)
67737   ip_hdr_t *ip_hdr;
67738   tcp_hdr_t *tcp_hdr;
67739   {
67740
67741           int best_level, new_level;
67742           tcp_conn_t *best_conn, *listen_conn, *tcp_conn;
67743           tcp_fd_t *tcp_fd;
67744           int i;
67745           ipaddr_t locaddr;
67746           ipaddr_t remaddr;
67747           tcpport_t locport;
67748           tcpport_t remport;
67749
67750           locaddr= ip_hdr->ih_dst;
67751           remaddr= ip_hdr->ih_src;
67752           locport= tcp_hdr->th_dstport;
67753           remport= tcp_hdr->th_srcport;
67754           if (!remport)   /* This can interfere with a listen, so we reject it
67755                            * by clearing the requested port
67756                            */
67757                   locport= 0;
67758
67759   #if DEBUG & 256
67760    { where(); printf("find_best_conn(locport= %u, locaddr= ",
67761           ntohs(locport)); writeIpAddr(locaddr);
67762           printf("\nremport= %u, remaddr= ", ntohs(remport));
67763           writeIpAddr(remaddr); printf(")\n"); }
67764   #endif
67765           best_level= 0;
67766           best_conn= NULL;
67767           listen_conn= NULL;
67768           for (i= TCP_PORT_NR, tcp_conn= tcp_conn_table+i;
67769                   i<TCP_CONN_NR; i++, tcp_conn++)
67770                   /* the first TCP_PORT_NR connection are reserved for
67771                           RSTs */
67772           {
67773                   if (!(tcp_conn->tc_flags & TCF_INUSE))
67774                           continue;
67775                   /* First fast check for open connections. */
67776                   if (tcp_conn->tc_locaddr == locaddr &&
67777                           tcp_conn->tc_locport == locport &&
67778                           tcp_conn->tc_remport == remport &&
67779                           tcp_conn->tc_remaddr == remaddr &&
67780                           tcp_conn->tc_mainuser)
67781                           return tcp_conn;
67782
67783                   /* Now check for listens and abandoned connections. */
67784                   if (tcp_conn->tc_locaddr != locaddr)
```

```
67785                   {
67786   #if DEBUG
67787    { where(); printf("conn %d: wrong locaddr\n",i); }
67788   #endif
67789                           continue;
67790                   }
67791                   new_level= 0;
67792                   if (tcp_conn->tc_locport)
67793                   {
67794                           if (locport != tcp_conn->tc_locport)
67795                           {
67796   #if DEBUG & 256
67797    { where(); printf("conn %d: wrong locport(%u)\n",i,
67798           ntohs(tcp_conn->tc_locport)); }
67799   #endif
67800                                   continue;
67801                           }
67802                           new_level += 4;
67803                   }
67804                   if (tcp_conn->tc_remport)
67805                   {
67806                           if (remport != tcp_conn->tc_remport)
67807                           {
67808   #if DEBUG & 256
67809    { where(); printf("conn %d: wrong remport\n",i); }
67810   #endif
67811                                   continue;
67812                           }
67813                           new_level += 1;
67814                   }
67815                   if (tcp_conn->tc_remaddr)
67816                   {
67817                           if (remaddr != tcp_conn->tc_remaddr)
67818                           {
67819   #if DEBUG & 256
67820    { where(); printf("conn %d: wrong remaddr\n",i); }
67821   #endif
67822                                   continue;
67823                           }
67824                           new_level += 2;
67825                   }
67826                   if (new_level<best_level)
67827                           continue;
67828                   if (new_level != 7 && tcp_conn->tc_state != TCS_LISTEN)
67829                           continue;
67830                   if (new_level == 7 && !tcp_conn->tc_mainuser)
67831                           /* We found an abandoned connection */
67832                   {
67833   assert (!best_conn);
67834                           best_conn= tcp_conn;
67835                           continue;
67836                   }
67837                   if (!(tcp_hdr->th_flags & THF_SYN))
67838                           continue;
67839                   best_level= new_level;
67840                   listen_conn= tcp_conn;
67841           }
67842           if (!best_conn && !listen_conn)
67843           {
67844   #if DEBUG & 256
```

```
67845     { where(); printf("refusing connection for locport= %u, locaddr= ",
67846             ntohs(locport)); writeIpAddr(locaddr);
67847             printf("\nremport= %u, remaddr= ", ntohs(remport));
67848             writeIpAddr(remaddr); printf("\n"); }
67849     #endif
67850                     for (i=0, tcp_conn= tcp_conn_table; i<TCP_PORT_NR;
67851                             i++, tcp_conn++)
67852                             /* find valid port to send RST */
67853                             if ((tcp_conn->tc_flags & TCF_INUSE) &&
67854                                     tcp_conn->tc_locaddr==locaddr)
67855                             {
67856                                     break;
67857                             }
67858     assert (tcp_conn);
67859     assert (tcp_conn->tc_state == TCS_CLOSED);
67860                     tcp_conn->tc_locport= locport;
67861                     tcp_conn->tc_locaddr= locaddr;
67862                     tcp_conn->tc_remport= remport;
67863                     tcp_conn->tc_remaddr= remaddr;
67864     assert (!tcp_conn->tc_mainuser);
67865     assert (!tcp_conn->tc_readuser);
67866     assert (!tcp_conn->tc_writeuser);
67867                     return tcp_conn;
67868
67869             }
67870             if (best_conn)
67871             {
67872     assert(!best_conn->tc_mainuser);
67873                     if (!listen_conn)
67874                             return best_conn;
67875                     tcp_fd= listen_conn->tc_mainuser;
67876
67877     assert(tcp_fd && tcp_fd == listen_conn->tc_connuser &&
67878             tcp_fd->tf_conn == listen_conn);
67879
67880                     if (best_conn->tc_state != TCS_CLOSED)
67881                             tcp_close_connection(best_conn, ENOCONN);
67882                     best_conn->tc_state= TCS_LISTEN;
67883     #if DEBUG
67884      { where(); tcp_write_state(best_conn); }
67885     #endif
67886                     best_conn->tc_mainuser= best_conn->tc_connuser=
67887                             tcp_fd;
67888                     best_conn->tc_flags= listen_conn->tc_flags;
67889                     tcp_fd->tf_conn= best_conn;
67890                     listen_conn->tc_flags= TCF_EMPTY;
67891     #if DEBUG & 16
67892      { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
67893             listen_conn-tcp_conn_table, listen_conn->tc_flags); }
67894     #endif
67895                     return best_conn;
67896             }
67897     assert (listen_conn);
67898             return listen_conn;
67899     }


67902     PUBLIC void tcp_reply_ioctl(tcp_fd, reply)
67903     tcp_fd_t *tcp_fd;
67904     int reply;
```

```
67905   {
67906   #if DEBUG & 256
67907    { where(); printf("tcp_reply_ioctl called\n"); }
67908   #endif
67909   #if DEBUG
67910    if (!(tcp_fd->tf_flags & TFF_IOCTL_IP))
67911    { where(); printf("not TFF_IOCTL_IP\n"); }
67912   #endif
67913   assert (tcp_fd->tf_flags & TFF_IOCTL_IP);
67914   #if DEBUG
67915   if (tcp_fd->tf_ioreq != NWIOTCPSHUTDOWN && tcp_fd->tf_ioreq != NWIOTCPLISTEN &&
67916           tcp_fd->tf_ioreq != NWIOTCPCONN)
67917    { where(); printf("wrong value in ioreq (0x%lx)\n", tcp_fd->tf_ioreq); }
67918   #endif
67919   assert (tcp_fd->tf_ioreq == NWIOTCPSHUTDOWN ||
67920           tcp_fd->tf_ioreq == NWIOTCPLISTEN || tcp_fd->tf_ioreq == NWIOTCPCONN);
67921
67922           tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
67923           reply_thr_get (tcp_fd, reply, TRUE);
67924   }

67926   PUBLIC void tcp_reply_write(tcp_fd, reply)
67927   tcp_fd_t *tcp_fd;
67928   size_t reply;
67929   {
67930           assert (tcp_fd->tf_flags & TFF_WRITE_IP);
67931
67932           tcp_fd->tf_flags &= ~TFF_WRITE_IP;
67933           reply_thr_get (tcp_fd, reply, FALSE);
67934   }

67936   PUBLIC void tcp_reply_read(tcp_fd, reply)
67937   tcp_fd_t *tcp_fd;
67938   size_t reply;
67939   {
67940           assert (tcp_fd->tf_flags & TFF_READ_IP);
67941
67942   #if DEBUG & 256
67943    { where(); printf("tcp_reply_read(.., %d)\n", reply); }
67944   #endif
67945           tcp_fd->tf_flags &= ~TFF_READ_IP;
67946           reply_thr_put (tcp_fd, reply, FALSE);
67947   }

67949   PUBLIC int tcp_write(fd, count)
67950   int fd;
67951   size_t count;
67952   {
67953           tcp_fd_t *tcp_fd;
67954           tcp_conn_t *tcp_conn;
67955
67956           tcp_fd= &tcp_fd_table[fd];
67957
67958           assert (tcp_fd->tf_flags & TFF_INUSE);
67959
67960           if (!(tcp_fd->tf_flags & TFF_CONNECTED))
67961           {
67962                   reply_thr_get (tcp_fd, ENOTCONN, FALSE);
67963                   return NW_OK;
67964           }
```

```
67965           tcp_conn= tcp_fd->tf_conn;
67966           if (tcp_conn->tc_flags & TCF_FIN_SENT)
67967           {
67968   #if DEBUG & 16
67969    { where(); printf("replying ESHUTDOWN for connection %d and fd %d\n",
67970           tcp_conn-tcp_conn_table, tcp_fd-tcp_fd_table); }
67971   #endif
67972                   reply_thr_get (tcp_fd, ESHUTDOWN, FALSE);
67973                   return NW_OK;
67974           }
67975
67976           tcp_fd->tf_flags |= TFF_WRITE_IP;
67977           tcp_fd->tf_write_offset= 0;
67978           tcp_fd->tf_write_count= count;
67979
67980           if (tcp_conn->tc_writeuser)
67981                   return NW_SUSPEND;
67982
67983           tcp_conn->tc_writeuser= tcp_fd;
67984           tcp_restart_fd_write (tcp_conn);
67985           if (!(tcp_fd->tf_flags & TFF_WRITE_IP))
67986                   return NW_OK;
67987           else
67988                   return NW_SUSPEND;
67989   }

67991   PUBLIC int tcp_read(fd, count)
67992   int fd;
67993   size_t count;
67994   {
67995           tcp_fd_t *tcp_fd;
67996           tcp_conn_t *tcp_conn;
67997
67998   #if DEBUG & 256
67999    { where(); printf("tcp_read(%d, %u)\n", fd, count); }
68000   #endif
68001           tcp_fd= &tcp_fd_table[fd];
68002
68003           assert (tcp_fd->tf_flags & TFF_INUSE);
68004
68005           if (!(tcp_fd->tf_flags & TFF_CONNECTED))
68006           {
68007                   reply_thr_put (tcp_fd, ENOTCONN, FALSE);
68008                   return NW_OK;
68009           }
68010           tcp_conn= tcp_fd->tf_conn;
68011
68012           tcp_fd->tf_flags |= TFF_READ_IP;
68013           tcp_fd->tf_read_offset= 0;
68014           tcp_fd->tf_read_count= count;
68015
68016           if (tcp_conn->tc_readuser)
68017                   return NW_SUSPEND;
68018
68019           tcp_conn->tc_readuser= tcp_fd;
68020           tcp_restart_fd_read (tcp_conn);
68021           if (!(tcp_fd->tf_flags & TFF_READ_IP))
68022                   return NW_OK;
68023           else
68024                   return NW_SUSPEND;
```

```
68025   }

68027   /*
68028   tcp_restart_connect
68029
68030   reply the success or failure of a connect to the user.
68031   */
68032
68033
68034   PUBLIC void tcp_restart_connect(tcp_fd)
68035   tcp_fd_t *tcp_fd;
68036   {
68037           tcp_conn_t *tcp_conn;
68038           int reply;
68039
68040   #if DEBUG & 256
68041    { where(); printf("tcp_restart_connect called\n"); }
68042   #endif
68043   assert (tcp_fd->tf_flags & TFF_IOCTL_IP);
68044   assert (tcp_fd->tf_ioreq == NWIOTCPLISTEN || tcp_fd->tf_ioreq == NWIOTCPCONN);
68045
68046           tcp_conn= tcp_fd->tf_conn;
68047
68048   assert (tcp_conn->tc_connuser == tcp_fd);
68049
68050           if (tcp_conn->tc_state == TCS_CLOSED)
68051                   reply= tcp_conn->tc_error;
68052           else
68053           {
68054                   tcp_fd->tf_flags |= TFF_CONNECTED;
68055                   reply= NW_OK;
68056           }
68057           tcp_conn->tc_connuser= 0;
68058   #if DEBUG & 256
68059    { where(); printf("tcp_conn_table[%d].tc_connuser= 0x%x\n", tcp_conn-
68060           tcp_conn_table, tcp_conn->tc_connuser); }
68061   #endif
68062           tcp_reply_ioctl (tcp_fd, reply);
68063   }

68065   PUBLIC void tcp_close(fd)
68066   int fd;
68067   {
68068           tcp_fd_t *tcp_fd;
68069           tcp_conn_t *tcp_conn;
68070
68071           tcp_fd= &tcp_fd_table[fd];
68072
68073   #if DEBUG
68074    { if (!(tcp_fd->tf_flags & TFF_INUSE)) { where(); printf("not inuse\n"); }
68075      if (tcp_fd->tf_flags & TFF_IOCTL_IP) { where(); printf("ioctl ip\n"); }
68076      if (tcp_fd->tf_flags & TFF_READ_IP) { where(); printf("read ip\n"); }
68077      if (tcp_fd->tf_flags & TFF_WRITE_IP) { where(); printf("write ip\n"); }
68078    }
68079   #endif
68080   assert (tcp_fd->tf_flags & TFF_INUSE);
68081   assert (!(tcp_fd->tf_flags & (TFF_IOCTL_IP|TFF_READ_IP|TFF_WRITE_IP)));
68082
68083           tcp_fd->tf_flags &= ~TFF_INUSE;
68084           if (!tcp_fd->tf_conn)
```

```
68085                   return;
68086
68087           tcp_conn= tcp_fd->tf_conn;
68088           close_mainuser(tcp_conn, tcp_fd);
68089           tcp_shutdown (tcp_conn);
68090   }

68092   PUBLIC int tcp_cancel(fd, which_operation)
68093   int fd;
68094   int which_operation;
68095   {
68096           tcp_fd_t *tcp_fd;
68097           tcp_conn_t *tcp_conn;
68098           int i;
68099           int type;
68100
68101   #if DEBUG & 256
68102    { where(); printf("tcp_cancel(%d, %d)\n", fd, which_operation); }
68103   #endif
68104           tcp_fd= &tcp_fd_table[fd];
68105
68106           assert (tcp_fd->tf_flags & TFF_INUSE);
68107
68108           tcp_conn= tcp_fd->tf_conn;
68109
68110           switch (which_operation)
68111           {
68112           case SR_CANCEL_WRITE:
68113   assert (tcp_fd->tf_flags & TFF_WRITE_IP);
68114                   tcp_fd->tf_flags &= ~TFF_WRITE_IP;
68115                   if (tcp_conn->tc_writeuser == tcp_fd)
68116                   {
68117                           tcp_conn->tc_writeuser= 0;
68118                           tcp_restart_fd_write (tcp_conn);
68119                   }
68120                   if (tcp_fd->tf_write_count)
68121                           reply_thr_get (tcp_fd, tcp_fd->tf_write_count, FALSE);
68122                   else
68123                           reply_thr_get (tcp_fd, EINTR, FALSE);
68124                   break;
68125           case SR_CANCEL_READ:
68126   assert (tcp_fd->tf_flags & TFF_READ_IP);
68127                   tcp_fd->tf_flags &= ~TFF_READ_IP;
68128                   if (tcp_conn->tc_readuser == tcp_fd)
68129                   {
68130                           tcp_conn->tc_readuser= 0;
68131                           tcp_restart_fd_read (tcp_conn);
68132                   }
68133                   if (tcp_fd->tf_read_count)
68134                           reply_thr_put (tcp_fd, tcp_fd->tf_read_count, FALSE);
68135                   else
68136                           reply_thr_put (tcp_fd, EINTR, FALSE);
68137                   break;
68138           case SR_CANCEL_IOCTL:
68139   assert (tcp_fd->tf_flags & TFF_IOCTL_IP);
68140                   tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
68141
68142                   type= tcp_fd->tf_ioreq & IOCTYPE_MASK;
68143                   switch (type)
68144                   {
```

```
68145                   case NWIOGTCPCONF & IOCTYPE_MASK:
68146                           reply_thr_put (tcp_fd, EINTR, TRUE);
68147                           break;
68148                   case NWIOSTCPCONF & IOCTYPE_MASK:
68149                   case NWIOTCPSHUTDOWN & IOCTYPE_MASK:
68150                   case NWIOTCPATTACH & IOCTYPE_MASK:
68151                           reply_thr_get (tcp_fd, EINTR, TRUE);
68152                           break;
68153                   case NWIOTCPCONN & IOCTYPE_MASK:
68154                   case NWIOTCPLISTEN & IOCTYPE_MASK:
68155   assert (tcp_conn->tc_connuser == tcp_fd);
68156                           tcp_conn->tc_connuser= 0;
68157                           tcp_conn->tc_mainuser= 0;
68158                           tcp_close_connection(tcp_conn, ENOCONN);
68159                           reply_thr_get (tcp_fd, EINTR, TRUE);
68160                           break;
68161                   default:
68162                           ip_warning(( "unknown ioctl inprogress: 0x%x",
68163                                   tcp_fd->tf_ioreq ));
68164                           reply_thr_get (tcp_fd, EINTR, TRUE);
68165                           break;
68166                   }
68167                   break;
68168           default:
68169                   ip_panic(( "unknown cancel request" ));
68170                   break;
68171           }
68172           return NW_OK;
68173   }

68175   /*
68176   tcp_connect
68177   */
68178
68179   PRIVATE int tcp_connect(tcp_fd)
68180   tcp_fd_t *tcp_fd;
68181   {
68182           tcp_conn_t *tcp_conn;
68183           int state;
68184
68185           if (!(tcp_fd->tf_flags & TFF_OPTSET))
68186           {
68187   #if DEBUG
68188    { where(); }
68189   #endif
68190                   tcp_reply_ioctl(tcp_fd, EBADMODE);
68191                   return NW_OK;
68192           }
68193           if (tcp_fd->tf_flags & TFF_CONNECT)
68194           {
68195                   tcp_reply_ioctl(tcp_fd, EISCONN);
68196                   return NW_OK;
68197           }
68198           if ((tcp_fd->tf_tcpconf.nwtc_flags & (NWTC_SET_RA|NWTC_SET_RP))
68199                   != (NWTC_SET_RA|NWTC_SET_RP))
68200           {
68201   #if DEBUG
68202    { where(); printf("tcp_fd_table[%d].tf_tcpconf.nwtc_flags= 0x%x\n",
68203           tcp_fd-tcp_fd_table, tcp_fd->tf_tcpconf.nwtc_flags); }
68204   #endif
```

```
68205                   tcp_reply_ioctl(tcp_fd, EBADMODE);
68206                   return NW_OK;
68207           }
68208
68209   assert(!tcp_fd->tf_conn);
68210           tcp_conn= find_conn_entry(tcp_fd->tf_tcpconf.nwtc_locport,
68211                   tcp_fd->tf_port->tp_ipaddr,
68212                   tcp_fd->tf_tcpconf.nwtc_remport,
68213                   tcp_fd->tf_tcpconf.nwtc_remaddr);
68214           if (tcp_conn)
68215           {
68216                   if (tcp_conn->tc_mainuser)
68217                   {
68218                           tcp_reply_ioctl(tcp_fd, EADDRINUSE);
68219                           return NW_OK;
68220                   }
68221           }
68222           else
68223           {
68224                   tcp_conn= find_empty_conn();
68225                   if (!tcp_conn)
68226                   {
68227                           tcp_reply_ioctl(tcp_fd, EAGAIN);
68228                           return NW_OK;
68229                   }
68230           }
68231           tcp_fd->tf_conn= tcp_conn;
68232           return tcp_su4connect(tcp_fd);
68233   }

68235   /*
68236   tcp_su4connect
68237   */
68238
68239   PRIVATE int tcp_su4connect(tcp_fd)
68240   tcp_fd_t *tcp_fd;
68241   {
68242           tcp_conn_t *tcp_conn;
68243           acc_t *tmp_acc;
68244
68245           tcp_conn= tcp_fd->tf_conn;
68246
68247           tcp_conn->tc_locport= tcp_fd->tf_tcpconf.nwtc_locport;
68248           tcp_conn->tc_locaddr= tcp_fd->tf_port->tp_ipaddr;
68249
68250   assert (tcp_fd->tf_tcpconf.nwtc_flags & NWTC_SET_RP);
68251   assert (tcp_fd->tf_tcpconf.nwtc_flags & NWTC_SET_RA);
68252
68253           tcp_conn->tc_remport= tcp_fd->tf_tcpconf.nwtc_remport;
68254           tcp_conn->tc_remaddr= tcp_fd->tf_tcpconf.nwtc_remaddr;
68255           tcp_conn->tc_mainuser= tcp_fd;
68256
68257           tcp_setup_conn(tcp_conn);
68258           tcp_conn->tc_port= tcp_fd->tf_port;
68259           tcp_conn->tc_connuser= tcp_fd;
68260   #if DEBUG & 256
68261    { where(); printf("tcp_conn_table[%d].tc_connuser= 0x%x\n", tcp_conn-
68262           tcp_conn_table, tcp_conn->tc_connuser); }
68263   #endif
68264           tcp_conn->tc_orglisten= FALSE;
```

```
68265           tcp_conn->tc_state= TCS_SYN_SENT;
68266   #if DEBUG & 16
68267    { where(); tcp_write_state(tcp_conn); }
68268   #endif
68269           tcp_restart_write (tcp_conn);
68270
68271           if (tcp_conn->tc_connuser)
68272                   return NW_SUSPEND;
68273           else
68274                   return NW_OK;
68275   }

68277   PRIVATE void close_mainuser(tcp_conn, tcp_fd)
68278   tcp_conn_t *tcp_conn;
68279   tcp_fd_t *tcp_fd;
68280   {
68281           int i;
68282
68283           if (tcp_conn->tc_mainuser != tcp_fd)
68284                   return;
68285
68286           tcp_conn->tc_mainuser= 0;
68287           assert (tcp_conn->tc_writeuser != tcp_fd);
68288           assert (tcp_conn->tc_readuser != tcp_fd);
68289           assert (tcp_conn->tc_connuser != tcp_fd);
68290
68291           for (i= 0, tcp_fd= tcp_fd_table; i<TCP_FD_NR; i++, tcp_fd++)
68292           {
68293                   if (!(tcp_fd->tf_flags & TFF_INUSE))
68294                           continue;
68295                   if (tcp_fd->tf_conn != tcp_conn)
68296                           continue;
68297                   tcp_conn->tc_mainuser= tcp_fd;
68298                   return;
68299           }
68300   }


68303   PRIVATE int conn_right4fd(tcp_conn, tcp_fd)
68304   tcp_fd_t *tcp_fd;
68305   tcp_conn_t *tcp_conn;
68306   {
68307           unsigned long flags;
68308
68309           flags= tcp_fd->tf_tcpconf.nwtc_flags;
68310
68311           if (tcp_fd->tf_tcpconf.nwtc_locport != tcp_conn->tc_locport)
68312                   return FALSE;
68313
68314           if ((flags & NWTC_SET_RA) && tcp_fd->tf_tcpconf.nwtc_remaddr !=
68315                   tcp_conn->tc_remaddr)
68316                   return FALSE;
68317
68318           if ((flags & NWTC_SET_RP) && tcp_fd->tf_tcpconf.nwtc_remport !=
68319                   tcp_conn->tc_remport)
68320                   return FALSE;
68321
68322           if (tcp_fd->tf_port != tcp_conn->tc_port)
68323                   return FALSE;
68324
```

```
68325           return TRUE;
68326   }

68328   PRIVATE int tcp_attache(tcp_fd)
68329   tcp_fd_t *tcp_fd;
68330   {
68331           tcp_conn_t *tcp_conn;
68332           int state;
68333
68334           if (!(tcp_fd->tf_flags & TFF_OPTSET))
68335           {
68336                   tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
68337                   reply_thr_get(tcp_fd, EBADMODE, TRUE);
68338                   return NW_OK;
68339           }
68340           if (tcp_fd->tf_flags & TFF_CONNECT)
68341           {
68342                   tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
68343                   reply_thr_get(tcp_fd, EISCONN, TRUE);
68344                   return NW_OK;
68345           }
68346           if ((tcp_fd->tf_tcpconf.nwtc_flags & (NWTC_SET_RA|NWTC_SET_RP))
68347                   != (NWTC_SET_RA|NWTC_SET_RP))
68348           {
68349                   tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
68350                   reply_thr_get(tcp_fd, EBADMODE, TRUE);
68351                   return NW_OK;
68352           }
68353
68354           tcp_conn= find_conn_entry(tcp_fd->tf_tcpconf.nwtc_locport,
68355                   tcp_fd->tf_port->tp_ipaddr,
68356                   tcp_fd->tf_tcpconf.nwtc_remport,
68357                   tcp_fd->tf_tcpconf.nwtc_remaddr);
68358           if (!tcp_conn)
68359           {
68360   #if DEBUG
68361    { where(); printf("conn_entry not found\n"); }
68362   #endif
68363                   tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
68364                   reply_thr_get (tcp_fd, ENOCONN, TRUE);
68365                   return NW_OK;
68366           }
68367           assert (tcp_conn->tc_flags & TCF_INUSE);
68368           state= tcp_conn->tc_state;
68369           if (state == TCS_CLOSED || state == TCS_LISTEN || state ==
68370                   TCS_SYN_SENT || state == TCS_SYN_RECEIVED)
68371           {
68372   #if DEBUG
68373    { where(); printf("conn_entry in wrong state: ");
68374           tcp_write_state(tcp_conn); printf("\n"); }
68375   #endif
68376                   tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
68377                   reply_thr_get (tcp_fd, ENOCONN, TRUE);
68378                   return NW_OK;
68379           }
68380           tcp_fd->tf_conn= tcp_conn;
68381           tcp_fd->tf_flags |= TFF_CONNECTED;
68382           tcp_fd->tf_flags |= TFF_PUSH_DATA;      /* XXX */
68383           return NW_OK;
68384   }
```

```
68386   /*
68387   tcp_listen
68388   */
68389
68390   PRIVATE int tcp_listen(tcp_fd)
68391   tcp_fd_t *tcp_fd;
68392   {
68393           tcp_conn_t *tcp_conn;
68394           int state;
68395
68396   #if DEBUG & 256
68397    { where(); printf("tcp_listen(&tcp_fd_table[%d]) called\n", tcp_fd-
68398           tcp_fd_table); }
68399   #endif
68400           if (!(tcp_fd->tf_flags & TFF_OPTSET))
68401           {
68402                   tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
68403                   reply_thr_get(tcp_fd, EBADMODE, TRUE);
68404                   return NW_OK;
68405           }
68406           if (tcp_fd->tf_flags & TFF_CONNECT)
68407           {
68408                   tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
68409                   reply_thr_get(tcp_fd, EISCONN, TRUE);
68410                   return NW_OK;
68411           }
68412           tcp_conn= tcp_fd->tf_conn;
68413   assert(!tcp_conn);
68414           if ((tcp_fd->tf_tcpconf.nwtc_flags & (NWTC_SET_RA|NWTC_SET_RP))
68415                   == (NWTC_SET_RA|NWTC_SET_RP))
68416           {
68417                   tcp_conn= find_conn_entry(
68418                           tcp_fd->tf_tcpconf.nwtc_locport,
68419                           tcp_fd->tf_port->tp_ipaddr,
68420                           tcp_fd->tf_tcpconf.nwtc_remport,
68421                           tcp_fd->tf_tcpconf.nwtc_remaddr);
68422                   if (tcp_conn)
68423                   {
68424                           if (tcp_conn->tc_mainuser)
68425                           {
68426                                   tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
68427                                   reply_thr_get (tcp_fd, EADDRINUSE, TRUE);
68428                                   return NW_OK;
68429                           }
68430                           tcp_fd->tf_conn= tcp_conn;
68431                           return tcp_su4listen(tcp_fd);
68432                   }
68433           }
68434           tcp_conn= find_empty_conn();
68435           if (!tcp_conn)
68436           {
68437                   tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
68438                   reply_thr_get (tcp_fd, EAGAIN, TRUE);
68439                   return NW_OK;
68440           }
68441           tcp_fd->tf_conn= tcp_conn;
68442   #if DEBUG & 256
68443    { where(); printf("tcp_conn_table[%d].tc_connuser= 0x%x\n", tcp_conn-
68444           tcp_conn_table, tcp_conn->tc_connuser); }
```

```
68445   #endif
68446           return tcp_su4listen(tcp_fd);
68447   }

68449   PRIVATE void tcp_buffree (priority, reqsize)
68450   int priority;
68451   size_t reqsize;
68452   {
68453           int i;
68454           tcp_conn_t *tcp_conn;
68455
68456           if (priority <TCP_PRI_FRAG2SEND)
68457                   return;
68458
68459   #if DEBUG & 256
68460    { where(); printf("tcp_buffree called\n"); }
68461   #endif
68462
68463           for (i=0, tcp_conn= tcp_conn_table; i<TCP_CONN_NR; i++,
68464                   tcp_conn++)
68465           {
68466                   if (!(tcp_conn->tc_flags & TCF_INUSE))
68467                           continue;
68468                   if (!tcp_conn->tc_frag2send)
68469                           continue;
68470                   bf_afree(tcp_conn->tc_frag2send);
68471                   tcp_conn->tc_frag2send= 0;
68472                   if (bf_free_buffsize >= reqsize)
68473                           return;
68474           }
68475
68476           if (priority <TCP_PRI_CONNwoUSER)
68477                   return;
68478
68479           for (i=0, tcp_conn= tcp_conn_table; i<TCP_CONN_NR; i++,
68480                   tcp_conn++)
68481           {
68482                   if (!(tcp_conn->tc_flags & TCF_INUSE))
68483                           continue;
68484                   if (tcp_conn->tc_mainuser)
68485                           continue;
68486                   if (tcp_conn->tc_state == TCS_CLOSED)
68487                           continue;
68488   #if DEBUG
68489    { where(); printf("calling tcp_close_connection (out of memory)\n"); }
68490   #endif
68491                   tcp_close_connection (tcp_conn, EOUTOFBUFS);
68492                   if (bf_free_buffsize >= reqsize)
68493                           return;
68494           }
68495
68496           if (priority <TCP_PRI_CONN_INUSE)
68497                   return;
68498
68499           for (i=0, tcp_conn= tcp_conn_table; i<TCP_CONN_NR; i++,
68500                   tcp_conn++)
68501           {
68502                   if (!(tcp_conn->tc_flags & TCF_INUSE))
68503                           continue;
68504                   if (tcp_conn->tc_state == TCS_CLOSED)
```

```
68505                           continue;
68506   #if DEBUG
68507    { where(); printf("calling tcp_close_connection (out of memory)\n"); }
68508   #endif
68509                   tcp_close_connection (tcp_conn, EOUTOFBUFS);
68510                   if (bf_free_buffsize >= reqsize)
68511                           return;
68512           }
68513   }

68515   PRIVATE void tcp_notreach(pack)
68516   acc_t *pack;
68517   {
68518           acc_t *tcp_pack;
68519           ip_hdr_t *ip_hdr;
68520           tcp_hdr_t *tcp_hdr;
68521           tcp_conn_t *tcp_conn;
68522           int ip_hdr_size;
68523           int new_ttl;
68524
68525           pack->acc_linkC++;
68526           pack= bf_packIffLess(pack, IP_MIN_HDR_SIZE);
68527           ip_hdr= (ip_hdr_t *)ptr2acc_data(pack);
68528           ip_hdr_size= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
68529
68530           tcp_pack= bf_cut(pack, ip_hdr_size, TCP_MIN_HDR_SIZE);
68531           tcp_pack= bf_packIffLess(tcp_pack, TCP_MIN_HDR_SIZE);
68532           tcp_hdr= (tcp_hdr_t *)ptr2acc_data(tcp_pack);
68533
68534           tcp_conn= find_conn_entry( tcp_hdr->th_srcport, ip_hdr->ih_src,
68535                   tcp_hdr->th_dstport, ip_hdr->ih_dst);
68536           if (tcp_conn)
68537           {
68538                   new_ttl= tcp_conn->tc_ttl;
68539                   if (new_ttl == IP_MAX_TTL)
68540                   {
68541   #if DEBUG
68542    { where(); printf("calling tcp_close_connection\n"); }
68543   #endif
68544                           tcp_close_connection(tcp_conn, EDSTNOTRCH);
68545                           bf_afree(pack);
68546                           bf_afree(tcp_pack);
68547                           return;
68548                   }
68549                   new_ttl *= 2;
68550                   if (new_ttl> IP_MAX_TTL)
68551                           new_ttl= IP_MAX_TTL;
68552                   tcp_conn->tc_ttl= new_ttl;
68553                   tcp_conn->tc_no_retrans= 0;
68554           }
68555           else
68556           {
68557   #if DEBUG
68558    { where(); printf("got a EDSTNOTRCH for non existing connection\n"); }
68559   #endif
68560           }
68561           bf_afree(pack);
68562           bf_afree(tcp_pack);
68563   }
```

```
68565   /*
68566   tcp_setup_conn
68567   */
68568
68569   PRIVATE void tcp_setup_conn(tcp_conn)
68570   tcp_conn_t *tcp_conn;
68571   {
68572   assert(!tcp_conn->tc_readuser);
68573   assert(!tcp_conn->tc_writeuser);
68574   assert(!tcp_conn->tc_connuser);
68575           if (tcp_conn->tc_flags & TCF_INUSE)
68576           {
68577   assert (tcp_conn->tc_state == TCS_CLOSED ||
68578           tcp_conn->tc_state == TCS_TIME_WAIT);
68579   assert (!tcp_conn->tc_send_data);
68580                   if (tcp_conn->tc_senddis < get_time())
68581                           tcp_conn->tc_ISS= 0;
68582           }
68583           else
68584           {
68585                   tcp_conn->tc_senddis= 0;
68586                   tcp_conn->tc_ISS= 0;
68587                   tcp_conn->tc_tos= TCP_DEF_TOS;
68588                   tcp_conn->tc_ttl= TCP_DEF_TTL;
68589                   tcp_conn->tc_rcv_wnd= TCP_MAX_WND_SIZE;
68590                   tcp_conn->tc_urg_wnd= TCP_DEF_URG_WND;
68591           }
68592           if (!tcp_conn->tc_ISS)
68593           {
68594                   tcp_conn->tc_ISS= (get_time()/HZ)*ISS_INC_FREQ;
68595           }
68596           tcp_conn->tc_SND_UNA= tcp_conn->tc_ISS;
68597           tcp_conn->tc_SND_TRM= tcp_conn->tc_ISS;
68598           tcp_conn->tc_SND_NXT= tcp_conn->tc_ISS+1;
68599           tcp_conn->tc_SND_UP= tcp_conn->tc_ISS;
68600           tcp_conn->tc_SND_PSH= tcp_conn->tc_ISS;
68601           tcp_conn->tc_SND_WL2= tcp_conn->tc_ISS;
68602           tcp_conn->tc_IRS= 0;
68603           tcp_conn->tc_SND_WL1= tcp_conn->tc_IRS;
68604           tcp_conn->tc_RCV_LO= tcp_conn->tc_IRS;
68605           tcp_conn->tc_RCV_NXT= tcp_conn->tc_IRS;
68606           tcp_conn->tc_RCV_HI= tcp_conn->tc_IRS;
68607           tcp_conn->tc_RCV_UP= tcp_conn->tc_IRS;
68608           tcp_conn->tc_rcvd_data= 0;
68609           tcp_conn->tc_rcv_queue= 0;
68610           tcp_conn->tc_send_data= 0;
68611           tcp_conn->tc_remipopt= 0;
68612           tcp_conn->tc_remtcpopt= 0;
68613           tcp_conn->tc_frag2send= 0;
68614           tcp_conn->tc_no_retrans= 0;
68615           tcp_conn->tc_max_no_retrans= TCP_DEF_MAX_NO_RETRANS;
68616           tcp_conn->tc_0wnd_to= 0;
68617           tcp_conn->tc_rtt= TCP_DEF_RTT;
68618           tcp_conn->tc_ett= 0;
68619           tcp_conn->tc_mss= TCP_DEF_MSS;
68620           tcp_conn->tc_error= NW_OK;
68621           tcp_conn->tc_snd_cwnd= tcp_conn->tc_SND_UNA + tcp_conn->tc_mss;
68622           tcp_conn->tc_snd_cthresh= TCP_MAX_WND_SIZE;
68623           tcp_conn->tc_snd_cinc= 0;
68624           tcp_conn->tc_snd_wnd= TCP_MAX_WND_SIZE;
```

```
68625           tcp_conn->tc_flags= TCF_INUSE;
68626   #if DEBUG & 256
68627    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
68628           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
68629   #endif
68630   }
```

++
src/inet/generic/tcp_lib.c
++

```
68700   /*
68701   tcp_lib.c
68702   */
68703
68704   #include "inet.h"
68705   #include "buf.h"
68706   #include "clock.h"
68707   #include "io.h"
68708   #include "type.h"
68709
68710   #include "assert.h"
68711   #include "tcp_int.h"
68712
68713   INIT_PANIC();
68714
68715   PUBLIC int tcp_LEmod4G (n1, n2)
68716   u32_t n1;
68717   u32_t n2;
68718   {
68719           return !((u32_t)(n2-n1) & 0x80000000L);
68720   }

68722   PUBLIC int tcp_GEmod4G (n1, n2)
68723   u32_t n1;
68724   u32_t n2;
68725   {
68726           return !((u32_t)(n1-n2) & 0x80000000L);
68727   }

68729   PUBLIC int tcp_Lmod4G (n1, n2)
68730   u32_t n1;
68731   u32_t n2;
68732   {
68733           return !!((u32_t)(n1-n2) & 0x80000000L);
68734   }

68736   PUBLIC int tcp_Gmod4G (n1, n2)
68737   u32_t n1;
68738   u32_t n2;
68739   {
68740           return !!((u32_t)(n2-n1) & 0x80000000L);
68741   }

68743   PUBLIC void tcp_extract_ipopt(tcp_conn, ip_hdr)
68744   tcp_conn_t *tcp_conn;
```

```
68745   ip_hdr_t *ip_hdr;
68746   {
68747           int ip_hdr_len;
68748
68749           ip_hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
68750           if (ip_hdr_len == IP_MIN_HDR_SIZE)
68751                   return;
68752
68753   #if DEBUG
68754    { where(); printf("ip_hdr options NOT supported (yet?)\n"); }
68755   #endif
68756   }

68758   PUBLIC void tcp_extract_tcpopt(tcp_conn, tcp_hdr)
68759   tcp_conn_t *tcp_conn;
68760   tcp_hdr_t *tcp_hdr;
68761   {
68762           int tcp_hdr_len;
68763
68764           tcp_hdr_len= (tcp_hdr->th_data_off & TH_DO_MASK) >> 2;
68765           if (tcp_hdr_len == TCP_MIN_HDR_SIZE)
68766                   return;
68767
68768   #if DEBUG  & 2
68769    { where(); printf("tcp_hdr options NOT supported (yet?)\n"); }
68770   #endif
68771   }

68773   PUBLIC u16_t tcp_pack_oneCsum(ip_pack, ip_pack_size)
68774   acc_t *ip_pack;
68775   size_t ip_pack_size;
68776   {
68777           ip_hdr_t *ip_hdr;
68778           tcp_hdr_t *tcp_hdr;
68779           size_t ip_hdr_len;
68780           acc_t *tcp_pack, *save_pack;
68781           u16_t prev;
68782           u8_t byte_buf[2];
68783           u16_t word_buf;
68784           int odd_byte;
68785           char *data_ptr;
68786           int length;
68787
68788
68789   assert (ip_pack);
68790           ip_pack->acc_linkC++;
68791
68792           ip_pack= bf_packIffLess(ip_pack, IP_MIN_HDR_SIZE);
68793           ip_hdr= (ip_hdr_t *)ptr2acc_data(ip_pack);
68794           ip_hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
68795
68796           prev= 0;
68797
68798           prev= oneC_sum(prev, (u16_t *)&ip_hdr->ih_src,
68799                   sizeof(ip_hdr->ih_src));
68800                   /* Source address in pseudo header */
68801
68802           prev= oneC_sum(prev, (u16_t *)&ip_hdr->ih_dst,
68803                   sizeof(ip_hdr->ih_dst));
68804                   /* Destination address in pseudo header */
```

```
68805
68806           byte_buf[0]= 0;
68807           byte_buf[1]= ip_hdr->ih_proto;
68808   assert(ip_hdr->ih_proto == IPPROTO_TCP);
68809
68810           prev= oneC_sum(prev, (u16_t *)byte_buf, 2);
68811                   /* Protocol and zero byte */
68812
68813           word_buf= htons(ntohs(ip_hdr->ih_length)-ip_hdr_len);
68814           prev= oneC_sum(prev, &word_buf, sizeof(word_buf));
68815   assert (ntohs(ip_hdr->ih_length) == ip_pack_size);
68816
68817           tcp_pack= bf_cut(ip_pack, ip_hdr_len, ip_pack_size-ip_hdr_len);
68818           bf_afree(ip_pack);
68819           tcp_pack= bf_packIffLess(tcp_pack, TCP_MIN_HDR_SIZE);
68820           tcp_hdr= (tcp_hdr_t *)ptr2acc_data(tcp_pack);
68821
68822           save_pack= tcp_pack;
68823
68824           odd_byte= FALSE;
68825           for (; tcp_pack; tcp_pack= tcp_pack->acc_next)
68826           {
68827
68828                   data_ptr= ptr2acc_data(tcp_pack);
68829                   length= tcp_pack->acc_length;
68830
68831                   if (!length)
68832                           continue;
68833                   if (odd_byte)
68834                   {
68835                           byte_buf[1]= *data_ptr;
68836                           prev= oneC_sum(prev, (u16_t *)byte_buf, 2);
68837                           data_ptr++;
68838                           length--;
68839                           odd_byte= FALSE;
68840                   }
68841                   if (length & 1)
68842                   {
68843                           odd_byte= TRUE;
68844                           length--;
68845                           byte_buf[0]= data_ptr[length];
68846                   }
68847                   if (!length)
68848                           continue;
68849                   prev= oneC_sum (prev, (u16_t *)data_ptr, length);
68850           }
68851           if (odd_byte)
68852                   prev= oneC_sum (prev, (u16_t *)byte_buf, 1);
68853           bf_afree(save_pack);
68854           return prev;
68855   }

68857   PUBLIC void tcp_get_ipopt(tcp_conn, ip_hdropt)
68858   tcp_conn_t *tcp_conn;
68859   ip_hdropt_t *ip_hdropt;
68860   {
68861           if (!tcp_conn->tc_remipopt)
68862           {
68863                   ip_hdropt->iho_opt_siz= 0;
68864                   return;
```

```
68865           }
68866   #if DEBUG
68867    { where(); printf("ip_hdr options NOT supported (yet?)\n"); }
68868   #endif
68869           ip_hdropt->iho_opt_siz= 0;
68870           return;
68871   }

68873   PUBLIC void tcp_get_tcpopt(tcp_conn, tcp_hdropt)
68874   tcp_conn_t *tcp_conn;
68875   tcp_hdropt_t *tcp_hdropt;
68876   {
68877           if (!tcp_conn->tc_remtcpopt)
68878           {
68879                   tcp_hdropt->tho_opt_siz= 0;
68880                   return;
68881           }
68882
68883   #if DEBUG
68884    { where(); printf("tcp_hdr options NOT supported (yet?)\n"); }
68885   #endif
68886           tcp_hdropt->tho_opt_siz= 0;
68887           return;
68888   }

68890   PUBLIC acc_t *tcp_make_header(tcp_conn, ref_ip_hdr, ref_tcp_hdr, data)
68891   tcp_conn_t *tcp_conn;
68892   ip_hdr_t **ref_ip_hdr;
68893   tcp_hdr_t **ref_tcp_hdr;
68894   acc_t *data;
68895   {
68896           ip_hdropt_t ip_hdropt;
68897           tcp_hdropt_t tcp_hdropt;
68898           ip_hdr_t *ip_hdr;
68899           tcp_hdr_t *tcp_hdr;
68900           acc_t *hdr_acc;
68901           char *ptr2hdr;
68902           int closed_connection;
68903
68904           closed_connection= (tcp_conn->tc_state == TCS_CLOSED);
68905
68906           tcp_get_ipopt (tcp_conn, &ip_hdropt);
68907           tcp_get_tcpopt (tcp_conn, &tcp_hdropt);
68908   assert (!(ip_hdropt.iho_opt_siz & 3));
68909   assert (!(tcp_hdropt.tho_opt_siz & 3));
68910
68911           if (IP_MIN_HDR_SIZE+ip_hdropt.iho_opt_siz+
68912                   TCP_MIN_HDR_SIZE+tcp_hdropt.tho_opt_siz <= BUF_S)
68913           {
68914                   hdr_acc= bf_memreq(IP_MIN_HDR_SIZE+
68915                           ip_hdropt.iho_opt_siz+TCP_MIN_HDR_SIZE+
68916                           tcp_hdropt.tho_opt_siz);
68917                   ptr2hdr= ptr2acc_data(hdr_acc);
68918
68919                   ip_hdr= (ip_hdr_t *)ptr2hdr;
68920                   ptr2hdr += IP_MIN_HDR_SIZE;
68921
68922                   if (ip_hdropt.iho_opt_siz)
68923                   {
68924   #if DEBUG
```

```
68925     { where(); printf("doing memcpy\n"); }
68926     #endif
68927                             memcpy(ptr2hdr, (char *)ip_hdropt.iho_data,
68928                                     ip_hdropt.iho_opt_siz);
68929                     }
68930                     ptr2hdr += ip_hdropt.iho_opt_siz;
68931
68932                     tcp_hdr= (tcp_hdr_t *)ptr2hdr;
68933                     ptr2hdr += TCP_MIN_HDR_SIZE;
68934
68935                     if (tcp_hdropt.tho_opt_siz)
68936                     {
68937     #if DEBUG
68938                      { where(); printf("doing memcpy\n"); }
68939     #endif
68940                             memcpy (ptr2hdr, (char *)tcp_hdropt.tho_data,
68941                                     tcp_hdropt.tho_opt_siz);
68942                     }
68943                     hdr_acc->acc_next= data;
68944             }
68945             else
68946             {
68947                     hdr_acc= bf_memreq(IP_MIN_HDR_SIZE+
68948                             ip_hdropt.iho_opt_siz);
68949                     ptr2hdr= ptr2acc_data(hdr_acc);
68950
68951                     ip_hdr= (ip_hdr_t *)ptr2hdr;
68952                     ptr2hdr += IP_MIN_HDR_SIZE;
68953
68954                     if (ip_hdropt.iho_opt_siz)
68955                     {
68956     #if DEBUG
68957                      { where(); printf("doing memcpy\n"); }
68958     #endif
68959                             memcpy (ptr2hdr, (char *)ip_hdropt.iho_data,
68960                                     ip_hdropt.iho_opt_siz);
68961                     }
68962
68963                     hdr_acc->acc_next= bf_memreq(TCP_MIN_HDR_SIZE+
68964                             tcp_hdropt.tho_opt_siz);
68965
68966                     ptr2hdr= ptr2acc_data(hdr_acc->acc_next);
68967                     tcp_hdr= (tcp_hdr_t *)ptr2hdr;
68968                     ptr2hdr += TCP_MIN_HDR_SIZE;
68969
68970                     if (tcp_hdropt.tho_opt_siz)
68971                     {
68972     #if DEBUG
68973                      { where(); printf("doing memcpy\n"); }
68974     #endif
68975                             memcpy (ptr2hdr, (char *)tcp_hdropt.tho_data,
68976                                     tcp_hdropt.tho_opt_siz);
68977                     }
68978                     hdr_acc->acc_next->acc_next= data;
68979             }
68980             if (!closed_connection && (tcp_conn->tc_state == TCS_CLOSED))
68981             {
68982     #if DEBUG
68983                      { where(); printf("connection closed while inuse\n"); }
68984     #endif
```

```
68985                   bf_afree(hdr_acc);
68986                   return 0;
68987           }
68988
68989           ip_hdr->ih_vers_ihl= (IP_MIN_HDR_SIZE+
68990                   ip_hdropt.iho_opt_siz) >> 2;
68991           ip_hdr->ih_tos= tcp_conn->tc_tos;
68992           ip_hdr->ih_ttl= tcp_conn->tc_ttl;
68993           ip_hdr->ih_proto= IPPROTO_TCP;
68994           ip_hdr->ih_src= tcp_conn->tc_locaddr;
68995           ip_hdr->ih_dst= tcp_conn->tc_remaddr;
68996           ip_hdr->ih_flags_fragoff= 0;
68997
68998           tcp_hdr->th_srcport= tcp_conn->tc_locport;
68999           tcp_hdr->th_dstport= tcp_conn->tc_remport;
69000           tcp_hdr->th_seq_nr= tcp_conn->tc_RCV_NXT;
69001           tcp_hdr->th_flags= 0;
69002           tcp_hdr->th_data_off= (TCP_MIN_HDR_SIZE+
69003                   tcp_hdropt.tho_opt_siz) << 2;
69004           tcp_hdr->th_window= htons(tcp_conn->tc_RCV_HI-
69005                   tcp_conn->tc_RCV_LO);
69006           tcp_hdr->th_chksum= 0;
69007           *ref_ip_hdr= ip_hdr;
69008           *ref_tcp_hdr= tcp_hdr;
69009           return hdr_acc;
69010   }

69012   PUBLIC void tcp_write_state (tcp_conn)
69013   tcp_conn_t *tcp_conn;
69014   {
69015           printf("tcp_conn_table[%d]->tc_state= ", tcp_conn-
69016                   tcp_conn_table);
69017           if (!(tcp_conn->tc_flags & TCF_INUSE))
69018           {
69019                   printf("not inuse\n");
69020                   return;
69021           }
69022           switch (tcp_conn->tc_state)
69023           {
69024           case TCS_CLOSED: printf("CLOSED"); break;
69025           case TCS_LISTEN: printf("LISTEN"); break;
69026           case TCS_SYN_RECEIVED: printf("SYN_RECEIVED"); break;
69027           case TCS_SYN_SENT: printf("SYN_SENT"); break;
69028           case TCS_ESTABLISHED: printf("ESTABLISHED"); break;
69029           case TCS_FIN_WAIT_1: printf("FIN_WAIT_1"); break;
69030           case TCS_FIN_WAIT_2: printf("FIN_WAIT_2"); break;
69031           case TCS_CLOSE_WAIT: printf("CLOSE_WAIT"); break;
69032           case TCS_CLOSING: printf("CLOSING"); break;
69033           case TCS_LAST_ACK: printf("LAST_ACK"); break;
69034           case TCS_TIME_WAIT: printf("TIME_WAIT"); break;
69035           default: printf("unknown (=%d)", tcp_conn->tc_state); break;
69036           }
69037           printf("\n");
69038   }

69040   PUBLIC int tcp_check_conn(tcp_conn)
69041   tcp_conn_t *tcp_conn;
69042   {
69043           int allright;
69044           u32_t lo_queue, hi_queue;
```

```
69045           int size;
69046
69047   #if DEBUG & 256
69048    { where(); printf("in tcp_check_conn\n"); }
69049   #endif
69050           allright= TRUE;
69051
69052           /* checking receive queue */
69053           lo_queue= tcp_conn->tc_RCV_LO;
69054           if (lo_queue == tcp_conn->tc_IRS)
69055                   lo_queue++;
69056           if (lo_queue == tcp_conn->tc_RCV_NXT && (tcp_conn->tc_flags &
69057                   TCF_FIN_RECV))
69058                   lo_queue--;
69059           hi_queue= tcp_conn->tc_RCV_NXT;
69060           if (hi_queue == tcp_conn->tc_IRS)
69061                   hi_queue++;
69062           if (tcp_conn->tc_flags & TCF_FIN_RECV)
69063                   hi_queue--;
69064
69065           size= hi_queue-lo_queue;
69066           if (size<0)
69067           {
69068                   printf("RCV_NXT-RCV_LO < 0\n");
69069                   allright= FALSE;
69070           }
69071           else if (!tcp_conn->tc_rcvd_data)
69072           {
69073                   if (size)
69074                   {
69075                           printf("RCV_NXT-RCV_LO != 0\n");
69076                           tcp_print_conn(tcp_conn);
69077                           printf("lo_queue= %lu, hi_queue= %lu\n",
69078                                   lo_queue, hi_queue);
69079                           allright= FALSE;
69080                   }
69081           }
69082           else if (size != bf_bufsize(tcp_conn->tc_rcvd_data))
69083           {
69084                   printf("RCV_NXT-RCV_LO != sizeof tc_rcvd_data\n");
69085                   tcp_print_conn(tcp_conn);
69086                   printf(
69087                   "lo_queue= %lu, hi_queue= %lu, sizeof tc_rcvd_data= %d\n",
69088                           lo_queue, hi_queue, bf_bufsize(tcp_conn->tc_rcvd_data));
69089                   allright= FALSE;
69090           }
69091
69092           /* checking send data */
69093           lo_queue= tcp_conn->tc_SND_UNA;
69094           if (lo_queue == tcp_conn->tc_ISS)
69095                   lo_queue++;
69096           if (lo_queue == tcp_conn->tc_SND_NXT && (tcp_conn->tc_flags &
69097                   TCF_FIN_SENT))
69098                   lo_queue--;
69099           hi_queue= tcp_conn->tc_SND_NXT;
69100           if (hi_queue == tcp_conn->tc_ISS)
69101                   hi_queue++;
69102           if (tcp_conn->tc_flags & TCF_FIN_SENT)
69103                   hi_queue--;
69104
```

```
69105           size= hi_queue-lo_queue;
69106           if (size<0)
69107           {
69108                   printf("SND_NXT-SND_UNA < 0\n");
69109                   allright= FALSE;
69110           }
69111           else if (!tcp_conn->tc_send_data)
69112           {
69113                   if (size)
69114                   {
69115                           printf("SND_NXT-SND_UNA != 0\n");
69116                           printf("SND_NXT= %d, SND_UNA= %d\n",
69117                                   tcp_conn->tc_SND_NXT, tcp_conn->tc_SND_UNA);
69118                           printf("lo_queue= %d, hi_queue= %d\n",
69119                                   lo_queue, hi_queue);
69120                           allright= FALSE;
69121                   }
69122           }
69123           else if (size != bf_bufsize(tcp_conn->tc_send_data))
69124           {
69125                   printf("SND_NXT-SND_UNA != sizeof tc_send_data\n");
69126                   printf("SND_NXT= %d, SND_UNA= %d\n",
69127                           tcp_conn->tc_SND_NXT, tcp_conn->tc_SND_UNA);
69128                   printf("lo_queue= %d, lo_queue= %d\n",
69129                           lo_queue, hi_queue);
69130                   printf("bf_bufsize(data)= %d\n",
69131                           bf_bufsize(tcp_conn->tc_send_data));
69132
69133                   allright= FALSE;
69134           }
69135
69136           /* checking counters */
69137           if (!tcp_GEmod4G(tcp_conn->tc_SND_UNA, tcp_conn->tc_ISS))
69138           {
69139                   printf("SND_UNA < ISS\n");
69140                   allright= FALSE;
69141           }
69142           if (!tcp_GEmod4G(tcp_conn->tc_SND_NXT, tcp_conn->tc_SND_UNA))
69143           {
69144                   printf("SND_NXT<SND_UNA\n");
69145                   allright= FALSE;
69146           }
69147           if (!tcp_GEmod4G(tcp_conn->tc_SND_TRM, tcp_conn->tc_SND_UNA))
69148           {
69149                   printf("SND_TRM<SND_UNA\n");
69150                   allright= FALSE;
69151           }
69152           if (!tcp_GEmod4G(tcp_conn->tc_SND_NXT, tcp_conn->tc_SND_TRM))
69153           {
69154                   printf("SND_NXT<SND_TRM\n");
69155                   allright= FALSE;
69156           }
69157
69158   #if DEBUG
69159    if (!allright)
69160    { printf("tcp_check_conn: not allright\n"); }
69161   #endif
69162           return allright;
69163   }
```

```
69165   PUBLIC void tcp_print_pack(ip_hdr, tcp_hdr)
69166   ip_hdr_t *ip_hdr;
69167   tcp_hdr_t *tcp_hdr;
69168   {
69169           int tcp_hdr_len;
69170
69171   assert(tcp_hdr);
69172           if (ip_hdr)
69173                   writeIpAddr(ip_hdr->ih_src);
69174           else
69175                   printf("???");
69176           printf(",%u ", ntohs(tcp_hdr->th_srcport));
69177           if (ip_hdr)
69178                   writeIpAddr(ip_hdr->ih_dst);
69179           else
69180                   printf("???");
69181           printf(",%u ", ntohs(tcp_hdr->th_dstport));
69182           printf(" 0x%lx", ntohl(tcp_hdr->th_seq_nr));
69183           if (tcp_hdr->th_flags & THF_FIN)
69184                   printf(" <FIN>");
69185           if (tcp_hdr->th_flags & THF_SYN)
69186                   printf(" <SYN>");
69187           if (tcp_hdr->th_flags & THF_RST)
69188                   printf(" <RST>");
69189           if (tcp_hdr->th_flags & THF_PSH)
69190                   printf(" <PSH>");
69191           if (tcp_hdr->th_flags & THF_ACK)
69192                   printf(" <ACK 0x%x %u>", ntohl(tcp_hdr->th_ack_nr),
69193                           ntohs(tcp_hdr->th_window));
69194           if (tcp_hdr->th_flags & THF_URG)
69195                   printf(" <URG %u>", tcp_hdr->th_urgptr);
69196           tcp_hdr_len= (tcp_hdr->th_data_off & TH_DO_MASK) >> 2;
69197           if (tcp_hdr_len != TCP_MIN_HDR_SIZE)
69198                   printf(" <options %d>", tcp_hdr_len-TCP_MIN_HDR_SIZE);
69199   }

69201   PUBLIC void tcp_print_conn(tcp_conn)
69202   tcp_conn_t *tcp_conn;
69203   {
69204           int iss, irs;
69205
69206           iss= tcp_conn->tc_ISS;
69207           irs= tcp_conn->tc_IRS;
69208
69209           tcp_write_state (tcp_conn);
69210           printf(" ISS 0x%lx UNA +0x%lx TRM +0x%lx NXT +0x%lx",
69211                   iss, tcp_conn->tc_SND_UNA-iss, tcp_conn->tc_SND_TRM-iss,
69212                   tcp_conn->tc_SND_NXT-iss);
69213           printf(" IRS 0x%lx LO +0x%x NXT +0x%x HI +0x%x",
69214                   irs, tcp_conn->tc_RCV_LO-irs, tcp_conn->tc_RCV_NXT-irs,
69215                   tcp_conn->tc_RCV_HI-irs);
69216           if (tcp_conn->tc_flags & TCF_INUSE)
69217                   printf(" TCF_INUSE");
69218           if (tcp_conn->tc_flags & TCF_FIN_RECV)
69219                   printf(" TCF_FIN_RECV");
69220           if (tcp_conn->tc_flags & TCF_RCV_PUSH)
69221                   printf(" TCF_RCV_PUSH");
69222           if (tcp_conn->tc_flags & TCF_MORE2WRITE)
69223                   printf(" TCF_MORE2WRITE");
69224           if (tcp_conn->tc_flags & TCF_SEND_ACK)
```

```
69225                   printf(" TCF_SEND_ACK");
69226           if (tcp_conn->tc_flags & TCF_FIN_SENT)
69227                   printf(" TCF_FIN_SENT");
69228           if (tcp_conn->tc_flags & TCF_ACK_TIMER_SET)
69229                   printf(" TCF_ACK_TIMER_SET");
69230   }
```

++
src/inet/generic/tcp_recv.c
++

```
69300   /*
69301   tcp_recv.c
69302   */
69303
69304   #include "inet.h"
69305   #include "buf.h"
69306   #include "clock.h"
69307   #include "type.h"
69308
69309   #include "tcp_int.h"
69310   #include "tcp.h"
69311   #include "assert.h"
69312
69313   INIT_PANIC();
69314
69315   FORWARD void create_RST ARGS(( tcp_conn_t *tcp_conn,
69316           ip_hdr_t *ip_hdr, tcp_hdr_t *tcp_hdr ));
69317   FORWARD void process_data ARGS(( tcp_conn_t *tcp_conn,
69318           tcp_hdr_t *tcp_hdr, int tcp_hdr_len, acc_t *tcp_data,
69319           int data_len ));
69320   FORWARD void process_advanced_data ARGS(( tcp_conn_t *tcp_conn,
69321           tcp_hdr_t *tcp_hdr, int tcp_hdr_len, acc_t *tcp_data,
69322           int data_len ));
69323   FORWARD acc_t *merge_packs ARGS(( acc_t *first, acc_t *next ));
69324   FORWARD void fd_read ARGS(( tcp_fd_t *tcp_fd ));
69325   FORWARD void switch_read_fd ARGS(( tcp_conn_t *tcp_conn,
69326           tcp_fd_t *tcp_fd, tcp_fd_t **ref_urg_fd,
69327           tcp_fd_t **ref_norm_fd ));
69328
69329   PUBLIC void tcp_frag2conn(tcp_conn, ip_pack, tcp_pack)
69330   tcp_conn_t *tcp_conn;
69331   acc_t *ip_pack;
69332   acc_t *tcp_pack;
69333   {
69334           tcp_fd_t *connuser;
69335           int tcp_hdr_flags;
69336           ip_hdr_t *ip_hdr;
69337           tcp_hdr_t *tcp_hdr;
69338           int ip_hdr_len, tcp_hdr_len;
69339           u32_t seg_ack, seg_seq, rcv_hi;
69340           u16_t data_length, seg_wnd;
69341           int acceptable_ACK, segm_acceptable;
69342
69343   #if DEBUG & 256
69344    { where(); printf("tcp_frag2conn(&tcp_conn_table[%d],..) called\n",
```

```
69345           tcp_conn-tcp_conn_table); }
69346   #endif
69347   #if DEBUG & 256
69348    { where(); printf("tc_connuser= 0x%x\n", tcp_conn->tc_connuser); }
69349   #endif
69350
69351           ip_pack= bf_packIffLess(ip_pack, IP_MIN_HDR_SIZE);
69352           ip_hdr= (ip_hdr_t *)ptr2acc_data(ip_pack);
69353           ip_hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
69354           if (ip_hdr_len>IP_MIN_HDR_SIZE)
69355           {
69356                   ip_pack= bf_packIffLess(ip_pack, ip_hdr_len);
69357                   ip_hdr= (ip_hdr_t *)ptr2acc_data(ip_pack);
69358           }
69359
69360           tcp_pack= bf_packIffLess(tcp_pack, TCP_MIN_HDR_SIZE);
69361           tcp_hdr= (tcp_hdr_t *)ptr2acc_data(tcp_pack);
69362           tcp_hdr_len= (tcp_hdr->th_data_off & TH_DO_MASK) >> 2;
69363                   /* actualy (>> 4) << 2 */
69364           if (tcp_hdr_len>TCP_MIN_HDR_SIZE)
69365           {
69366                   tcp_pack= bf_packIffLess(tcp_pack, tcp_hdr_len);
69367                   tcp_hdr= (tcp_hdr_t *)ptr2acc_data(tcp_pack);
69368           }
69369           data_length= tcp_hdr->th_chksum-tcp_hdr_len;
69370                   /* th_chksum is used for packet size internally */
69371           tcp_hdr_flags= tcp_hdr->th_flags & TH_FLAGS_MASK;
69372           seg_ack= ntohl(tcp_hdr->th_ack_nr);
69373           seg_seq= ntohl(tcp_hdr->th_seq_nr);
69374           seg_wnd= ntohs(tcp_hdr->th_window);
69375
69376           switch (tcp_conn->tc_state)
69377           {
69378           case TCS_CLOSED:
69379   /*
69380   CLOSED:
69381           discard all data.
69382           !RST ?
69383                   ACK ?
69384                           <SEQ=SEG.ACK><CTL=RST>
69385                           exit
69386                   :
69387                           <SEQ=0><ACK=SEG.SEQ+SEG.LEN><CTL=RST,ACK>
69388                           exit
69389           :
69390                   discard packet
69391                   exit
69392   */
69393
69394                   if (!(tcp_hdr_flags & THF_RST))
69395                   {
69396                           create_RST (tcp_conn, ip_hdr, tcp_hdr);
69397                           tcp_restart_write(tcp_conn);
69398                   }
69399                   break;
69400           case TCS_LISTEN:
69401   /*
69402   LISTEN:
69403           RST ?
69404                   discard packet
```

```
69405                   exit
69406           ACK ?
69407                   <SEQ=SEG.ACK><CTL=RST>
69408                   exit
69409           SYN ?
69410                   BUG: no security check
69411                   RCV.NXT= SEG.SEQ+1
69412                   IRS= SEG.SEQ
69413                   ISS should already be selected
69414                   <SEQ=ISS><ACK=RCV.NXT><CTL=SYN,ACK>
69415                   SND.NXT=ISS+1
69416                   SND.UNA=ISS
69417                   state= SYN-RECEIVED
69418                   exit
69419           :
69420                   shouldnot occur
69421                   discard packet
69422                   exit
69423   */
69424   #if DEBUG & 256
69425    { where(); printf("\n"); }
69426   #endif
69427                   if (tcp_hdr_flags & THF_RST)
69428                           break;
69429                   if (tcp_hdr_flags & THF_ACK)
69430                   {
69431                           create_RST (tcp_conn, ip_hdr, tcp_hdr);
69432                           tcp_restart_write(tcp_conn);
69433                           break;
69434                   }
69435                   if (tcp_hdr_flags & THF_SYN)
69436                   {
69437                           tcp_extract_ipopt(tcp_conn, ip_hdr);
69438                           tcp_extract_tcpopt(tcp_conn, tcp_hdr);
69439                           tcp_conn->tc_RCV_LO= seg_seq+1;
69440                           tcp_conn->tc_RCV_NXT= seg_seq+1;
69441                           tcp_conn->tc_RCV_HI= tcp_conn->tc_RCV_LO+
69442                                   tcp_conn->tc_rcv_wnd;
69443                           tcp_conn->tc_RCV_UP= seg_seq;
69444                           tcp_conn->tc_IRS= seg_seq;
69445                           tcp_conn->tc_SND_UNA= tcp_conn->tc_ISS;
69446                           tcp_conn->tc_SND_TRM= tcp_conn->tc_ISS;
69447                           tcp_conn->tc_SND_NXT= tcp_conn->tc_ISS+1;
69448                           tcp_conn->tc_SND_UP= tcp_conn->tc_ISS-1;
69449                           tcp_conn->tc_SND_PSH= tcp_conn->tc_ISS-1;
69450                           tcp_conn->tc_SND_WL1= seg_seq;
69451                           tcp_conn->tc_state= TCS_SYN_RECEIVED;
69452                           tcp_conn->tc_no_retrans= 0;
69453   assert (tcp_check_conn(tcp_conn));
69454   #if DEBUG & 2
69455    { where(); tcp_write_state(tcp_conn); }
69456   #endif
69457                           tcp_conn->tc_locaddr= ip_hdr->ih_dst;
69458                           tcp_conn->tc_locport= tcp_hdr->th_dstport;
69459                           tcp_conn->tc_remaddr= ip_hdr->ih_src;
69460                           tcp_conn->tc_remport= tcp_hdr->th_srcport;
69461   #if DEBUG & 256
69462    { where(); printf("calling tcp_restart_write(&tcp_conn_table[%d])\n",
69463           tcp_conn-tcp_conn_table); }
69464   #endif
```

```
69465                           tcp_restart_write(tcp_conn);
69466                           break;
69467                   }
69468   #if DEBUG
69469    { where(); printf("this shouldn't happen\n"); }
69470   #endif
69471                   break;
69472           case TCS_SYN_SENT:
69473   /*
69474   SYN-SENT:
69475           ACK ?
69476                   SEG.ACK <= ISS || SEG.ACK > SND.NXT ?
69477                           RST ?
69478                                   discard packet
69479                                   exit
69480                           :
69481                                   <SEQ=SEG.ACK><CTL=RST>
69482                                   exit
69483                   SND.UNA <= SEG.ACK && SEG.ACK <= SND.NXT ?
69484                           ACK is acceptable
69485                   :
69486                           ACK is !acceptable
69487           :
69488                   ACK is !acceptable
69489           RST ?
69490                   ACK acceptable ?
69491                           discard segment
69492                           state= CLOSED
69493                           error "connection refused"
69494                           exit
69495                   :
69496                           discard packet
69497                           exit
69498           BUG: no security check
69499           SYN ?
69500                   IRS= SEG.SEQ
69501                   RCV.NXT= IRS+1
69502                   ACK ?
69503                           SND.UNA= SEG.ACK
69504                   SND.UNA > ISS ?
69505                           state= ESTABLISHED
69506                           <SEQ=SND.NXT><ACK= RCV.NXT><CTL=ACK>
69507                           process ev. URG and text
69508                           exit
69509                   :
69510                           state= SYN-RECEIVED
69511                           SND.WND= SEG.WND
69512                           SND.WL1= SEG.SEQ
69513                           SND.WL2= SEG.ACK
69514                           <SEQ=ISS><ACK=RCV.NXT><CTL=SYN,ACK>
69515                           exit
69516           :
69517                   discard segment
69518                   exit
69519   */
69520                   if (tcp_hdr_flags & THF_ACK)
69521                   {
69522                           if (tcp_LEmod4G(seg_ack, tcp_conn->tc_ISS) ||
69523                                   tcp_Gmod4G(seg_ack, tcp_conn->tc_SND_NXT))
69524                                   if (tcp_hdr_flags & THF_RST)
```

```
69525                                           break;
69526                                   else
69527                                   {
69528                                           create_RST (tcp_conn, ip_hdr,
69529                                                   tcp_hdr);
69530                                           tcp_restart_write(tcp_conn);
69531                                           break;
69532                                   }
69533                                   acceptable_ACK= (tcp_LEmod4G(tcp_conn->tc_SND_UNA,
69534                                           seg_ack) && tcp_LEmod4G(seg_ack,
69535                                           tcp_conn->tc_SND_NXT));
69536                           }
69537                           else
69538                                   acceptable_ACK= FALSE;
69539                           if (tcp_hdr_flags & THF_RST)
69540                           {
69541                                   if (acceptable_ACK)
69542                                   {
69543   #if DEBUG & 256
69544    { where(); printf("calling tcp_close_connection\n"); }
69545   #endif
69546                                           tcp_close_connection(tcp_conn,
69547                                                   ECONNREFUSED);
69548                                   }
69549                                   break;
69550                           }
69551                           if (tcp_hdr_flags & THF_SYN)
69552                           {
69553                                   tcp_conn->tc_RCV_LO= seg_seq+1;
69554                                   tcp_conn->tc_RCV_NXT= seg_seq+1;
69555                                   tcp_conn->tc_RCV_HI= tcp_conn->tc_RCV_LO +
69556                                           tcp_conn->tc_rcv_wnd;
69557                                   tcp_conn->tc_RCV_UP= seg_seq;
69558                                   tcp_conn->tc_IRS= seg_seq;
69559                                   if (tcp_hdr_flags & THF_ACK)
69560                                           tcp_conn->tc_SND_UNA= seg_ack;
69561                                   if (tcp_Gmod4G(tcp_conn->tc_SND_UNA,
69562                                           tcp_conn->tc_ISS))
69563                                   {
69564                                           tcp_conn->tc_state= TCS_ESTABLISHED;
69565   assert (tcp_check_conn(tcp_conn));
69566   #if DEBUG & 2
69567    { where(); tcp_write_state(tcp_conn); }
69568   #endif
69569   #if DEBUG & 256
69570    { where(); printf("ISS= 0x%lx\n", tcp_conn->tc_ISS); }
69571   #endif
69572   assert(tcp_conn->tc_connuser);
69573                                   tcp_restart_connect(tcp_conn-> tc_connuser);
69574                                   if (tcp_conn->tc_state == TCS_CLOSED)
69575                                   {
69576    { where(); printf("connection closed while inuse\n"); }
69577                                           break;
69578                                   }
69579                                   tcp_conn->tc_flags &= ~TCF_ACK_TIMER_SET;
69580   #if DEBUG & 256
69581    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
69582           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
69583   #endif
69584                                   tcp_conn->tc_flags |= TCF_SEND_ACK;
```

```
69585   #if DEBUG & 256
69586    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
69587           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
69588   #endif
69589                                   tcp_restart_write (tcp_conn);
69590                                   tcp_frag2conn(tcp_conn, ip_pack, tcp_pack);
69591                                   return;
69592                                   /* ip_pack and tcp_pack are already
69593                                           freed */
69594                           }
69595                           tcp_conn->tc_state= TCS_SYN_RECEIVED;
69596   assert (tcp_check_conn(tcp_conn));
69597   #if DEBUG
69598    { where(); tcp_write_state(tcp_conn); }
69599   #endif
69600                           tcp_conn->tc_SND_TRM= tcp_conn->tc_ISS;
69601                           tcp_restart_write(tcp_conn);
69602                   }
69603                   break;
69604           case TCS_SYN_RECEIVED:
69605           case TCS_ESTABLISHED:
69606           case TCS_FIN_WAIT_1:
69607           case TCS_FIN_WAIT_2:
69608           case TCS_CLOSE_WAIT:
69609           case TCS_CLOSING:
69610           case TCS_LAST_ACK:
69611           case TCS_TIME_WAIT:
69612   /*
69613   SYN-RECEIVED:
69614   ESTABLISHED:
69615   FIN-WAIT-1:
69616   FIN-WAIT-2:
69617   CLOSE-WAIT:
69618   CLOSING:
69619   LAST-ACK:
69620   TIME-WAIT:
69621           test if segment is acceptable:
69622           Segment Receive Test
69623           Length  Window
69624           0       0       SEG.SEQ == RCV.NXT
69625           0       >0      RCV.NXT <= SEG.SEQ && SEG.SEQ < RCV.NXT+RCV.WND
69626           >0      0       not acceptable
69627           >0      >0      (RCV.NXT <= SEG.SEQ && SEG.SEQ < RCV.NXT+RCV.WND)
69628                           || (RCV.NXT <= SEG.SEQ+SEG.LEN-1 &&
69629                           SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND)
69630           for urgent data: use RCV.WND+URG.WND for RCV.WND
69631   */
69632   #if DEBUG & 256
69633    { where(); printf("\n"); }
69634   #endif
69635                   rcv_hi= tcp_conn->tc_RCV_HI;
69636                   if (tcp_hdr_flags & THF_URG)
69637                           rcv_hi= tcp_conn->tc_RCV_LO + tcp_conn->tc_rcv_wnd +
69638                                   tcp_conn->tc_urg_wnd;
69639                   if (!data_length)
69640                   {
69641                           if (rcv_hi == tcp_conn->tc_RCV_NXT)
69642                           {
69643   #if DEBUG & 256
69644    { where(); printf("\n"); }
```

```
69645   #endif
69646                                   segm_acceptable= (seg_seq == rcv_hi);
69647   #if DEBUG
69648    if (!segm_acceptable)
69649    { where(); printf("!segm_acceptable\n"); }
69650   #endif
69651                           }
69652                           else
69653                           {
69654   #if DEBUG & 256
69655    { where(); printf("\n"); }
69656   #endif
69657   assert (tcp_Gmod4G(rcv_hi, tcp_conn->tc_RCV_NXT));
69658                                   segm_acceptable= (tcp_LEmod4G(tcp_conn->
69659                                           tc_RCV_NXT, seg_seq) &&
69660                                           tcp_Lmod4G(seg_seq, rcv_hi));
69661   #if DEBUG & 256
69662    if (!segm_acceptable)
69663    { where(); printf("RCV_NXT= 0x%lx, seg_seq= 0x%lx, rcv_hi= 0x%lx\n",
69664           tcp_conn->tc_RCV_NXT, seg_seq, rcv_hi); }
69665   #endif
69666                           }
69667                   }
69668                   else
69669                   {
69670                           if (tcp_Gmod4G(rcv_hi, tcp_conn->tc_RCV_NXT))
69671                           {
69672   #if DEBUG & 256
69673    { where(); printf("RCV_NXT= %lu, rcv_hi= %lu, seg_seq= %lu, data_length= %u\n",
69674           tcp_conn->tc_RCV_NXT, rcv_hi, seg_seq, data_length); }
69675   #endif
69676                                   segm_acceptable= (tcp_LEmod4G(tcp_conn->
69677                                           tc_RCV_NXT, seg_seq) &&
69678                                           tcp_Lmod4G(seg_seq, rcv_hi)) ||
69679                                           (tcp_LEmod4G(tcp_conn->tc_RCV_NXT,
69680                                           seg_seq+data_length-1) &&
69681                                           tcp_Lmod4G(seg_seq+data_length-1,
69682                                           rcv_hi));
69683   #if DEBUG & 256
69684    if (!segm_acceptable)
69685    { where(); printf("!segm_acceptable\n"); }
69686   #endif
69687                           }
69688                           else
69689                           {
69690   #if DEBUG
69691    { where(); printf("\n"); }
69692   #endif
69693                                   segm_acceptable= FALSE;
69694   #if DEBUG
69695    if (!segm_acceptable)
69696    { where(); printf("!segm_acceptable\n"); }
69697   #endif
69698                           }
69699                   }
69700   /*
69701           !segment acceptable ?
69702                   RST ?
69703                           discard packet
69704                           exit
```

```
69705                   :
69706                           <SEG=SND.NXT><ACK=RCV.NXT><CTL=ACK>
69707                           exit
69708   */
69709   #if DEBUG & 256
69710    { where(); printf("\n"); }
69711   #endif
69712                   if (!segm_acceptable)
69713                   {
69714   #if DEBUG & 256
69715    { where(); printf("segment is not acceptable\n"); }
69716   #endif
69717                           if (!(tcp_hdr_flags & THF_RST))
69718                           {
69719   #if DEBUG & 256
69720    { where(); printf("segment is not acceptable setting ack timer\n"); }
69721   #endif
69722                                   tcp_set_ack_timer(tcp_conn);
69723                           }
69724                           break;
69725                   }
69726   /*
69727           RST ?
69728                   state == SYN-SECEIVED ?
69729                           initiated by a LISTEN ?
69730                                   state= LISTEN
69731                                   exit
69732                           :
69733                                   state= CLOSED
69734                                   error "connection refused"
69735                                   exit
69736                   state == CLOSING || state == LAST-ACK ||
69737                           state == TIME-WAIT ?
69738                           state= CLOSED
69739                           exit
69740                   :
69741                           state= CLOSED
69742                           error "connection reset"
69743                           exit
69744   */
69745   #if DEBUG & 256
69746    { where(); printf("\n"); }
69747   #endif
69748                   if (tcp_hdr_flags & THF_RST)
69749                   {
69750                           if (tcp_conn->tc_state == TCS_SYN_RECEIVED)
69751                           {
69752                                   if (tcp_conn->tc_orglisten)
69753                                   {
69754                                           connuser= tcp_conn->tc_connuser;
69755                                           tcp_conn->tc_connuser= 0;
69756   #if DEBUG
69757    { where(); printf("calling tcp_close_connection\n"); }
69758   #endif
69759                                           tcp_close_connection (tcp_conn,
69760                                                   ECONNREFUSED);
69761                                           if (connuser)
69762                                                   (void)tcp_su4listen(
69763                                                           connuser);
69764                                           break;
```

```
69765                                   }
69766                                   else
69767                                   {
69768   #if DEBUG
69769    { where(); printf("calling tcp_close_connection\n"); }
69770   #endif
69771                                           tcp_close_connection(tcp_conn,
69772                                                   ECONNREFUSED);
69773                                           break;
69774                                   }
69775                           }
69776                           if (tcp_conn->tc_state == TCS_CLOSING ||
69777                                   tcp_conn->tc_state == TCS_LAST_ACK ||
69778                                   tcp_conn->tc_state == TCS_TIME_WAIT)
69779                           {
69780   #if DEBUG
69781    { where(); printf("calling tcp_close_connection\n"); }
69782   #endif
69783                                   tcp_close_connection (tcp_conn,
69784                                           ENOCONN);
69785                                   break;
69786                           }
69787   #if DEBUG
69788    { where(); printf("calling tcp_close_connection\n"); }
69789   #endif
69790                           tcp_close_connection(tcp_conn, ECONNRESET);
69791                           break;
69792                   }
69793   /*
69794           SYN in window ?
69795                   state == SYN-RECEIVED && initiated by a LISTEN ?
69796                           state= LISTEN
69797                           exit
69798                   :
69799                           state= CLOSED
69800                           error "connection reset"
69801                           exit
69802   */
69803   #if DEBUG  & 256
69804    { where(); printf("\n"); }
69805   #endif
69806                   if ((tcp_hdr_flags & THF_SYN) && tcp_GEmod4G(seg_seq,
69807                           tcp_conn->tc_RCV_NXT))
69808                   {
69809                           if (tcp_conn->tc_state == TCS_SYN_RECEIVED &&
69810                                   tcp_conn->tc_orglisten)
69811                           {
69812                                   connuser= tcp_conn->tc_connuser;
69813                                   tcp_conn->tc_connuser= 0;
69814   #if DEBUG
69815    { where(); printf("calling tcp_close_connection\n"); }
69816   #endif
69817                                   tcp_close_connection(tcp_conn,
69818                                           ECONNRESET);
69819                                   if (connuser)
69820                                           (void)tcp_su4listen(connuser);
69821                                   break;
69822                           }
69823   #if DEBUG
69824    { where(); printf("calling tcp_close_connection\n"); }
```

```
69825   #endif
69826                           tcp_close_connection(tcp_conn, ECONNRESET);
69827                           break;
69828                   }
69829   /*
69830           !ACK ?
69831                   discard packet
69832                   exit
69833   */
69834   #if DEBUG & 256
69835    { where(); printf("\n"); }
69836   #endif
69837                   if (!(tcp_hdr_flags & THF_ACK))
69838                           break;
69839   /*
69840           state == SYN-RECEIVED ?
69841                   SND.UNA <= SEG.ACK <= SND.NXT ?
69842                           state= ESTABLISHED
69843                   :
69844                           <SEG=SEG.ACK><CTL=RST>
69845                           exit
69846   */
69847   #if DEBUG & 256
69848    { where(); printf("\n"); }
69849   #endif
69850           if (tcp_conn->tc_state == TCS_SYN_RECEIVED)
69851           {
69852                   if (tcp_LEmod4G(tcp_conn->tc_SND_UNA, seg_ack) &&
69853                           tcp_LEmod4G(seg_ack, tcp_conn->tc_SND_NXT))
69854                   {
69855                           tcp_conn->tc_state= TCS_ESTABLISHED;
69856   assert (tcp_check_conn(tcp_conn));
69857   #if DEBUG & 2
69858    { where(); tcp_write_state(tcp_conn); }
69859   #endif
69860   assert(tcp_conn->tc_connuser);
69861                           tcp_restart_connect(tcp_conn->tc_connuser);
69862                           if (tcp_conn->tc_state == TCS_CLOSED)
69863                           {
69864   #if DEBUG
69865    { where(); printf("connection closed while inuse\n"); }
69866   #endif
69867                                   break;
69868                           }
69869                   }
69870                   else
69871                   {
69872                           create_RST (tcp_conn, ip_hdr, tcp_hdr);
69873                           tcp_restart_write(tcp_conn);
69874                           break;
69875                   }
69876           }
69877   /*
69878           state == ESTABLISHED || state == FIN-WAIT-1 ||
69879                   state == FIN-WAIT-2 || state == CLOSE-WAIT ||
69880                   state == LAST-ACK || state == TIME_WAIT || state == CLOSING ?
69881                   SND.UNA < SEG.ACK <= SND.NXT ?
69882                           SND.UNA= SEG.ACK
69883                           reply "send ok"
69884                           SND.WL1 < SEG.SEQ || (SND.WL1 == SEG.SEQ &&
```

```
69885                                   SND.WL2 <= SEG.ACK ?
69886                                   SND.WND= SEG.WND
69887                                   SND.Wl1= SEG.SEQ
69888                                   SND.WL2= SEG.ACK
69889                   SEG.ACK <= SND.UNA ?
69890                           ignore ACK
69891                   SEG.ACK > SND.NXT ?
69892                           <SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>
69893                           discard packet
69894                           exit
69895   */
69896   #if DEBUG & 256
69897    { where(); printf("\n"); }
69898   #endif
69899                   if (tcp_conn->tc_state == TCS_ESTABLISHED ||
69900                           tcp_conn->tc_state == TCS_FIN_WAIT_1 ||
69901                           tcp_conn->tc_state == TCS_FIN_WAIT_2 ||
69902                           tcp_conn->tc_state == TCS_CLOSE_WAIT ||
69903                           tcp_conn->tc_state == TCS_LAST_ACK ||
69904                           tcp_conn->tc_state == TCS_TIME_WAIT ||
69905                           tcp_conn->tc_state == TCS_CLOSING)
69906                   {
69907                           if (tcp_LEmod4G(tcp_conn->tc_SND_UNA, seg_ack)
69908                                   && tcp_LEmod4G(seg_ack, tcp_conn->
69909                                   tc_SND_NXT))
69910                           {
69911                                   if (tcp_Lmod4G(tcp_conn->tc_SND_WL1,
69912                                           seg_seq) || (tcp_conn->
69913                                           tc_SND_WL1==seg_seq &&
69914                                           tcp_LEmod4G(tcp_conn->
69915                                           tc_SND_WL2, seg_ack)))
69916                                   {
69917                                           if (seg_wnd > TCP_MAX_WND_SIZE)
69918                                                   seg_wnd= TCP_MAX_WND_SIZE;
69919                                           if (!seg_wnd)
69920                                                   seg_wnd++;
69921                                           tcp_conn->tc_SND_WL1= seg_seq;
69922                                           tcp_conn->tc_SND_WL2= seg_ack;
69923   #if SUN_0WND_BUG
69924                                           if (seg_wnd && seg_ack == tcp_conn->
69925                                                   tc_SND_UNA && tcp_LEmod4G(
69926                                                   seg_ack + seg_wnd,
69927                                                   tcp_conn->tc_SND_NXT) &&
69928                                                   tcp_LEmod4G(seg_ack + seg_wnd,
69929                                                   tcp_conn->tc_snd_cwnd))
69930                                                   seg_wnd= 0;
69931   #endif
69932                                   }
69933                                   else
69934                                   {
69935                                           seg_wnd= tcp_conn->tc_mss;
69936                                           /* assume 1 segment if not a valid
69937                                            * window */
69938                                   }
69939                                   tcp_release_retrans(tcp_conn, seg_ack, seg_wnd);
69940                                   if (tcp_conn->tc_state == TCS_CLOSED)
69941                                   {
69942   #if DEBUG
69943    { where(); printf("connection closed while inuse\n"); }
69944   #endif
```

```
69945                                           break;
69946                                   }
69947                           }
69948                           else if (tcp_Gmod4G(seg_ack,
69949                                   tcp_conn->tc_SND_NXT))
69950                           {
69951                                   tcp_set_ack_timer(tcp_conn);
69952   #if DEBUG
69953    { where(); printf("got an ack of something I haven't send\n");
69954      printf("seg_ack= %lu, SND_NXT= %lu\n", seg_ack, tcp_conn->tc_SND_NXT); }
69955   #endif
69956                                   break;
69957                           }
69958   #if DEBUG & 256
69959    if (!seg_wnd) { where(); printf("SND_UNA= %lu, SND_NXT= %lu\n",
69960           tcp_conn->tc_SND_UNA, tcp_conn->tc_SND_NXT); }
69961   #endif
69962                           if (!seg_wnd &&
69963                           /* tcp_GEmod4G(seg_wnd, tcp_conn->tc_SND_UNA) &&
69964                           */
69965                                   tcp_Lmod4G(tcp_conn->tc_SND_UNA,
69966                                   tcp_conn->tc_SND_NXT))
69967                           {       /* zero window */
69968   #if DEBUG & 256
69969    { where(); printf("setting 0wnd_to\n"); }
69970   #endif
69971                                   clck_untimer(&tcp_conn->tc_minor_timer);
69972                                   if (!tcp_conn->tc_0wnd_to)
69973                                   {
69974   assert (tcp_conn->tc_rtt);
69975                                           tcp_conn->tc_0wnd_to=
69976                                                   tcp_conn->tc_rtt;
69977                                   }
69978                                   clck_timer(&tcp_conn->tc_major_timer,
69979                                           get_time()+tcp_conn->tc_0wnd_to,
69980                                           tcp_zero_wnd_to, tcp_conn-
69981                                           tcp_conn_table);
69982                           }
69983                           else
69984                           {
69985   #if DEBUG & 256
69986    { where(); printf("not setting 0wnd_to\n"); }
69987   #endif
69988                                   if (tcp_conn->tc_0wnd_to)
69989                                   {
69990   #if DEBUG & 256
69991    { where(); printf("resetting 0wnd_to\n"); }
69992   #endif
69993                                           tcp_conn->tc_0wnd_to= 0;
69994                                           tcp_conn->tc_SND_TRM=
69995                                                   tcp_conn->tc_SND_UNA;
69996                                           clck_untimer(&tcp_conn->
69997                                                   tc_major_timer);
69998                                           tcp_restart_write (tcp_conn);
69999                                   }
70000                           }
70001                   }
70002   /*
70003           state == FIN-WAIT-1 && FIN acknowledged ?
70004                   state= FIN-WAIT-2
```

```
70005
70006           state == CLOSING && FIN acknowledged ?
70007                   state= TIME-WAIT
70008
70009           state == LAST-ACK && FIN acknowledged ?
70010                   state= CLOSED
70011
70012           state == TIME-WAIT ?
70013                   <SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>
70014                   restart 2 MSL timeout
70015   */
70016   #if DEBUG & 256
70017    { where(); printf("\n"); }
70018   #endif
70019                   if (tcp_conn->tc_SND_UNA == tcp_conn->tc_SND_NXT)
70020                   {
70021                           switch (tcp_conn->tc_state)
70022                           {
70023                           case TCS_FIN_WAIT_1:
70024
70025                                   tcp_conn->tc_state= TCS_FIN_WAIT_2;
70026   assert (tcp_check_conn(tcp_conn));
70027   #if DEBUG
70028    { where(); tcp_write_state(tcp_conn); }
70029   #endif
70030                                   break;
70031                           case TCS_CLOSING:
70032                                   tcp_conn->tc_state= TCS_TIME_WAIT;
70033   assert (tcp_check_conn(tcp_conn));
70034   #if DEBUG
70035    { where(); tcp_write_state(tcp_conn); }
70036   #endif
70037                                   tcp_set_time_wait_timer(tcp_conn);
70038                                   break;
70039                           case TCS_LAST_ACK:
70040   #if DEBUG & 256
70041    { where(); printf("calling tcp_close_connection\n"); }
70042   #endif
70043                                   tcp_close_connection(tcp_conn, ENOCONN);
70044                                   break;
70045                           }
70046                           if (!tcp_conn->tc_mainuser)
70047                           {
70048                                   tcp_close_connection(tcp_conn, ENOCONN);
70049                                   break;
70050                           }
70051                   }
70052                   if (tcp_conn->tc_state == TCS_TIME_WAIT)
70053                   {
70054                           tcp_set_ack_timer(tcp_conn);
70055                           tcp_set_time_wait_timer(tcp_conn);
70056                   }
70057
70058   /*
70059           process data...
70060   */
70061   #if DEBUG & 256
70062    { where(); printf("\n"); }
70063   #endif
70064                   tcp_extract_ipopt(tcp_conn, ip_hdr);
```

```
70065                   tcp_extract_tcpopt(tcp_conn, tcp_hdr);
70066
70067                   if (data_length)
70068                   {
70069                           if (tcp_LEmod4G(seg_seq, tcp_conn->tc_RCV_NXT))
70070                                   process_data (tcp_conn, tcp_hdr,
70071                                           tcp_hdr_len, tcp_pack,
70072                                           data_length);
70073                           else
70074                                   process_advanced_data (tcp_conn,
70075                                           tcp_hdr, tcp_hdr_len, tcp_pack,
70076                                           data_length);
70077                           if (tcp_conn->tc_state == TCS_CLOSED)
70078                                   break;
70079                           tcp_conn->tc_flags |= TCF_SEND_ACK;
70080   #if DEBUG & 256
70081    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
70082           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
70083   #endif
70084                           tcp_restart_write (tcp_conn);
70085                           if (tcp_conn->tc_state == TCS_CLOSED)
70086                                   break;
70087                   }
70088   /*
70089           FIN ?
70090                   reply pending receives
70091                   advace RCV.NXT over the FIN
70092                   <SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>
70093
70094                   state == SYN-RECEIVED || state == ESTABLISHED ?
70095                           state= CLOSE-WAIT
70096                   state == FIN-WAIT-1 ?
70097                           state= CLOSING
70098                   state == FIN-WAIT-2 ?
70099                           state= TIME-WAIT
70100                   state == TIME-WAIT ?
70101                           restart the TIME-WAIT timer
70102           exit
70103   */
70104   #if DEBUG & 256
70105    { where(); printf("\n"); }
70106   #endif
70107                   if ((tcp_hdr_flags & THF_FIN) && tcp_LEmod4G(seg_seq,
70108                           tcp_conn->tc_RCV_NXT))
70109                   {
70110                           switch (tcp_conn->tc_state)
70111                           {
70112                           case TCS_SYN_RECEIVED:
70113                                   break;
70114                           case TCS_ESTABLISHED:
70115                                   tcp_conn->tc_state= TCS_CLOSE_WAIT;
70116   assert (tcp_check_conn(tcp_conn));
70117   #if DEBUG
70118    { where(); tcp_write_state(tcp_conn); }
70119   #endif
70120                                   break;
70121                           case TCS_FIN_WAIT_1:
70122                                   tcp_conn->tc_state= TCS_CLOSING;
70123   assert (tcp_check_conn(tcp_conn));
70124   #if DEBUG
```

```
70125     { where(); tcp_write_state(tcp_conn); }
70126     #endif
70127                                     break;
70128                             case TCS_FIN_WAIT_2:
70129                                     tcp_conn->tc_state= TCS_TIME_WAIT;
70130     assert (tcp_check_conn(tcp_conn));
70131     #if DEBUG
70132     { where(); tcp_write_state(tcp_conn); }
70133     #endif
70134                                     /* drops through */
70135                             case TCS_TIME_WAIT:
70136                                     tcp_set_time_wait_timer(tcp_conn);
70137                                     break;
70138                             }
70139                             if (!(tcp_conn->tc_flags & TCF_FIN_RECV))
70140                             {
70141                                     tcp_conn->tc_RCV_NXT++;
70142                                     tcp_conn->tc_flags |= TCF_FIN_RECV;
70143     #if DEBUG & 256
70144     { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
70145             tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
70146     #endif
70147                             }
70148                             tcp_set_ack_timer(tcp_conn);
70149                             if (tcp_conn->tc_readuser)
70150                                     tcp_restart_fd_read(tcp_conn);
70151                     }
70152                     break;
70153             default:
70154                     printf("unknown state: tcp_conn->tc_state== %d\n",
70155                             tcp_conn->tc_state);
70156                     break;
70157             }
70158             bf_afree(ip_pack);
70159             bf_afree(tcp_pack);
70160     }


70163     PRIVATE void process_data(tcp_conn, tcp_hdr, tcp_hdr_len, tcp_data,
70164             data_len)
70165     tcp_conn_t *tcp_conn;
70166     tcp_hdr_t *tcp_hdr;
70167     int tcp_hdr_len;
70168     acc_t *tcp_data;
70169     int data_len;
70170     {
70171             u32_t lo_seq, hi_seq, urg_seq, seq_nr;
70172             u16_t urgptr;
70173             int tcp_hdr_flags;
70174             unsigned int offset;
70175             acc_t *all_data, *tmp_data, *rcvd_data;
70176
70177     #if DEBUG & 256
70178     { where(); printf("in process data\n"); }
70179     #endif
70180             seq_nr= ntohl(tcp_hdr->th_seq_nr);
70181             urgptr= ntohs(tcp_hdr->th_urgptr);
70182             while (tcp_data)
70183             {
70184     assert (tcp_check_conn(tcp_conn));
```

```
70185                   all_data= bf_cut(tcp_data, tcp_hdr_len, data_len);
70186                   tcp_data= 0;
70187
70188                   lo_seq= seq_nr;
70189                   tcp_hdr_flags= tcp_hdr->th_flags & TH_FLAGS_MASK;
70190
70191                   if (tcp_hdr_flags & THF_URG)
70192                   {
70193                           urg_seq= lo_seq+ urgptr;
70194                           tcp_conn->tc_RCV_HI= tcp_conn->tc_RCV_LO+
70195                                   tcp_conn->tc_rcv_wnd+tcp_conn->
70196                                   tc_urg_wnd;
70197                           if (tcp_GEmod4G(urg_seq, tcp_conn->tc_RCV_HI))
70198                                   urg_seq= tcp_conn->tc_RCV_HI;
70199                           if (tcp_Gmod4G(urg_seq, tcp_conn->tc_RCV_UP))
70200                                   tcp_conn->tc_RCV_UP= urg_seq;
70201                   }
70202                   if (tcp_hdr_flags & THF_SYN)
70203                           lo_seq++;
70204
70205                   if (tcp_hdr_flags & THF_PSH)
70206                   {
70207                           tcp_conn->tc_flags |= TCF_RCV_PUSH;
70208   #if DEBUG & 256
70209    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
70210           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
70211   #endif
70212                   }
70213
70214                   if (tcp_Lmod4G(lo_seq, tcp_conn->tc_RCV_NXT))
70215                   {
70216                           offset= tcp_conn->tc_RCV_NXT-lo_seq;
70217                           tmp_data= bf_cut(all_data, offset, data_len-
70218                                   offset);
70219                           bf_afree(all_data);
70220                           lo_seq += offset;
70221                           data_len -= offset;
70222                           all_data= tmp_data;
70223                           tmp_data= 0;
70224                   }
70225                   assert (lo_seq == tcp_conn->tc_RCV_NXT);
70226
70227                   hi_seq= lo_seq+data_len;
70228                   if (tcp_Gmod4G(hi_seq, tcp_conn->tc_RCV_HI))
70229                   {
70230                           data_len= tcp_conn->tc_RCV_HI-lo_seq;
70231                           tmp_data= bf_cut(all_data, 0, data_len);
70232                           bf_afree(all_data);
70233                           all_data= tmp_data;
70234                           hi_seq= lo_seq+data_len;
70235                           tcp_hdr_flags &= ~THF_FIN;
70236                   }
70237                   assert (tcp_LEmod4G (hi_seq, tcp_conn->tc_RCV_HI));
70238
70239   #if DEBUG & 256
70240    { where(); printf("in process data: lo_seq= %lu, hi_seq= %lu\n",
70241           lo_seq, hi_seq); }
70242   #endif
70243                   rcvd_data= tcp_conn->tc_rcvd_data;
70244                   tcp_conn->tc_rcvd_data= 0;
```

```
70245                   tmp_data= bf_append(rcvd_data, all_data);
70246                   if (tcp_conn->tc_state == TCS_CLOSED)
70247                   {
70248   #if DEBUG
70249    { where(); printf("connection closed while inuse\n"); }
70250   #endif
70251                           bf_afree(tmp_data);
70252                           return;
70253                   }
70254                   tcp_conn->tc_rcvd_data= tmp_data;
70255                   tcp_conn->tc_RCV_NXT= hi_seq;
70256
70257                   assert (tcp_conn->tc_RCV_LO + bf_bufsize(tcp_conn->
70258                           tc_rcvd_data) == tcp_conn->tc_RCV_NXT);
70259
70260                   if (tcp_hdr_flags & THF_FIN)
70261                   {
70262   #if DEBUG
70263    { where(); printf("got a FIN\n"); }
70264   #endif
70265                           tcp_conn->tc_RCV_NXT++;
70266                           tcp_conn->tc_flags |= TCF_FIN_RECV;
70267   #if DEBUG & 16
70268    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
70269           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
70270   #endif
70271                   }
70272                   tcp_set_ack_timer(tcp_conn);
70273                   while (tcp_conn->tc_rcv_queue)
70274                   {
70275                           tmp_data= tcp_conn->tc_rcv_queue;
70276                           assert (tmp_data->acc_length >= TCP_MIN_HDR_SIZE);
70277                           tcp_hdr= (tcp_hdr_t *)ptr2acc_data(tmp_data);
70278                           lo_seq= tcp_hdr->th_seq_nr;
70279                           /* th_seq_nr is changed to host byte order */
70280                           if (tcp_Gmod4G(lo_seq, tcp_conn->tc_RCV_NXT))
70281                                   break;
70282                           tcp_hdr_len= (tcp_hdr->th_data_off &
70283                                   TH_DO_MASK) >> 2;
70284                           data_len= tcp_hdr->th_chksum-tcp_hdr_len;
70285                           if (tcp_LEmod4G(lo_seq+data_len, tcp_conn->
70286                                   tc_RCV_NXT))
70287                           {
70288                                   tcp_conn->tc_rcv_queue= tmp_data->
70289                                           acc_ext_link;
70290                                   bf_afree(tmp_data);
70291                                   continue;
70292                           }
70293                           tcp_data= tmp_data;
70294                           seq_nr= tcp_hdr->th_seq_nr;
70295                           urgptr= tcp_hdr->th_urgptr;
70296                           break;
70297                   }
70298           }
70299   assert (tcp_check_conn(tcp_conn));
70300           if (tcp_conn->tc_readuser)
70301                   tcp_restart_fd_read(tcp_conn);
70302           else if (!tcp_conn->tc_mainuser)
70303           {
70304   #if DEBUG
```

```
70305     { where(); printf("calling tcp_close_connection\n"); }
70306     #endif
70307                     tcp_close_connection (tcp_conn, ENOCONN);
70308             }
70309     assert (tcp_check_conn(tcp_conn));
70310     }

70312     PRIVATE void process_advanced_data(tcp_conn, tcp_hdr, tcp_hdr_len,
70313             tcp_data, data_len)
70314     tcp_conn_t *tcp_conn;
70315     tcp_hdr_t *tcp_hdr;
70316     int tcp_hdr_len;
70317     acc_t *tcp_data;
70318     int data_len;
70319     {
70320             u32_t seg_seq, seg_hi;
70321             u16_t seg_up;
70322             acc_t *hdr_acc, *next_acc, **tail_acc_ptr, *head_acc, *data_acc,
70323                     *tmp_acc;
70324             tcp_hdr_t *tmp_hdr;
70325             int tmp_hdr_len;
70326
70327     #if DEBUG & 256
70328     { where(); printf ("processing advanced data\n"); }
70329     #endif
70330             hdr_acc= bf_memreq(tcp_hdr_len);        /* make fresh header */
70331             if (tcp_conn->tc_state == TCS_CLOSED)
70332             {
70333     #if DEBUG
70334     { where(); printf("connection closed while inuse\n"); }
70335     #endif
70336                     bf_afree(hdr_acc);
70337                     return;
70338             }
70339             assert (hdr_acc->acc_length == tcp_hdr_len);
70340             tmp_hdr= (tcp_hdr_t *)ptr2acc_data(hdr_acc);
70341     #if DEBUG & 256
70342     { where(); printf("doing memcpy\n"); }
70343     #endif
70344             memcpy ((char *)tmp_hdr, (char *)tcp_hdr, tcp_hdr_len);
70345             tcp_hdr= tmp_hdr;
70346
70347             data_acc= bf_cut (tcp_data, tcp_hdr_len, data_len);
70348
70349             seg_seq= ntohl(tcp_hdr->th_seq_nr);
70350             tcp_hdr->th_seq_nr= seg_seq;    /* seq_nr in host format */
70351
70352             if (tcp_hdr->th_flags & THF_URG)
70353             {
70354                     seg_up= ntohs(tcp_hdr->th_urgptr);
70355                     tcp_hdr->th_urgptr= seg_up;
70356                                             /* urgptr in host format */
70357                     tcp_conn->tc_RCV_HI= tcp_conn->tc_RCV_LO+
70358                             tcp_conn->tc_rcv_wnd+tcp_conn->tc_urg_wnd;
70359             }
70360
70361             assert (!(tcp_hdr->th_flags & THF_SYN));
70362             assert (tcp_Gmod4G(seg_seq, tcp_conn->tc_RCV_NXT));
70363
70364             tcp_hdr->th_flags &= ~THF_FIN;  /* it is too difficult to
```

```
70365                   preserve a FIN */
70366
70367           seg_hi= seg_seq + data_len;
70368
70369           if (tcp_Gmod4G(seg_hi, tcp_conn->tc_RCV_HI))
70370           {
70371                   seg_hi= tcp_conn->tc_RCV_HI;
70372                   data_len= seg_hi-seg_seq;
70373                   if (!data_len)
70374                   {
70375                           bf_afree(hdr_acc);
70376                           bf_afree(data_acc);
70377                           return;
70378                   }
70379   #if DEBUG
70380    { where(); printf("Cutting packet\n"); }
70381   #endif
70382                   tmp_acc= bf_cut(data_acc, 0, data_len);
70383                   bf_afree(data_acc);
70384                   data_acc= tmp_acc;
70385           }
70386           hdr_acc->acc_next= data_acc;
70387           hdr_acc->acc_ext_link= 0;
70388
70389           head_acc= tcp_conn->tc_rcv_queue;
70390           tcp_conn->tc_rcv_queue= 0;
70391           tail_acc_ptr= 0;
70392           next_acc= head_acc;
70393
70394           while (next_acc)
70395           {
70396                   assert (next_acc->acc_length >= TCP_MIN_HDR_SIZE);
70397                   tmp_hdr= (tcp_hdr_t *)ptr2acc_data(next_acc);
70398                   if (tcp_Lmod4G(seg_seq, tmp_hdr->th_seq_nr))
70399                   {
70400   #if DEBUG & 256
70401    { where(); printf("calling merge_packs\n"); }
70402   #endif
70403                           next_acc= merge_packs(hdr_acc, next_acc);
70404                           hdr_acc= 0;
70405                           if (tail_acc_ptr)
70406                           {
70407   assert (*tail_acc_ptr);
70408                                   (*tail_acc_ptr)->acc_ext_link= 0;
70409   #if DEBUG & 256
70410    { where(); printf("calling merge_packs\n"); }
70411   #endif
70412                                   *tail_acc_ptr= merge_packs(
70413                                           *tail_acc_ptr, next_acc);
70414                           }
70415                           else
70416                                   head_acc= next_acc;
70417                           break;
70418                   }
70419                   if (!tail_acc_ptr)
70420                           tail_acc_ptr=  &head_acc;
70421                   else
70422                           tail_acc_ptr= &(*tail_acc_ptr)->acc_ext_link;
70423                   next_acc= next_acc->acc_ext_link;
70424           }
```

```
70425           if (hdr_acc)
70426           {
70427                   next_acc= hdr_acc;
70428                   hdr_acc= 0;
70429                   if (tail_acc_ptr)
70430                   {
70431                           if (*tail_acc_ptr)
70432                           {
70433                                   (*tail_acc_ptr)->acc_ext_link= 0;
70434   #if DEBUG & 256
70435    { where(); printf("calling merge_packs\n"); }
70436   #endif
70437                                   *tail_acc_ptr= merge_packs(
70438                                           *tail_acc_ptr, next_acc);
70439                           }
70440                           else
70441                                   *tail_acc_ptr= next_acc;
70442                   }
70443                   else
70444                           head_acc= next_acc;
70445           }
70446           if (tcp_conn->tc_state == TCS_CLOSED)
70447           {
70448                   while (head_acc)
70449                   {
70450                           next_acc= head_acc->acc_ext_link;
70451                           bf_afree(head_acc);
70452                           head_acc= next_acc;
70453                   }
70454                   return;
70455           }
70456           tcp_conn->tc_rcv_queue= head_acc;
70457   }
70458
70459   PRIVATE acc_t *merge_packs(first, next)
70460   acc_t *first;
70461   acc_t *next;
70462   {
70463           tcp_hdr_t *first_hdr, *next_hdr;
70464           int first_hdr_len, next_hdr_len, first_data_len, next_data_len;
70465           acc_t *next_acc, *tmp_acc;
70466
70467           assert (first->acc_length >= TCP_MIN_HDR_SIZE);
70468           assert (next->acc_length >= TCP_MIN_HDR_SIZE);
70469
70470           first_hdr= (tcp_hdr_t *)ptr2acc_data(first);
70471           next_hdr= (tcp_hdr_t *)ptr2acc_data(next);
70472
70473           first_hdr_len= (first_hdr->th_data_off & TH_DO_MASK) >>2;
70474           next_hdr_len= (next_hdr->th_data_off & TH_DO_MASK) >> 2;
70475
70476           first_data_len= first_hdr->th_chksum-first_hdr_len;
70477           next_data_len= next_hdr->th_chksum-next_hdr_len;
70478
70479   assert (tcp_LEmod4G(first_hdr->th_seq_nr, next_hdr->th_seq_nr));
70480   assert (first_hdr_len + first_data_len == bf_bufsize(first));
70481   #if DEBUG
70482    if (next_hdr_len + next_data_len != bf_bufsize(next))
70483    { ip_panic(( "fatal error: %d + %d != %d\n", next_hdr_len, next_data_len,
70484       bf_bufsize(next) )); }
```

```
70485    #endif
70486    assert (next_hdr_len + next_data_len == bf_bufsize(next));
70487
70488            if (tcp_Lmod4G(first_hdr->th_seq_nr+first_data_len,
70489                    next_hdr->th_seq_nr))
70490            {
70491                    first->acc_ext_link= next;
70492                    return first;
70493            }
70494            if (first_hdr->th_seq_nr == next_hdr->th_seq_nr)
70495                    if (first_data_len <= next_data_len)
70496                    {
70497                            bf_afree(first);
70498                            return next;
70499                    }
70500                    else
70501                    {
70502                            first->acc_ext_link= next->acc_ext_link;
70503                            bf_afree(next);
70504                            return first;
70505                    }
70506            if (tcp_GEmod4G(first_hdr->th_seq_nr+first_data_len,
70507                    next_hdr->th_seq_nr+next_data_len))
70508            {
70509                    first->acc_ext_link= next->acc_ext_link;
70510                    bf_afree(next);
70511                    return first;
70512            }
70513            first_data_len= next_hdr->th_seq_nr-first_hdr->th_seq_nr;
70514            first_hdr->th_chksum= first_data_len+first_hdr_len+
70515                    next_data_len;
70516            tmp_acc= bf_cut(first, 0, first_hdr_len + first_data_len);
70517            bf_afree(first);
70518            first= tmp_acc;
70519
70520            if (next_hdr->th_flags & THF_PSH)
70521                    first_hdr->th_flags |= THF_PSH;
70522            if (next_hdr->th_flags & THF_URG)
70523            {
70524                    if (!(first_hdr->th_flags & THF_URG))
70525                    {
70526                            first_hdr->th_flags |= THF_URG;
70527                            first_hdr->th_urgptr= next_hdr->th_seq_nr+
70528                                    next_hdr->th_urgptr-
70529                                    first_hdr->th_seq_nr;
70530                    }
70531                    else if (next_hdr->th_seq_nr+next_hdr->th_urgptr-
70532                            first_hdr->th_seq_nr > first_hdr->th_urgptr)
70533                            first_hdr->th_urgptr= next_hdr->th_seq_nr+
70534                                    next_hdr->th_urgptr-
70535                                    first_hdr->th_seq_nr;
70536            }
70537
70538            next_acc= next->acc_ext_link;
70539            tmp_acc= bf_cut(next, next_hdr_len ,next_data_len);
70540            bf_afree(next);
70541            next= tmp_acc;
70542            tmp_acc= bf_append (first, next);
70543            tmp_acc->acc_ext_link= next_acc;
70544            return tmp_acc;
```

```
70545   }

70547   PRIVATE void create_RST(tcp_conn, ip_hdr, tcp_hdr)
70548   tcp_conn_t *tcp_conn;
70549   ip_hdr_t *ip_hdr;
70550   tcp_hdr_t *tcp_hdr;
70551   {
70552           acc_t *tmp_ipopt, *tmp_tcpopt;
70553           ip_hdropt_t ip_hdropt;
70554           tcp_hdropt_t tcp_hdropt;
70555           acc_t *RST_acc;
70556           ip_hdr_t *RST_ip_hdr;
70557           tcp_hdr_t *RST_tcp_hdr;
70558           char *ptr2RSThdr;
70559           size_t pack_size;
70560
70561           tmp_ipopt= tcp_conn->tc_remipopt;
70562           if (tmp_ipopt)
70563                   tmp_ipopt->acc_linkC++;
70564           tmp_tcpopt= tcp_conn->tc_remtcpopt;
70565           if (tmp_tcpopt)
70566                   tmp_tcpopt->acc_linkC++;
70567
70568           tcp_extract_ipopt (tcp_conn, ip_hdr);
70569           tcp_extract_tcpopt (tcp_conn, tcp_hdr);
70570
70571           RST_acc= tcp_make_header (tcp_conn, &RST_ip_hdr, &RST_tcp_hdr,
70572                   (acc_t *)0);
70573           if (!RST_acc)
70574           {
70575   #if DEBUG
70576    { where(); printf("connection closed while inuse\n"); }
70577   #endif
70578                   return;
70579           }
70580
70581           if (tcp_conn->tc_remipopt)
70582                   bf_afree(tcp_conn->tc_remipopt);
70583           tcp_conn->tc_remipopt= tmp_ipopt;
70584           if (tcp_conn->tc_remtcpopt)
70585                   bf_afree(tcp_conn->tc_remtcpopt);
70586           tcp_conn->tc_remtcpopt= tmp_tcpopt;
70587
70588           RST_ip_hdr->ih_src= ip_hdr->ih_dst;
70589           RST_ip_hdr->ih_dst= ip_hdr->ih_src;
70590
70591           RST_tcp_hdr->th_srcport= tcp_hdr->th_dstport;
70592           RST_tcp_hdr->th_dstport= tcp_hdr->th_srcport;
70593           if (tcp_hdr->th_flags & THF_ACK)
70594           {
70595                   RST_tcp_hdr->th_seq_nr= tcp_hdr->th_ack_nr;
70596                   RST_tcp_hdr->th_flags= THF_RST;
70597           }
70598           else
70599           {
70600                   RST_tcp_hdr->th_seq_nr= 0;
70601                   RST_tcp_hdr->th_ack_nr= htonl(ntohl(tcp_hdr->th_seq_nr)+
70602                           tcp_hdr->th_chksum-((tcp_hdr->th_data_off &
70603                           TH_DO_MASK) >> 2)+ (tcp_hdr->th_flags &
70604                           THF_SYN ? 1 : 0) + (tcp_hdr->th_flags &
```

```
70605                           THF_FIN ? 1 : 0));
70606                   RST_tcp_hdr->th_flags= THF_RST|THF_ACK;
70607           }
70608
70609           pack_size= bf_bufsize(RST_acc);
70610           RST_ip_hdr->ih_length= htons(pack_size);
70611           RST_tcp_hdr->th_window= htons(tcp_conn->tc_rcv_wnd);
70612           RST_tcp_hdr->th_chksum= 0;
70613           RST_tcp_hdr->th_chksum= ~tcp_pack_oneCsum (RST_acc, pack_size);
70614
70615           if (tcp_conn->tc_frag2send)
70616                   bf_afree(tcp_conn->tc_frag2send);
70617           tcp_conn->tc_frag2send= RST_acc;
70618   }

70620   PUBLIC void tcp_restart_fd_read (tcp_conn)
70621   tcp_conn_t *tcp_conn;
70622   {
70623           tcp_fd_t *new_fd, *hi_fd, *urgent_fd, *normal_fd, *tcp_fd;
70624
70625   #if DEBUG & 256
70626    { where(); printf("tcp_restart_fd_read called\n"); }
70627   #endif
70628           do
70629           {
70630                   tcp_fd= tcp_conn->tc_readuser;
70631
70632                   if (tcp_fd)
70633                           fd_read (tcp_fd);
70634                   else
70635                           tcp_fd= &tcp_fd_table[TCP_FD_NR-1];
70636
70637                   if (!tcp_conn->tc_readuser)
70638                   {
70639                           urgent_fd= 0;
70640                           normal_fd= 0;
70641                           for (new_fd= tcp_fd+1, hi_fd=
70642                                   &tcp_fd_table[TCP_FD_NR]; new_fd<hi_fd;
70643                                   new_fd++)
70644                                   switch_read_fd(tcp_conn, new_fd,
70645                                           &urgent_fd, &normal_fd);
70646                           for (new_fd= tcp_fd_table, hi_fd= tcp_fd+1;
70647                                   new_fd < hi_fd; new_fd++)
70648                                   switch_read_fd(tcp_conn, new_fd,
70649                                           &urgent_fd, &normal_fd);
70650                           if (urgent_fd)
70651                                   tcp_fd= urgent_fd;
70652                           else
70653                                   tcp_fd= normal_fd;
70654                           tcp_conn->tc_readuser= tcp_fd;
70655                   }
70656                   else
70657                           return;
70658           } while (tcp_conn->tc_readuser);
70659   }

70661   PRIVATE void switch_read_fd (tcp_conn, new_fd, ref_urg_fd,
70662           ref_norm_fd)
70663   tcp_conn_t *tcp_conn;
70664   tcp_fd_t *new_fd, **ref_urg_fd, **ref_norm_fd;
```

```
70665   {
70666           if (!(new_fd->tf_flags & TFF_INUSE))
70667                   return;
70668           if (new_fd->tf_conn != tcp_conn)
70669                   return;
70670           if (!(new_fd->tf_flags & TFF_READ_IP))
70671                   return;
70672           if (new_fd->tf_flags & TFF_RECV_URG)
70673           {
70674                   if (!*ref_urg_fd)
70675                           *ref_urg_fd= new_fd;
70676           }
70677           else
70678           {
70679                   if (!*ref_norm_fd)
70680                           *ref_norm_fd= new_fd;
70681           }
70682   }

70684   PRIVATE void fd_read(tcp_fd)
70685   tcp_fd_t *tcp_fd;
70686   {
70687           tcp_conn_t *tcp_conn;
70688           size_t data_size, read_size;
70689           acc_t * data;
70690           int urg, result;
70691           int more2write;
70692
70693           more2write= FALSE;
70694           tcp_conn= tcp_fd->tf_conn;
70695
70696           assert (tcp_fd->tf_flags & TFF_READ_IP);
70697           if (tcp_conn->tc_state == TCS_CLOSED)
70698           {
70699                   tcp_conn->tc_readuser= 0;
70700   #if DEBUG
70701    { where(); printf("calling tcp_reply_read\n"); }
70702   #endif
70703                   if (tcp_fd->tf_read_offset)
70704                           tcp_reply_read (tcp_fd, tcp_fd->tf_read_offset);
70705                   else
70706                           tcp_reply_read (tcp_fd, tcp_conn->tc_error);
70707                   return;
70708           }
70709
70710   #if URG_PROPERLY_IMPLEMENTED
70711           urg= (tcp_GEmod4G(tcp_conn->tc_RCV_UP, tcp_conn->tc_RCV_LO) &&
70712                   tcp_Lmod4G(tcp_conn->tc_RCV_UP, tcp_conn->tc_RCV_NXT));
70713   #else
70714   #define urg 0
70715   #endif
70716           if (urg && !(tcp_fd->tf_flags & TFF_RECV_URG))
70717           {
70718                   tcp_conn->tc_readuser= 0;
70719   #if DEBUG
70720    { where(); printf("calling tcp_reply_read\n"); }
70721   #endif
70722                   if (tcp_fd->tf_read_offset)
70723                           tcp_reply_read (tcp_fd, tcp_fd->tf_read_offset);
70724                   else
```

```
70725                           tcp_reply_read (tcp_fd, EURG);
70726                   return;
70727           }
70728           else if (!urg && (tcp_fd->tf_flags & TFF_RECV_URG))
70729           {
70730                   tcp_conn->tc_readuser= 0;
70731   #if DEBUG
70732    { where(); printf("calling tcp_reply_read\n"); }
70733   #endif
70734                   if (tcp_fd->tf_read_offset)
70735                           tcp_reply_read (tcp_fd, tcp_fd->tf_read_offset);
70736                   else
70737                           tcp_reply_read(tcp_fd, ENOURG);
70738                   return;
70739           }
70740           data_size= tcp_conn->tc_RCV_NXT-tcp_conn->tc_RCV_LO;
70741           if (tcp_conn->tc_flags & TCF_FIN_RECV)
70742                   data_size--;
70743           if (urg)
70744                   read_size= tcp_conn->tc_RCV_UP+1-tcp_conn->tc_RCV_LO;
70745           else
70746                   read_size= data_size;
70747
70748           if (read_size>tcp_fd->tf_read_count)
70749                   read_size= tcp_fd->tf_read_count;
70750
70751           if (read_size)
70752           {
70753                   if (read_size == data_size)
70754                           data= bf_dupacc(tcp_conn->tc_rcvd_data);
70755                   else
70756                           data= bf_cut(tcp_conn->tc_rcvd_data, 0, read_size);
70757                   result= (*tcp_fd->tf_put_userdata) (tcp_fd->tf_srfd,
70758                           tcp_fd->tf_read_offset, data, FALSE);
70759                   if (tcp_conn->tc_state == TCS_CLOSED)
70760                   {
70761   #if DEBUG
70762    { where(); printf("connection closed while inuse\n"); }
70763   #endif
70764                           return;
70765                   }
70766                   if (result<0)
70767                   {
70768                           tcp_conn->tc_readuser= 0;
70769   #if DEBUG
70770    { where(); printf("calling tcp_reply_read\n"); }
70771   #endif
70772                           if (tcp_fd->tf_read_offset)
70773                                   tcp_reply_read(tcp_fd, tcp_fd->
70774                                           tf_read_offset);
70775                           else
70776                                   tcp_reply_read(tcp_fd, result);
70777                           return;
70778                   }
70779                   tcp_fd->tf_read_offset += read_size;
70780                   tcp_fd->tf_read_count -= read_size;
70781
70782                   if (data_size == read_size)
70783                   {
70784                           bf_afree(tcp_conn->tc_rcvd_data);
```

```
70785                           tcp_conn->tc_rcvd_data= 0;
70786                   }
70787                   else
70788                   {
70789                           data= tcp_conn->tc_rcvd_data;
70790                           tcp_conn->tc_rcvd_data= bf_cut(data,
70791                                   read_size, data_size-read_size);
70792                           bf_afree(data);
70793                   }
70794                   tcp_conn->tc_RCV_LO += read_size;
70795                   data_size -= read_size;
70796           }
70797           if (tcp_conn->tc_RCV_HI-tcp_conn->tc_RCV_LO < (tcp_conn->
70798                   tc_rcv_wnd >> 2))
70799           {
70800                   tcp_conn->tc_RCV_HI= tcp_conn->tc_RCV_LO +
70801                           tcp_conn->tc_rcv_wnd;
70802                   tcp_conn->tc_flags |= TCF_SEND_ACK;
70803   #if DEBUG & 256
70804    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
70805           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
70806   #endif
70807                   more2write= TRUE;
70808           }
70809           if (!data_size && (tcp_conn->tc_flags & TCF_RCV_PUSH))
70810           {
70811                   tcp_conn->tc_flags &= ~TCF_RCV_PUSH;
70812   #if DEBUG & 256
70813    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
70814           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
70815   #endif
70816                   if (tcp_fd->tf_read_offset)
70817                   {
70818                           tcp_conn->tc_readuser= 0;
70819   #if DEBUG & 256
70820    { where(); printf("calling tcp_reply_read\n"); }
70821   #endif
70822                           tcp_reply_read (tcp_fd, tcp_fd->tf_read_offset);
70823                           if (more2write)
70824                                   tcp_restart_write(tcp_conn);
70825                           return;
70826                   }
70827           }
70828           if ((tcp_conn->tc_flags & TCF_FIN_RECV) ||
70829                   !tcp_fd->tf_read_count)
70830           {
70831                   tcp_conn->tc_readuser= 0;
70832   #if DEBUG & 256
70833    { where(); printf("calling tcp_reply_read\n"); }
70834   #endif
70835                   tcp_reply_read (tcp_fd, tcp_fd->tf_read_offset);
70836                   if (more2write)
70837                           tcp_restart_write(tcp_conn);
70838                   return;
70839           }
70840           if (more2write)
70841                   tcp_restart_write(tcp_conn);
70842   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/tcp_send.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

70900    /*
70901    tcp_send.c
70902    */
70903
70904    #include "inet.h"
70905    #include "buf.h"
70906    #include "clock.h"
70907    #include "type.h"
70908
70909    #include "assert.h"
70910    #include "ip.h"
70911    #include "tcp.h"
70912    #include "tcp_int.h"
70913
70914    INIT_PANIC();
70915
70916    #if DEBUG
70917    #include "tcp_delay.h"
70918
70919    int tcp_delay_on;
70920    u32_t tcp_delay;
70921    #endif
70922
70923    #if DEBUG & 2
70924    #define d_bufcut 1
70925    #endif
70926
70927    FORWARD void write2port ARGS(( tcp_port_t *tcp_port,
70928            acc_t *data ));
70929    FORWARD void major_to ARGS(( int conn, struct timer *timer ));
70930    FORWARD void minor_to ARGS(( int conn, struct timer *timer ));
70931    FORWARD void ack_to ARGS(( int conn, struct timer *timer ));
70932    FORWARD void time_wait_to ARGS(( int conn,
70933            struct timer *timer ));
70934    FORWARD acc_t *make_pack ARGS(( tcp_conn_t *tcp_conn ));
70935    FORWARD void fd_write ARGS(( tcp_fd_t *tcp_fd ));
70936    FORWARD void switch_write_fd ARGS(( tcp_conn_t *tcp_conn,
70937            tcp_fd_t *new_fd, tcp_fd_t **ref_urgent_fd,
70938            tcp_fd_t **ref_normal_fd, tcp_fd_t **ref_shutdown_fd ));
70939    FORWARD void tcp_restart_write_conn ARGS (( tcp_conn_t *tcp_conn ));
70940    #if DEBUG
70941    FORWARD void tcp_delay_to ARGS(( int ref, timer_t *timer ));
70942    #endif
70943
70944    PUBLIC void tcp_restart_write (tcp_conn)
70945    tcp_conn_t *tcp_conn;
70946    {
70947            tcp_port_t *tcp_port;
70948
70949    #if DEBUG & 256
70950     { where(); printf("in tcp_restart_write\n"); }
70951    #endif
70952    assert (tcp_conn->tc_flags & TCF_INUSE);
70953
70954            tcp_conn->tc_flags |= TCF_MORE2WRITE;
```

```
70955   #if DEBUG & 256
70956    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
70957           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
70958   #endif
70959           tcp_port= tcp_conn->tc_port;
70960
70961           tcp_port->tp_flags |= TPF_MORE2WRITE;
70962
70963           if (tcp_port->tp_flags & TPF_WRITE_IP)
70964                   return;
70965
70966           tcp_restart_write_port (tcp_port);
70967   }

70969   PUBLIC void tcp_restart_write_port (tcp_port)
70970   tcp_port_t *tcp_port;
70971   {
70972           tcp_conn_t *tcp_conn, *hi_conn;
70973
70974   #if DEBUG & 256
70975    { where(); printf("in tcp_restart_write_port\n"); }
70976   #endif
70977   assert (tcp_port->tp_flags & TPF_MORE2WRITE);
70978   assert (!(tcp_port->tp_flags & TPF_WRITE_IP));
70979
70980           while(tcp_port->tp_flags & TPF_MORE2WRITE)
70981           {
70982                   tcp_port->tp_flags &= ~TPF_MORE2WRITE;
70983                   for (tcp_conn= tcp_conn_table, hi_conn=
70984                           &tcp_conn_table[TCP_CONN_NR]; tcp_conn<hi_conn;
70985                           tcp_conn++)
70986                   {
70987                           if ((tcp_conn->tc_flags & (TCF_INUSE|
70988                                   TCF_MORE2WRITE)) != (TCF_INUSE|
70989                                   TCF_MORE2WRITE))
70990                                   continue;
70991                           if (tcp_conn->tc_port != tcp_port)
70992                                   continue;
70993                           tcp_conn->tc_flags &= ~TCF_MORE2WRITE;
70994   #if DEBUG & 256
70995    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
70996           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
70997   #endif
70998   #if DEBUG & 256
70999    { where(); printf("calling tcp_restart_write_conn(&tcp_conn_table[%d])\n",
71000           tcp_conn - tcp_conn_table); }
71001   #endif
71002                           tcp_restart_write_conn (tcp_conn);
71003                           if (tcp_port->tp_flags & TPF_WRITE_IP)
71004                           {
71005   #if DEBUG & 256
71006    { where(); printf("setting TCF_MORE2WRITE because of TPF_WRITE_IP\n"); }
71007   #endif
71008                                   tcp_port->tp_flags |= TPF_MORE2WRITE;
71009                                   return;
71010                           }
71011                           tcp_port->tp_flags |= TPF_MORE2WRITE;
71012                   }
71013           }
71014   }
```

```
71016   PRIVATE void tcp_restart_write_conn (tcp_conn)
71017   tcp_conn_t *tcp_conn;
71018   {
71019           ip_hdr_t *ip_hdr;
71020           tcp_hdr_t *tcp_hdr;
71021           acc_t *pack2write;
71022
71023   #if DEBUG & 256
71024    { where(); printf("in tcp_restart_write_conn\n"); }
71025   #endif
71026   assert (tcp_conn->tc_flags & TCF_INUSE);
71027
71028           if (tcp_conn->tc_port->tp_flags & TPF_WRITE_IP)
71029           {
71030                   tcp_conn->tc_flags |= TCF_MORE2WRITE;
71031   #if DEBUG & 16
71032    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
71033           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
71034   #endif
71035                   tcp_conn->tc_port->tp_flags |= TPF_MORE2WRITE;
71036                   return;
71037           }
71038
71039           if (tcp_conn->tc_frag2send)
71040           {
71041                   pack2write= tcp_conn->tc_frag2send;
71042                   tcp_conn->tc_frag2send= 0;
71043   #if DEBUG & 256
71044    { where(); printf("calling write2port\n"); }
71045   #endif
71046                   write2port(tcp_conn->tc_port, pack2write);
71047
71048                   if (tcp_conn->tc_port->tp_flags & TPF_WRITE_IP)
71049                   {
71050                           tcp_conn->tc_flags |= TCF_MORE2WRITE;
71051   #if DEBUG & 16
71052    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
71053           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
71054   #endif
71055                           tcp_conn->tc_port->tp_flags |= TPF_MORE2WRITE;
71056                           return;
71057                   }
71058           }
71059
71060           if (tcp_conn->tc_state == TCS_CLOSED || tcp_conn->tc_state ==
71061                   TCS_LISTEN)
71062                   return;
71063
71064
71065           if (tcp_conn->tc_no_retrans >
71066                   tcp_conn->tc_max_no_retrans)
71067           {
71068   #if DEBUG
71069    { where(); printf("calling tcp_close_connection\n"); }
71070   #endif
71071                   tcp_close_connection(tcp_conn, ETIMEDOUT);
71072                   return;
71073           }
71074
```

```
71075           while (pack2write= make_pack(tcp_conn))
71076           {
71077   /*              if (tcp_conn->tc_state == TCS_CLOSED)
71078                           return; */ /* XXX Why is this? */
71079                   if (tcp_conn->tc_port->tp_flags & TPF_WRITE_IP)
71080                   {
71081                           tcp_conn->tc_frag2send= pack2write;
71082                           tcp_conn->tc_flags |= TCF_MORE2WRITE;
71083   #if DEBUG & 16
71084    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
71085           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
71086   #endif
71087                           tcp_conn->tc_port->tp_flags |= TPF_MORE2WRITE;
71088                           return;
71089                   }
71090                   write2port(tcp_conn->tc_port, pack2write);
71091                   if (tcp_conn->tc_port->tp_flags & TPF_WRITE_IP)
71092                   {
71093                           tcp_conn->tc_flags |= TCF_MORE2WRITE;
71094   #if DEBUG & 16
71095    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
71096           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
71097   #endif
71098                           tcp_conn->tc_port->tp_flags |= TPF_MORE2WRITE;
71099                           return;
71100                   }
71101           }
71102   }

71104   PRIVATE acc_t *make_pack(tcp_conn)
71105   tcp_conn_t *tcp_conn;
71106   {
71107           acc_t *pack2write, *tmp_pack;
71108           tcp_hdr_t *tcp_hdr;
71109           ip_hdr_t *ip_hdr;
71110           int tot_hdr_size;
71111           u32_t seg_seq, seg_lo_data, queue_lo_data, seg_hi, seg_hi_data;
71112           u16_t seg_up;
71113           u8_t seg_flags;
71114           time_t new_dis;
71115           size_t pack_size;
71116           time_t major_timeout, minor_timeout;
71117
71118   #if DEBUG & 256
71119    { where(); printf("make_pack called\n"); }
71120   #endif
71121           switch (tcp_conn->tc_state)
71122           {
71123           case TCS_CLOSED:
71124                   return 0;
71125           case TCS_SYN_RECEIVED:
71126           case TCS_SYN_SENT:
71127                   if (tcp_conn->tc_SND_TRM == tcp_conn->tc_SND_NXT &&
71128                           !(tcp_conn->tc_flags & TCF_SEND_ACK))
71129                   {
71130   #if DEBUG & 256
71131    { where(); printf("make_pack returned\n"); }
71132   #endif
71133                           return 0;
71134                   }
```

```
71135                   major_timeout= 0;
71136                   tcp_conn->tc_flags &= ~TCF_SEND_ACK;
71137   #if DEBUG & 256
71138    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
71139           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
71140   #endif
71141
71142                   pack2write= tcp_make_header(tcp_conn, &ip_hdr, &tcp_hdr,
71143                           (acc_t *)0);
71144                   if (!pack2write)
71145                   {
71146    { where(); printf("connection closed while inuse\n"); }
71147   #if DEBUG
71148    { where(); printf("make_pack returned\n"); }
71149   #endif
71150                           return 0;
71151                   }
71152                   tot_hdr_size= bf_bufsize(pack2write);
71153                   seg_seq= tcp_conn->tc_SND_TRM;
71154                   if (tcp_conn->tc_state == TCS_SYN_SENT)
71155                           seg_flags= 0;
71156                   else
71157                           seg_flags= THF_ACK;     /* except for TCS_SYN_SENT
71158                                                    * ack is always present */
71159
71160                   if (seg_seq == tcp_conn->tc_ISS)
71161                   {
71162                           seg_flags |= THF_SYN;
71163                           tcp_conn->tc_SND_TRM++;
71164                           if (!tcp_conn->tc_ett)
71165                                   tcp_conn->tc_ett= get_time();
71166                                   /* fill in estimated transmition time field */
71167
71168                           major_timeout= get_time() + (tcp_conn->tc_rtt *
71169                                   (tcp_conn->tc_no_retrans + 2));
71170                   }
71171                   tcp_hdr->th_seq_nr= htonl(seg_seq);
71172                   tcp_hdr->th_ack_nr= htonl(tcp_conn->tc_RCV_NXT);
71173                   tcp_hdr->th_flags= seg_flags;
71174                   tcp_hdr->th_window= htons(tcp_conn->tc_mss);
71175                           /* Initially we allow one segment */
71176
71177                   ip_hdr->ih_length= htons(tot_hdr_size);
71178                   tcp_hdr->th_chksum= ~tcp_pack_oneCsum(pack2write, tot_hdr_size);
71179
71180                   new_dis= get_time() + 2*HZ*tcp_conn->tc_ttl;
71181                   if (new_dis > tcp_conn->tc_senddis)
71182                           tcp_conn->tc_senddis= new_dis;
71183
71184                   if (major_timeout)
71185                   {
71186                           tcp_conn->tc_no_retrans++;
71187   #if DEBUG & 256
71188    { where(); printf("setting major_to\n"); }
71189   #endif
71190                           clck_timer(&tcp_conn->tc_major_timer,
71191                                   major_timeout, major_to,
71192                                   tcp_conn-tcp_conn_table);
71193                   }
71194                   if (tcp_conn->tc_flags & TCF_ACK_TIMER_SET)
```

```
71195                   {
71196                           tcp_conn->tc_flags &= ~TCF_ACK_TIMER_SET;
71197   #if DEBUG & 16
71198    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
71199           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
71200   #endif
71201                           clck_untimer(&tcp_conn->tc_ack_timer);
71202                   }
71203   #if DEBUG & 256
71204    { where(); printf("make_pack returned\n"); }
71205   #endif
71206                   return pack2write;
71207
71208                   break;
71209
71210           case TCS_ESTABLISHED:
71211           case TCS_FIN_WAIT_1:
71212           case TCS_FIN_WAIT_2:
71213           case TCS_CLOSE_WAIT:
71214           case TCS_CLOSING:
71215           case TCS_LAST_ACK:
71216           case TCS_TIME_WAIT:
71217
71218   #if DEBUG & 256
71219    { where(); printf("SND_TRM= 0x%x, snd_cwnd= 0x%x\n", tcp_conn->tc_SND_TRM,
71220           tcp_conn->tc_snd_cwnd); }
71221   #endif
71222   assert (tcp_LEmod4G(tcp_conn->tc_SND_TRM, tcp_conn->tc_snd_cwnd));
71223   assert (tcp_LEmod4G(tcp_conn->tc_SND_TRM, tcp_conn->tc_SND_NXT));
71224
71225                   if ((tcp_conn->tc_SND_TRM == tcp_conn->tc_snd_cwnd ||
71226                           tcp_conn->tc_SND_TRM == tcp_conn->tc_SND_NXT) &&
71227                           !(tcp_conn->tc_flags & TCF_SEND_ACK))
71228                   {
71229   #if DEBUG & 256
71230    { where(); printf("nothing to do\n"); }
71231   #endif
71232   #if DEBUG & 256
71233    { where(); printf("make_pack returned\n"); }
71234   #endif
71235                           return 0;
71236                   }
71237
71238                   major_timeout= 0;
71239                   tcp_conn->tc_flags &= ~TCF_SEND_ACK;
71240   #if DEBUG & 256
71241    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
71242           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
71243   #endif
71244
71245                   pack2write= tcp_make_header (tcp_conn, &ip_hdr,
71246                           &tcp_hdr, (acc_t *)0);
71247                   if (!pack2write)
71248                   {
71249    { where(); printf("connection closed while inuse\n"); }
71250   #if DEBUG
71251    { where(); printf("make_pack returned\n"); }
71252   #endif
71253                           return 0;
71254                   }
```

```
71255    bf_chkbuf(pack2write);
71256                    tot_hdr_size= bf_bufsize(pack2write);
71257                    seg_seq= tcp_conn->tc_SND_TRM;
71258                    seg_flags= THF_ACK;
71259    assert(tcp_Gmod4G(seg_seq, tcp_conn->tc_ISS));
71260
71261                    seg_lo_data= seg_seq;
71262                    queue_lo_data= tcp_conn->tc_SND_UNA;
71263    assert(tcp_Gmod4G(queue_lo_data, tcp_conn->tc_ISS));
71264    assert (tcp_check_conn(tcp_conn));
71265
71266                    seg_hi= tcp_conn->tc_SND_NXT;
71267                    seg_hi_data= seg_hi;
71268                    if (tcp_conn->tc_flags & TCF_FIN_SENT)
71269                    {
71270    #if DEBUG & 256
71271     { where(); printf("Setting FIN flags\n"); }
71272    #endif
71273                            if (seg_seq != seg_hi)
71274                                    seg_flags |= THF_FIN;
71275                            if (queue_lo_data == seg_hi_data)
71276                                    queue_lo_data--;
71277                            if (seg_lo_data == seg_hi_data)
71278                                    seg_lo_data--;
71279                            seg_hi_data--;
71280                    }
71281
71282                    if (seg_hi_data - seg_lo_data > tcp_conn->tc_mss -
71283                            tot_hdr_size)
71284                    {
71285                            seg_hi_data= seg_lo_data + tcp_conn->tc_mss -
71286                                    tot_hdr_size;
71287                            seg_hi= seg_hi_data;
71288    #if DEBUG & 256
71289     { where(); printf("Clearing FIN flags\n"); }
71290    #endif
71291                            seg_flags &= ~THF_FIN;
71292                    }
71293                    if (tcp_Gmod4G(seg_hi_data, tcp_conn->tc_snd_cwnd))
71294                    {
71295                            seg_hi_data= tcp_conn->tc_snd_cwnd;
71296                            seg_hi= seg_hi_data;
71297    #if DEBUG & 256
71298     { where(); printf("Clearing FIN flags\n"); }
71299    #endif
71300                            seg_flags &= ~THF_FIN;
71301                    }
71302
71303                    if (tcp_Gmod4G(tcp_conn->tc_SND_UP, seg_lo_data) &&
71304                            tcp_LEmod4G(tcp_conn->tc_SND_UP, seg_hi_data))
71305                    {
71306                            seg_up= tcp_conn->tc_SND_UP-seg_seq;
71307                            seg_flags |= THF_URG;
71308    #if DEBUG
71309     { where(); printf("seg_up= %d\n", seg_up); }
71310    #endif
71311                    }
71312                    if (tcp_Gmod4G(tcp_conn->tc_SND_PSH, seg_lo_data) &&
71313                            tcp_LEmod4G(tcp_conn->tc_SND_PSH, seg_hi_data))
71314                    {
```

```
71315                           seg_flags |= THF_PSH;
71316                   }
71317
71318                   tcp_conn->tc_SND_TRM= seg_hi;
71319
71320                   if (seg_hi-seg_seq)
71321                   {
71322                           if (!tcp_conn->tc_ett)
71323                                   tcp_conn->tc_ett= get_time();
71324
71325                           if (seg_seq == tcp_conn->tc_SND_UNA)
71326                                   major_timeout= get_time() + (tcp_conn->tc_rtt *
71327                                   (tcp_conn->tc_no_retrans + 1));
71328                   }
71329                   if (seg_hi_data-seg_lo_data)
71330                   {
71331                           tmp_pack= pack2write;
71332                           while (tmp_pack->acc_next)
71333                                   tmp_pack= tmp_pack->acc_next;
71334   assert (tcp_check_conn(tcp_conn));
71335   bf_chkbuf(pack2write);
71336                           tmp_pack->acc_next= bf_cut(tcp_conn->tc_send_data,
71337                                   (unsigned)(seg_lo_data-queue_lo_data),
71338                                   (unsigned) (seg_hi_data-seg_lo_data));
71339   bf_chkbuf(pack2write);
71340                   }
71341
71342                   tcp_hdr->th_seq_nr= htonl(seg_seq);
71343                   tcp_hdr->th_ack_nr= htonl(tcp_conn->tc_RCV_NXT);
71344                   tcp_hdr->th_flags= seg_flags;
71345                   tcp_hdr->th_window= htons(tcp_conn->tc_RCV_HI -
71346                           tcp_conn->tc_RCV_NXT);
71347                   tcp_hdr->th_urgptr= htons(seg_up);
71348
71349                   pack_size= bf_bufsize(pack2write);
71350                   ip_hdr->ih_length= htons(pack_size);
71351   bf_chkbuf(pack2write);
71352                   tcp_hdr->th_chksum= ~tcp_pack_oneCsum(pack2write,
71353                           pack_size);
71354   bf_chkbuf(pack2write);
71355                   new_dis= get_time() + 2*HZ*tcp_conn->tc_ttl;
71356                   if (new_dis > tcp_conn->tc_senddis)
71357                           tcp_conn->tc_senddis= new_dis;
71358   bf_chkbuf(pack2write);
71359                   if (major_timeout)
71360                   {
71361                           tcp_conn->tc_no_retrans++;
71362
71363   #if DEBUG & 256
71364    { where(); printf("setting major_to\n"); }
71365   #endif
71366                           clck_timer(&tcp_conn->tc_major_timer,
71367                                   major_timeout,
71368                                   major_to, tcp_conn-tcp_conn_table);
71369                   }
71370                   if (tcp_conn->tc_flags & TCF_ACK_TIMER_SET)
71371                   {
71372                           tcp_conn->tc_flags &= ~TCF_ACK_TIMER_SET;
71373   #if DEBUG & 256
71374    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
```

```
71375           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
71376   #endif
71377                           clck_untimer(&tcp_conn->tc_ack_timer);
71378                   }
71379   #if DEBUG & 256
71380    { where(); printf("make_pack returned\n"); }
71381   #endif
71382                   return pack2write;
71383           default:
71384   #if DEBUG
71385    { where(); printf("tcp_conn_table[%d].tc_state= %d\n", tcp_conn-tcp_conn_table,
71386           tcp_conn->tc_state); }
71387   #endif
71388                   ip_panic(( "Illegal state" ));
71389           }
71390   #if DEBUG
71391    { where(); printf("make_pack returned\n"); }
71392   #endif
71393   }

71395   PRIVATE void write2port(tcp_port, data)
71396   tcp_port_t *tcp_port;
71397   acc_t *data;
71398   {
71399           int result;
71400
71401   #if DEBUG & 256
71402    { where(); printf("in write2port\n"); }
71403   #endif
71404           assert (!(tcp_port->tp_flags & TPF_WRITE_IP));
71405
71406           tcp_port->tp_flags |= TPF_WRITE_IP;
71407           tcp_port->tp_pack= data;
71408
71409   bf_chkbuf(data);
71410   #if DEBUG
71411           if (tcp_delay_on)
71412           {
71413                   if (tcp_port->tp_flags & TPF_DELAY_TCP)
71414                           tcp_port->tp_flags &= ~TPF_DELAY_TCP;
71415                   else
71416                   {
71417                           tcp_port->tp_flags |= TPF_WRITE_SP;
71418                           clck_timer(&tcp_port->tp_delay_tim,
71419                                   get_time() + tcp_delay,
71420                                   tcp_delay_to, tcp_port-
71421                                   tcp_port_table);
71422                           return;
71423                   }
71424           }
71425   #endif
71426           result= ip_write (tcp_port->tp_ipfd, bf_bufsize(data));
71427
71428           if (result == NW_SUSPEND)
71429           {
71430                   tcp_port->tp_flags |= TPF_WRITE_SP;
71431                   return;
71432           }
71433   assert(result == NW_OK);
71434           tcp_port->tp_flags &= ~TPF_WRITE_IP;
```

```
71435   assert(!(tcp_port->tp_flags & (TPF_WRITE_IP|TPF_WRITE_SP)));
71436   }

71438   PRIVATE void major_to(conn, timer)
71439   int conn;
71440   struct timer *timer;
71441   {
71442           tcp_conn_t *tcp_conn;
71443           u16_t mss, mss2;
71444
71445   #if DEBUG & 256
71446    { where(); printf("in major_to\n"); }
71447   #endif
71448           tcp_conn= &tcp_conn_table[conn];
71449   #if DEBUG & 256
71450    { where(); printf("major_to: snd_cwnd-SND_UNA= %lu, no_retrans: %d snd_cthresh= %d\n",
71451           tcp_conn->tc_snd_cwnd-tcp_conn->tc_SND_UNA, tcp_conn->tc_no_retrans,
71452           tcp_conn->tc_snd_cthresh); }
71453   #endif
71454           assert(tcp_conn->tc_flags & TCF_INUSE);
71455           assert(tcp_conn->tc_state != TCS_CLOSED &&
71456                   tcp_conn->tc_state != TCS_LISTEN);
71457
71458           clck_untimer(&tcp_conn->tc_minor_timer);
71459           tcp_conn->tc_SND_TRM= tcp_conn->tc_SND_UNA;
71460
71461           mss= tcp_conn->tc_mss;
71462           mss2= 2*mss;
71463
71464           tcp_conn->tc_snd_cwnd= tcp_conn->tc_SND_TRM + mss2;
71465   #if DEBUG & 256
71466    { where(); printf("snd_cwnd is now %d\n", tcp_conn->tc_snd_cwnd); }
71467   #endif
71468           tcp_conn->tc_snd_cthresh /= 2;
71469           if (tcp_conn->tc_snd_cthresh < mss2)
71470                   tcp_conn->tc_snd_cthresh= mss2;
71471           tcp_conn->tc_snd_cinc= ((unsigned long)mss*mss)/tcp_conn->
71472                   tc_snd_cthresh;
71473
71474           tcp_restart_write(tcp_conn);
71475   }

71477   PRIVATE void minor_to(conn, timer)
71478   int conn;
71479   struct timer *timer;
71480   {
71481           tcp_conn_t *tcp_conn;
71482
71483   #if DEBUG
71484    { where(); printf("in minor_to\n"); }
71485   #endif
71486           tcp_conn= &tcp_conn_table[conn];
71487           assert(tcp_conn->tc_flags & TCF_INUSE);
71488           assert(tcp_conn->tc_state != TCS_CLOSED &&
71489                   tcp_conn->tc_state != TCS_LISTEN);
71490
71491           tcp_restart_write(tcp_conn);
71492   }

71494   PUBLIC void tcp_release_retrans(tcp_conn, seg_ack, new_win)
```

```
71495   tcp_conn_t *tcp_conn;
71496   u32_t seg_ack;
71497   u16_t new_win;
71498   {
71499           size_t size, offset;
71500           acc_t *old_pack, *new_pack;
71501           time_t retrans_time, curr_time;
71502           u32_t queue_lo, queue_hi;
71503           u16_t cwnd, incr, mss, mss2, cthresh;
71504
71505   #if DEBUG & 256
71506    { where(); printf("in release_retrans(&tcp_conn_table[%d], 0x%x, %d)\n",
71507           tcp_conn-tcp_conn_table, seg_ack, new_win); }
71508   #endif
71509   assert (tcp_GEmod4G(seg_ack, tcp_conn->tc_SND_UNA));
71510   assert (tcp_LEmod4G(seg_ack, tcp_conn->tc_SND_NXT));
71511
71512           if (tcp_Gmod4G(seg_ack, tcp_conn->tc_SND_UNA))
71513           {
71514                   tcp_conn->tc_no_retrans= 0;
71515                   curr_time= get_time();
71516                   if (curr_time < tcp_conn->tc_ett)
71517                           retrans_time= 0;
71518                   else
71519                           retrans_time= curr_time-tcp_conn->tc_ett;
71520                   if (tcp_conn->tc_rtt*2 < retrans_time)
71521                   {
71522                           tcp_conn->tc_rtt *= 2;
71523   assert (tcp_conn->tc_rtt);
71524                   }
71525                   else if (tcp_conn->tc_rtt > retrans_time*2)
71526                   {
71527                           tcp_conn->tc_rtt= tcp_conn->tc_rtt/2+1;
71528   assert (tcp_conn->tc_rtt);
71529                   }
71530
71531                   if (seg_ack == tcp_conn->tc_SND_NXT)
71532                           tcp_conn->tc_ett= 0;
71533                   else
71534                   {
71535                           tcp_conn->tc_ett += (seg_ack-tcp_conn->
71536                                   tc_SND_UNA) * tcp_conn->tc_rtt /
71537                                   (tcp_conn->tc_SND_NXT-tcp_conn->
71538                                   tc_SND_UNA);
71539                   }
71540                   queue_lo= tcp_conn->tc_SND_UNA;
71541                   queue_hi= tcp_conn->tc_SND_NXT;
71542
71543                   tcp_conn->tc_SND_UNA= seg_ack;
71544                   if (tcp_Lmod4G(tcp_conn->tc_SND_TRM, seg_ack))
71545                           tcp_conn->tc_SND_TRM= seg_ack;
71546                   if (tcp_Lmod4G(tcp_conn->tc_snd_cwnd, seg_ack))
71547                   {
71548                           tcp_conn->tc_snd_cwnd= seg_ack;
71549   #if DEBUG & 256
71550    { where(); printf("snd_cwnd is now %d\n", tcp_conn->tc_snd_cwnd); }
71551   #endif
71552                   }
71553
71554                   if (queue_lo == tcp_conn->tc_ISS)
```

```
71555                           queue_lo++;
71556
71557                   if (tcp_conn->tc_flags & TCF_FIN_SENT)
71558                   {
71559                           if (seg_ack == queue_hi)
71560                                   seg_ack--;
71561                           if (queue_lo == queue_hi)
71562                                   queue_lo--;
71563                           queue_hi--;
71564                   }
71565
71566                   offset= seg_ack - queue_lo;
71567                   size= queue_hi - seg_ack;
71568                   old_pack= tcp_conn->tc_send_data;
71569                   tcp_conn->tc_send_data= 0;
71570
71571                   if (!size)
71572                   {
71573                           new_pack= 0;
71574                           tcp_conn->tc_snd_cwnd= tcp_conn->tc_SND_UNA +
71575                                   2*tcp_conn->tc_mss;
71576                           /* Reset window if a write is completed */
71577   #if DEBUG & 256
71578    { where(); printf("snd_cwnd is now %d\n", tcp_conn->tc_snd_cwnd); }
71579   #endif
71580                   }
71581                   else
71582                           new_pack= bf_cut(old_pack, offset, size);
71583                   bf_afree(old_pack);
71584                   tcp_conn->tc_send_data= new_pack;
71585   assert (tcp_check_conn(tcp_conn));
71586
71587                   if (tcp_conn->tc_state == TCS_CLOSED)
71588                   {
71589            { where(); printf("connection closed while inuse\n"); }
71590                           return;
71591                   }
71592
71593                   if (tcp_conn->tc_ett)
71594                   {
71595   #if DEBUG & 256
71596    { where(); printf("setting major_to\n"); }
71597   #endif
71598                           clck_timer(&tcp_conn->tc_major_timer,
71599                           tcp_conn->tc_ett + tcp_conn->tc_rtt,
71600                           major_to, tcp_conn-tcp_conn_table);
71601                   }
71602           }
71603
71604           mss= tcp_conn->tc_mss;
71605           cthresh= tcp_conn->tc_snd_cthresh;
71606           mss2= 2*mss;
71607
71608           if (new_win < mss2)
71609           {
71610                   cwnd= 0;
71611                   if (new_win >= mss)
71612                           incr= mss;
71613                   else
71614                   {
```

```
71615                           if (new_win)
71616                                   incr= new_win;
71617                           else
71618                                   incr= 1;
71619                   }
71620           }
71621           else
71622           {
71623                   cwnd= tcp_conn->tc_snd_cwnd - tcp_conn->tc_SND_UNA;
71624                   incr= mss2;
71625                   if (cwnd+incr > new_win)
71626                   {
71627                           incr= mss;
71628                           if (cwnd+incr > new_win)
71629                                   incr= 0;
71630                   }
71631           }
71632
71633   assert (cthresh >= mss2);
71634
71635           if (incr && cwnd+incr >  cthresh)
71636           {
71637                   incr -= mss;
71638                   if (incr && cwnd+incr >  cthresh)
71639                           incr -= mss;
71640           }
71641           if (cwnd+incr+mss>cthresh && cthresh<tcp_conn->tc_snd_wnd)
71642                   tcp_conn->tc_snd_cthresh += tcp_conn->tc_snd_cinc;
71643
71644           tcp_conn->tc_snd_cwnd= tcp_conn->tc_SND_UNA+cwnd+incr;
71645   #if DEBUG & 256
71646    { where(); printf("snd_cwnd is now 0x%x\n", tcp_conn->tc_snd_cwnd); }
71647   #endif
71648           if (tcp_Gmod4G(tcp_conn->tc_SND_TRM, tcp_conn->tc_snd_cwnd))
71649                   tcp_conn->tc_SND_TRM= tcp_conn->tc_snd_cwnd;
71650
71651           if (tcp_Gmod4G(tcp_conn->tc_snd_cwnd, tcp_conn->tc_SND_TRM) &&
71652                   tcp_Gmod4G(tcp_conn->tc_SND_NXT, tcp_conn->tc_SND_TRM))
71653                   tcp_restart_write(tcp_conn);
71654
71655           if (tcp_conn->tc_writeuser)
71656                   tcp_restart_fd_write(tcp_conn);
71657
71658   assert (tcp_check_conn(tcp_conn));
71659
71660   }

71662   PUBLIC void tcp_restart_fd_write(tcp_conn)
71663   tcp_conn_t *tcp_conn;
71664   {
71665           tcp_fd_t *urgent_fd, *normal_fd, *shutdown_fd,
71666                   *new_fd, *hi_fd, *tcp_fd;
71667           int closed_connection;
71668
71669   #if DEBUG & 256
71670    { where(); printf("in restart_fd_write\n"); }
71671   #endif
71672           do
71673           {
71674                   tcp_fd= tcp_conn->tc_writeuser;
```

```
71675
71676                   closed_connection= (tcp_conn->tc_state == TCS_CLOSED);
71677                   if (tcp_fd)
71678                           fd_write(tcp_fd);
71679                   else
71680                           tcp_fd= &tcp_fd_table[TCP_FD_NR-1];
71681
71682                   if (!closed_connection &&
71683                           tcp_conn->tc_state == TCS_CLOSED)
71684                   {
71685   #if DEBUG
71686     { where(); printf("connection closed while inuse\n"); }
71687   #endif
71688                           return;
71689                   }
71690
71691                   if (!tcp_conn->tc_writeuser)
71692                   {
71693                           urgent_fd= 0;
71694                           normal_fd= 0;
71695                           shutdown_fd= 0;
71696                           for (new_fd= tcp_fd+1, hi_fd=
71697                                   &tcp_fd_table[TCP_FD_NR]; new_fd<hi_fd;
71698                                   new_fd++)
71699                                   switch_write_fd(tcp_conn, new_fd,
71700                                           &urgent_fd, &normal_fd,
71701                                           &shutdown_fd);
71702                           for (new_fd= tcp_fd_table, hi_fd= tcp_fd+1;
71703                                   new_fd < hi_fd; new_fd++)
71704                                   switch_write_fd(tcp_conn, new_fd,
71705                                           &urgent_fd, &normal_fd,
71706                                           &shutdown_fd);
71707                           if (urgent_fd)
71708                                   tcp_fd= urgent_fd;
71709                           else if (normal_fd)
71710                                   tcp_fd= normal_fd;
71711                           else
71712                                   tcp_fd= shutdown_fd;
71713                           tcp_conn->tc_writeuser= tcp_fd;
71714                   }
71715                   else
71716                           return;
71717           } while (tcp_conn->tc_writeuser);
71718   }



71722   PRIVATE void switch_write_fd (tcp_conn, new_fd, ref_urg_fd,
71723           ref_norm_fd, ref_shut_fd)
71724   tcp_conn_t *tcp_conn;
71725   tcp_fd_t *new_fd, **ref_urg_fd, **ref_norm_fd, **ref_shut_fd;
71726   {
71727           if (!(new_fd->tf_flags & TFF_INUSE))
71728                   return;
71729           if (new_fd->tf_conn != tcp_conn)
71730                   return;
71731           if (new_fd->tf_flags & TFF_WRITE_IP)
71732                   if (new_fd->tf_flags & TFF_WR_URG)
71733                   {
71734                           if (!*ref_urg_fd)
```

```
71735                                   *ref_urg_fd= new_fd;
71736                   }
71737                   else
71738                   {
71739                           if (!*ref_norm_fd)
71740                                   *ref_norm_fd= new_fd;
71741                   }
71742           else if ((new_fd->tf_flags & TFF_IOCTL_IP) &&
71743                   new_fd->tf_ioreq == NWIOTCPSHUTDOWN)
71744                   if (!*ref_shut_fd)
71745                           *ref_shut_fd= new_fd;
71746   }

71748   PRIVATE void fd_write (tcp_fd)
71749   tcp_fd_t *tcp_fd;
71750   {
71751           tcp_conn_t *tcp_conn;
71752           int urg, push;
71753           u32_t max_seq;
71754           size_t max_count, max_trans, write_count, send_count;
71755           acc_t *data, *tmp_acc, *send_data;
71756           int restart_write;
71757
71758           restart_write= FALSE;
71759           tcp_conn= tcp_fd->tf_conn;
71760
71761           if (tcp_fd->tf_flags & TFF_IOCTL_IP)
71762           {
71763   assert (tcp_fd->tf_ioreq == NWIOTCPSHUTDOWN);
71764                   if (tcp_conn->tc_state == TCS_CLOSED)
71765                   {
71766                           tcp_conn->tc_writeuser= 0;
71767                           tcp_reply_ioctl (tcp_fd, tcp_conn->tc_error);
71768                           return;
71769                   }
71770                   tcp_conn->tc_writeuser= 0;
71771                   tcp_reply_ioctl (tcp_fd, NW_OK);
71772                   tcp_shutdown (tcp_conn);
71773                   return;
71774           }
71775           assert (tcp_fd->tf_flags & TFF_WRITE_IP);
71776           if (tcp_conn->tc_state == TCS_CLOSED)
71777           {
71778                   tcp_conn->tc_writeuser= 0;
71779   #if DEBUG & 256
71780    { where(); printf("calling tcp_reply_write()\n"); }
71781   #endif
71782                   if (tcp_fd->tf_write_offset)
71783                           tcp_reply_write(tcp_fd,
71784                                   tcp_fd->tf_write_offset);
71785                   else
71786                           tcp_reply_write(tcp_fd, tcp_conn->tc_error);
71787                   return;
71788           }
71789   assert (!(tcp_conn->tc_flags & TCF_FIN_SENT));
71790   assert (tcp_conn->tc_SND_UNA != tcp_conn->tc_ISS);
71791
71792           urg= (tcp_fd->tf_flags & TFF_WR_URG);
71793           push= (tcp_fd->tf_flags & TFF_PUSH_DATA);
71794   #if DEBUG & 256
```

```
71795     if (push) { where(); printf("pushing data\n"); }
71796     #endif
71797
71798             max_seq= tcp_conn->tc_SND_UNA + tcp_conn->tc_snd_wnd;
71799             if (urg)
71800                     max_seq += tcp_conn->tc_urg_wnd;
71801             max_count= max_seq - tcp_conn->tc_SND_UNA;
71802             max_trans= max_seq - tcp_conn->tc_SND_NXT;
71803             if (tcp_fd->tf_write_count <= max_trans)
71804                     write_count= tcp_fd->tf_write_count;
71805             else if (!urg && max_trans < max_count/2)
71806                     return;
71807             else
71808                     write_count= max_trans;
71809             if (write_count)
71810             {
71811                     data= (*tcp_fd->tf_get_userdata)
71812                             (tcp_fd->tf_srfd, tcp_fd->tf_write_offset,
71813                             write_count, FALSE);
71814                     if (tcp_conn->tc_state == TCS_CLOSED)
71815                     {
71816     { where(); printf("connection closed while inuse\n"); }
71817                             if (data)
71818                                     bf_afree(data);
71819                             return;
71820                     }
71821                     if (!data)
71822                     {
71823                             tcp_conn->tc_writeuser= 0;
71824     assert(data);
71825     #if DEBUG
71826     { where(); printf("calling tcp_reply_write()\n"); }
71827     #endif
71828                             if (tcp_fd->tf_write_offset)
71829                                     tcp_reply_write(tcp_fd,
71830                                             tcp_fd->tf_write_offset);
71831                             else
71832                                     tcp_reply_write(tcp_fd, EFAULT);
71833                             return;
71834                     }
71835                     tcp_fd->tf_write_offset += write_count;
71836                     tcp_fd->tf_write_count -= write_count;
71837
71838                     send_data= tcp_conn->tc_send_data;
71839                     tcp_conn->tc_send_data= 0;
71840                     send_data= bf_append(send_data, data);
71841                     if (tcp_conn->tc_state == TCS_CLOSED)
71842                     {
71843     { where(); printf("connection closed while inuse\n"); }
71844                             bf_afree(send_data);
71845                             return;
71846                     }
71847                     tcp_conn->tc_send_data= send_data;
71848                     tcp_conn->tc_SND_NXT += write_count;
71849                     if (urg)
71850                             tcp_conn->tc_SND_UP= tcp_conn->tc_SND_NXT;
71851                     if (push && !tcp_fd->tf_write_count)
71852                             tcp_conn->tc_SND_PSH= tcp_conn->tc_SND_NXT;
71853
71854     assert (tcp_check_conn(tcp_conn));
```

```
71855                   if (tcp_Gmod4G(tcp_conn->tc_SND_NXT, tcp_conn->tc_SND_TRM) &&
71856                           tcp_Lmod4G(tcp_conn->tc_SND_TRM, tcp_conn->tc_snd_cwnd))
71857                           restart_write= TRUE;
71858           }
71859           if (!tcp_fd->tf_write_count)
71860           {
71861                   tcp_conn->tc_writeuser= 0;
71862   #if DEBUG & 256
71863    { where(); printf("calling tcp_reply_write()\n"); }
71864   #endif
71865                   tcp_reply_write(tcp_fd, tcp_fd->tf_write_offset);
71866           }
71867           if (restart_write)
71868                   tcp_restart_write(tcp_conn);
71869   }

71871   PUBLIC void tcp_shutdown(tcp_conn)
71872   tcp_conn_t *tcp_conn;
71873   {
71874           int closed_connection;
71875
71876   #if DEBUG & 256
71877    { where(); printf("in tcp_shutdown\n"); }
71878   #endif
71879           if (tcp_conn->tc_flags & TCF_FIN_SENT)
71880                   return;
71881           tcp_conn->tc_flags |= TCF_FIN_SENT;
71882   #if DEBUG & 256
71883    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
71884           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
71885   #endif
71886           tcp_conn->tc_SND_NXT++;
71887
71888           switch (tcp_conn->tc_state)
71889           {
71890           case TCS_CLOSED:
71891           case TCS_LISTEN:
71892           case TCS_SYN_SENT:
71893           case TCS_SYN_RECEIVED:
71894   #if DEBUG & 256
71895    { where(); printf("calling tcp_close_connection\n"); }
71896   #endif
71897                   tcp_close_connection(tcp_conn, ENOTCONN);
71898                   break;
71899           case TCS_ESTABLISHED:
71900                   tcp_conn->tc_state= TCS_FIN_WAIT_1;
71901   #if DEBUG
71902    { where(); tcp_write_state(tcp_conn); }
71903   #endif
71904                   break;
71905           case TCS_CLOSE_WAIT:
71906                   tcp_conn->tc_state= TCS_LAST_ACK;
71907   #if DEBUG
71908    { where(); tcp_write_state(tcp_conn); }
71909   #endif
71910                   break;
71911           }
71912           closed_connection= (tcp_conn->tc_state == TCS_CLOSED);
71913
71914   assert (tcp_check_conn(tcp_conn));
```

```
71915
71916   #if DEBUG & 256
71917    { where(); printf("calling tcp_restart_write\n"); }
71918   #endif
71919                   tcp_restart_write(tcp_conn);
71920   }

71922   PUBLIC void tcp_set_time_wait_timer(tcp_conn)
71923   tcp_conn_t *tcp_conn;
71924   {
71925           assert (tcp_conn->tc_state == TCS_TIME_WAIT);
71926
71927   #if DEBUG & 256
71928    { where(); printf("tcp_set_time_wait_timer, ttl= %d\n", tcp_conn->tc_ttl); }
71929   #endif
71930           clck_timer(&tcp_conn->tc_time_wait_timer, get_time() +
71931                   tcp_conn->tc_ttl * 2L * HZ, time_wait_to, tcp_conn-
71932                   tcp_conn_table);
71933   }

71935   PUBLIC void tcp_set_ack_timer (tcp_conn)
71936   tcp_conn_t *tcp_conn;
71937   {
71938           if (tcp_conn->tc_flags & TCF_ACK_TIMER_SET)
71939                   return;
71940
71941           tcp_conn->tc_flags |= TCF_ACK_TIMER_SET;
71942   #if DEBUG & 256
71943    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
71944           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
71945   #endif
71946
71947   #if DEBUG & 256
71948    { where(); printf("setting ack_timer\n"); };
71949   #endif
71950           clck_timer(&tcp_conn->tc_ack_timer, get_time() +
71951                   TCP_ACK_DELAY, ack_to, tcp_conn-tcp_conn_table);
71952   }

71954   /*
71955   tcp_close_connection
71956
71957   */
71958
71959   PUBLIC void tcp_close_connection(tcp_conn, error)
71960   tcp_conn_t *tcp_conn;
71961   int error;
71962   {
71963   #if DEBUG & 256
71964    { where(); printf("closing connection\n"); }
71965   #endif
71966
71967   assert (tcp_check_conn(tcp_conn));
71968   assert (tcp_conn->tc_flags & TCF_INUSE);
71969
71970           tcp_conn->tc_error= error;
71971           if (tcp_conn->tc_state == TCS_CLOSED)
71972                   return;
71973
71974           clck_untimer (&tcp_conn->tc_major_timer);
```

```
71975           clck_untimer (&tcp_conn->tc_minor_timer);
71976   #if DEBUG & 256
71977    { where(); printf("clearing ack_timer\n"); }
71978   #endif
71979           clck_untimer (&tcp_conn->tc_ack_timer);
71980           clck_untimer (&tcp_conn->tc_time_wait_timer);
71981
71982           tcp_conn->tc_state= TCS_CLOSED;
71983   #if DEBUG & 16
71984    { where(); tcp_write_state(tcp_conn); }
71985   #endif
71986
71987           if (tcp_conn->tc_readuser)
71988                   tcp_restart_fd_read (tcp_conn);
71989   assert (!tcp_conn->tc_readuser);
71990
71991           if (tcp_conn->tc_writeuser)
71992                   tcp_restart_fd_write (tcp_conn);
71993   assert (!tcp_conn->tc_writeuser);
71994
71995           if (tcp_conn->tc_connuser)
71996           {
71997   #if DEBUG & 256
71998    { where(); printf("closing and connuser present\n"); }
71999   #endif
72000                   tcp_restart_connect (tcp_conn->tc_connuser);
72001           }
72002   assert (!tcp_conn->tc_connuser);
72003
72004           if (tcp_conn->tc_rcvd_data)
72005           {
72006                   bf_afree(tcp_conn->tc_rcvd_data);
72007                   tcp_conn->tc_rcvd_data= 0;
72008           }
72009           tcp_conn->tc_flags &= ~TCF_FIN_RECV;
72010   #if DEBUG & 256
72011    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
72012           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
72013   #endif
72014           tcp_conn->tc_RCV_LO= tcp_conn->tc_RCV_NXT;
72015
72016           if (tcp_conn->tc_rcv_queue)
72017           {
72018                   bf_afree(tcp_conn->tc_rcv_queue);
72019                   tcp_conn->tc_rcv_queue= 0;
72020           }
72021
72022           if (tcp_conn->tc_send_data)
72023           {
72024   #if DEBUG & 256
72025    { where(); printf("releasing data\n"); }
72026   #endif
72027   #if DEBUG & 256
72028    { where(); printf("SND_TRM= 0x%lx, tc_SND_NXT= 0x%lx, SND_UNA=  0x%lx\n",
72029           tcp_conn->tc_SND_TRM, tcp_conn->tc_SND_NXT, tcp_conn->tc_SND_UNA); }
72030   #endif
72031                   bf_afree(tcp_conn->tc_send_data);
72032                   tcp_conn->tc_send_data= 0;
72033                   tcp_conn->tc_SND_TRM=
72034                           tcp_conn->tc_SND_NXT= tcp_conn->tc_SND_UNA;
```

```
72035           }
72036           tcp_conn->tc_SND_TRM= tcp_conn->tc_SND_NXT= tcp_conn->tc_SND_UNA;
72037
72038           if (tcp_conn->tc_remipopt)
72039           {
72040                   bf_afree(tcp_conn->tc_remipopt);
72041                   tcp_conn->tc_remipopt= 0;
72042           }
72043
72044           if (tcp_conn->tc_remtcpopt)
72045           {
72046                   bf_afree(tcp_conn->tc_remtcpopt);
72047                   tcp_conn->tc_remtcpopt= 0;
72048           }
72049
72050           if (tcp_conn->tc_frag2send)
72051           {
72052                   bf_afree(tcp_conn->tc_frag2send);
72053                   tcp_conn->tc_remtcpopt= 0;
72054           }
72055                                           /* clear all flags but TCF_INUSE */
72056           tcp_conn->tc_flags &= TCF_INUSE;
72057   #if DEBUG & 256
72058    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
72059           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
72060   #endif
72061   assert (tcp_check_conn(tcp_conn));
72062   }

72064   PRIVATE void ack_to(conn, timer)
72065   int conn;
72066   struct timer *timer;
72067   {
72068           tcp_conn_t *tcp_conn;
72069
72070   #if DEBUG & 256
72071    { where(); printf("in ack_to\n"); }
72072   #endif
72073           tcp_conn= &tcp_conn_table[conn];
72074
72075           assert (&tcp_conn->tc_ack_timer == timer);
72076
72077           assert (tcp_conn->tc_flags & TCF_ACK_TIMER_SET);
72078
72079           tcp_conn->tc_flags &= ~TCF_ACK_TIMER_SET;
72080   #if DEBUG & 256
72081    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
72082           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
72083   #endif
72084           tcp_conn->tc_flags |= TCF_SEND_ACK;
72085   #if DEBUG & 256
72086    { where(); printf("tcp_conn_table[%d].tc_flags= 0x%x\n",
72087           tcp_conn-tcp_conn_table, tcp_conn->tc_flags); }
72088   #endif
72089           tcp_restart_write (tcp_conn);
72090   }

72092   PRIVATE void time_wait_to(conn, timer)
72093   int conn;
72094   struct timer *timer;
```

```
72095   {
72096           tcp_conn_t *tcp_conn;
72097
72098           tcp_conn= &tcp_conn_table[conn];
72099
72100           assert (tcp_conn->tc_state == TCS_TIME_WAIT);
72101           assert (&tcp_conn->tc_time_wait_timer == timer);
72102
72103   #if DEBUG
72104    { where(); printf("calling tcp_close_connection\n"); }
72105   #endif
72106           tcp_close_connection (tcp_conn, ENOCONN);
72107   }

72109   PUBLIC void tcp_zero_wnd_to(conn, timer)
72110   int conn;
72111   struct timer *timer;
72112   {
72113           tcp_conn_t *tcp_conn;
72114
72115   #if DEBUG & 256
72116    { where(); printf("in tcp_zero_wnd_to\n"); }
72117   #endif
72118           tcp_conn= &tcp_conn_table[conn];
72119
72120           assert (&tcp_conn->tc_major_timer == timer);
72121           assert (tcp_conn->tc_0wnd_to);
72122
72123           tcp_conn->tc_0wnd_to *= 2;
72124           tcp_conn->tc_SND_TRM= tcp_conn->tc_SND_UNA;
72125
72126           tcp_restart_write (tcp_conn);
72127   }

72129   #if DEBUG
72130   PRIVATE void tcp_delay_to(ref, timer)
72131   int ref;
72132   timer_t *timer;
72133   {
72134           tcp_port_t *tcp_port;
72135
72136   assert(ref >= 0 && ref < TCP_PORT_NR);
72137           tcp_port= &tcp_port_table[ref];
72138   assert(timer == &tcp_port->tp_delay_tim);
72139
72140           tcp_port->tp_flags &= ~(TPF_WRITE_SP|TPF_WRITE_IP);
72141           tcp_port->tp_flags |= TPF_DELAY_TCP;
72142           if (tcp_port->tp_flags & TPF_MORE2WRITE)
72143           {
72144   #if DEBUG & 256
72145   { where(); printf("calling tcp_restart_write_port\n"); }
72146   #endif
72147                   write2port(tcp_port, tcp_port->tp_pack);
72148           }
72149   }
72150   #endif
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        src/inet/generic/udp.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

72200   /*
72201   udp.c
72202   */
72203
72204   #include "inet.h"
72205
72206   #include "assert.h"
72207   #include "buf.h"
72208   #include "clock.h"
72209   #include "io.h"
72210   #include "ip.h"
72211   #include "sr.h"
72212   #include "type.h"
72213   #include "udp.h"
72214
72215   INIT_PANIC();
72216
72217   #define UDP_PORT_NR     1
72218   #define UDP_FD_NR       32
72219
72220   typedef struct udp_port
72221   {
72222           int up_flags;
72223           int up_state;
72224           int up_ipfd;
72225           int up_minor;
72226           int up_ipdev;
72227           acc_t *up_wr_pack;
72228           ipaddr_t up_ipaddr;
72229           struct udp_fd *up_next_fd;
72230           struct udp_fd *up_write_fd;
72231   } udp_port_t;
72232
72233   #define UPF_EMPTY       0x0
72234   #define UPF_WRITE_IP    0x1
72235   #define UPF_WRITE_SP    0x2
72236   #define UPF_READ_IP     0x4
72237   #define UPF_READ_SP     0x8
72238   #define UPF_SUSPEND     0x10
72239   #define UPF_MORE2WRITE  0x20
72240
72241   #define UPS_EMPTY       0
72242   #define UPS_SETPROTO    1
72243   #define UPS_GETCONF     2
72244   #define UPS_MAIN        3
72245   #define UPS_ERROR       4
72246
72247   typedef struct udp_fd
72248   {
72249           int uf_flags;
72250           udp_port_t *uf_port;
72251           int uf_ioreq;
72252           int uf_srfd;
72253           nwio_udpopt_t uf_udpopt;
72254           get_userdata_t uf_get_userdata;
```

```
72255           put_userdata_t uf_put_userdata;
72256           acc_t *uf_pack;
72257           acc_t *uf_rd_buf;
72258           size_t uf_rd_count;
72259           size_t uf_wr_count;
72260           time_t uf_exp_tim;
72261   } udp_fd_t;
72262
72263   #define UFF_EMPTY       0x0
72264   #define UFF_INUSE       0x1
72265   #define UFF_IOCTL_IP    0x2
72266   #define UFF_READ_IP     0x4
72267   #define UFF_WRITE_IP    0x8
72268   #define UFF_OPTSET      0x10
72269
72270   FORWARD void read_ip_packets ARGS(( udp_port_t *udp_port ));
72271   FORWARD void udp_buffree ARGS(( int priority, size_t reqsize ));
72272   FORWARD void udp_main ARGS(( udp_port_t *udp_port ));
72273   FORWARD acc_t *udp_get_data ARGS(( int fd, size_t offset, size_t count,
72274           int for_ioctl ));
72275   FORWARD int udp_put_data ARGS(( int fd, size_t offset, acc_t *data,
72276           int for_ioctl ));
72277   FORWARD void udp_restart_write_port ARGS(( udp_port_t *udp_port ));
72278   FORWARD void process_inc_fragm ARGS(( udp_port_t *udp_port, acc_t *data ));
72279   FORWARD int reply_thr_put ARGS(( udp_fd_t *ucp_fd, int reply,
72280           int for_ioctl ));
72281   FORWARD void reply_thr_get ARGS(( udp_fd_t *udp_fd, int reply,
72282           int for_ioctl ));
72283   FORWARD int udp_setopt ARGS(( udp_fd_t *udp_fd ));
72284   FORWARD udpport_t find_unused_port ARGS(( int fd ));
72285   FORWARD int is_unused_port ARGS(( Udpport_t port ));
72286   FORWARD int udp_packet2user ARGS(( udp_fd_t *udp_fd ));
72287   FORWARD void restart_write_fd ARGS(( udp_fd_t *udp_fd ));
72288   FORWARD u16_t pack_oneCsum ARGS(( acc_t *pack ));
72289
72290   PRIVATE udp_port_t udp_port_table[UDP_PORT_NR];
72291   PRIVATE udp_fd_t udp_fd_table[UDP_FD_NR];
72292
72293   PUBLIC void udp_init()
72294   {
72295           udp_fd_t *udp_fd;
72296           udp_port_t *udp_port;
72297           int i, result;
72298
72299           assert (BUF_S >= sizeof(struct nwio_ipopt));
72300           assert (BUF_S >= sizeof(struct nwio_ipconf));
72301           assert (BUF_S >= sizeof(struct nwio_udpopt));
72302           assert (BUF_S >= sizeof(struct udp_io_hdr));
72303           assert (UDP_HDR_SIZE == sizeof(udp_hdr_t));
72304           assert (UDP_IO_HDR_SIZE == sizeof(udp_io_hdr_t));
72305
72306           udp_port_table[0].up_minor= UDP_DEV0;
72307           udp_port_table[0].up_ipdev= IP0;
72308
72309           for (i= 0, udp_fd= udp_fd_table; i<UDP_FD_NR; i++, udp_fd++)
72310           {
72311                   udp_fd->uf_flags= UFF_EMPTY;
72312           }
72313
72314           bf_logon(udp_buffree);
```

```
72315
72316           for (i= 0, udp_port= udp_port_table; i<UDP_PORT_NR; i++, udp_port++)
72317           {
72318                   udp_port->up_flags= UPF_EMPTY;
72319                   udp_port->up_state= UPS_EMPTY;
72320                   udp_port->up_next_fd= udp_fd_table;
72321                   udp_port->up_write_fd= NULL;
72322
72323                   result= sr_add_minor (udp_port->up_minor,
72324                           udp_port-udp_port_table, udp_open, udp_close, udp_read,
72325                           udp_write, udp_ioctl, udp_cancel);
72326                   assert (result >= 0);
72327
72328                   udp_main(udp_port);
72329           }
72330   }

72332   PRIVATE void udp_main(udp_port)
72333   udp_port_t *udp_port;
72334   {
72335           udp_fd_t *udp_fd;
72336           int result, i;
72337
72338           switch (udp_port->up_state)
72339           {
72340           case UPS_EMPTY:
72341                   udp_port->up_state= UPS_SETPROTO;
72342
72343                   udp_port->up_ipfd= ip_open(udp_port->up_ipdev,
72344                           udp_port-udp_port_table, udp_get_data, udp_put_data);
72345                   if (udp_port->up_ipfd < 0)
72346                   {
72347                           udp_port->up_state= UPS_ERROR;
72348                           printf("%s, %d: unable to open ip port\n", __FILE__,
72349                                   __LINE__);
72350                           return;
72351                   }
72352
72353                   result= ip_ioctl(udp_port->up_ipfd, NWIOSIPOPT);
72354                   if (result == NW_SUSPEND)
72355                           udp_port->up_flags |= UPF_SUSPEND;
72356                   if (result<0)
72357                   {
72358                           return;
72359                   }
72360                   if (udp_port->up_state != UPS_GETCONF)
72361                           return;
72362                   /* drops through */
72363           case UPS_GETCONF:
72364                   udp_port->up_flags &= ~UPF_SUSPEND;
72365
72366                   result= ip_ioctl(udp_port->up_ipfd, NWIOGIPCONF);
72367                   if (result == NW_SUSPEND)
72368                           udp_port->up_flags |= UPF_SUSPEND;
72369                   if (result<0)
72370                   {
72371                           return;
72372                   }
72373                   if (udp_port->up_state != UPS_MAIN)
72374                           return;
```

```
72375                   /* drops through */
72376           case UPS_MAIN:
72377                   udp_port->up_flags &= ~UPF_SUSPEND;
72378
72379                   for (i= 0, udp_fd= udp_fd_table; i<UDP_FD_NR; i++, udp_fd++)
72380                   {
72381                           if (!(udp_fd->uf_flags & UFF_INUSE))
72382                                   continue;
72383                           if (udp_fd->uf_port != udp_port)
72384                                   continue;
72385                           if (udp_fd->uf_flags & UFF_IOCTL_IP)
72386                                   udp_ioctl(i, udp_fd->uf_ioreq);
72387                   }
72388                   read_ip_packets(udp_port);
72389                   return;
72390           default:
72391   #if DEBUG
72392    { where(); printf("udp_port_table[%d].up_state= %d\n", udp_port-udp_port_table,
72393           udp_port->up_state); }
72394   #endif
72395                   ip_panic(( "unknown state" ));
72396                   break;
72397           }
72398   }

72400   int udp_open (port, srfd, get_userdata, put_userdata)
72401   int port;
72402   int srfd;
72403   get_userdata_t get_userdata;
72404   put_userdata_t put_userdata;
72405   {
72406           int i;
72407           udp_fd_t *udp_fd;
72408
72409           for (i= 0; i<UDP_FD_NR && (udp_fd_table[i].uf_flags & UFF_INUSE);
72410                   i++);
72411
72412           if (i>= UDP_FD_NR)
72413           {
72414   #if DEBUG
72415    { where(); printf("out of fds\n"); }
72416   #endif
72417                   return EOUTOFBUFS;
72418           }
72419
72420           udp_fd= &udp_fd_table[i];
72421
72422           udp_fd->uf_flags= UFF_INUSE;
72423           udp_fd->uf_port= &udp_port_table[port];
72424           udp_fd->uf_srfd= srfd;
72425           udp_fd->uf_udpopt.nwuo_flags= UDP_DEF_OPT;
72426           udp_fd->uf_get_userdata= get_userdata;
72427           udp_fd->uf_put_userdata= put_userdata;
72428           udp_fd->uf_pack= 0;
72429
72430           return i;
72431
72432   }

72434   PRIVATE acc_t *udp_get_data (port, offset, count, for_ioctl)
```

```
72435   int port;
72436   size_t offset;
72437   size_t count;
72438   int for_ioctl;
72439   {
72440           udp_port_t *udp_port;
72441           udp_fd_t *udp_fd;
72442           int result;
72443
72444           udp_port= &udp_port_table[port];
72445
72446           switch(udp_port->up_state)
72447           {
72448           case UPS_SETPROTO:
72449   assert (for_ioctl);
72450                   if (!count)
72451                   {
72452                           result= (int)offset;
72453                           if (result<0)
72454                           {
72455                                   udp_port->up_state= UPS_ERROR;
72456                                   break;
72457                           }
72458                           udp_port->up_state= UPS_GETCONF;
72459                           if (udp_port->up_flags & UPF_SUSPEND)
72460                                   udp_main(udp_port);
72461                           return NULL;
72462                   }
72463                   else
72464                   {
72465                           struct nwio_ipopt *ipopt;
72466                           acc_t *acc;
72467
72468   assert (!offset);
72469   assert (count == sizeof(*ipopt));
72470
72471                           acc= bf_memreq(sizeof(*ipopt));
72472                           ipopt= (struct nwio_ipopt *)ptr2acc_data(acc);
72473                           ipopt->nwio_flags= NWIO_COPY | NWIO_EN_LOC |
72474                                   NWIO_EN_BROAD | NWIO_REMANY | NWIO_PROTOSPEC |
72475                                   NWIO_HDR_O_ANY | NWIO_RWDATALL;
72476                           ipopt->nwio_proto= IPPROTO_UDP;
72477                           return acc;
72478                   }
72479           case UPS_MAIN:
72480   assert (!for_ioctl);
72481   assert (udp_port->up_flags & UPF_WRITE_IP);
72482                   if (!count)
72483                   {
72484                           result= (int)offset;
72485   #if DEBUG & 256
72486    { where(); printf("result of ip_write is %d\n", result); }
72487   #endif
72488   assert (udp_port->up_wr_pack);
72489                           bf_afree(udp_port->up_wr_pack);
72490                           udp_port->up_wr_pack= 0;
72491                           if (udp_port->up_flags & UPF_WRITE_SP)
72492                           {
72493                                   if (udp_port->up_write_fd)
72494                                   {
```

```
72495                                           udp_fd= udp_port->up_write_fd;
72496                                           udp_port->up_write_fd= NULL;
72497                                           udp_fd->uf_flags &= ~UFF_WRITE_IP;
72498                                           reply_thr_get(udp_fd, result, FALSE);
72499                                   }
72500                                   udp_port->up_flags &= ~(UPF_WRITE_SP |
72501                                           UPF_WRITE_IP);
72502                                   if (udp_port->up_flags & UPF_MORE2WRITE)
72503                                   {
72504                                           udp_restart_write_port(udp_port);
72505                                   }
72506                           }
72507                           else
72508                                   udp_port->up_flags &= ~UPF_WRITE_IP;
72509                   }
72510                   else
72511                   {
72512                           return bf_cut (udp_port->up_wr_pack, offset, count);
72513                   }
72514                   break;
72515           default:
72516                   printf("udp_get_data(%d, 0x%x, 0x%x) called but up_state= 0x%x\n",
72517                           port, offset, count, udp_port->up_state);
72518                   break;
72519           }
72520           return NULL;
72521   }

72523   PRIVATE int udp_put_data (fd, offset, data, for_ioctl)
72524   int fd;
72525   size_t offset;
72526   acc_t *data;
72527   int for_ioctl;
72528   {
72529           udp_port_t *udp_port;
72530           int result;
72531
72532           udp_port= &udp_port_table[fd];
72533
72534           switch (udp_port->up_state)
72535           {
72536           case UPS_GETCONF:
72537                   if (!data)
72538                   {
72539                           result= (int)offset;
72540                           if (result<0)
72541                           {
72542                                   udp_port->up_state= UPS_ERROR;
72543                                   return NW_OK;
72544                           }
72545                           udp_port->up_state= UPS_MAIN;
72546                           if (udp_port->up_flags & UPF_SUSPEND)
72547                                   udp_main(udp_port);
72548                   }
72549                   else
72550                   {
72551                           struct nwio_ipconf *ipconf;
72552
72553                           data= bf_packIffLess(data, sizeof(*ipconf));
72554                           ipconf= (struct nwio_ipconf *)ptr2acc_data(data);
```

```
72555   assert (ipconf->nwic_flags & NWIC_IPADDR_SET);
72556                           udp_port->up_ipaddr= ipconf->nwic_ipaddr;
72557                           bf_afree(data);
72558                   }
72559                   break;
72560           case UPS_MAIN:
72561   assert (udp_port->up_flags & UPF_READ_IP);
72562                   if (!data)
72563                   {
72564                           result= (int)offset;
72565   compare (result, >=, 0);
72566                           if (udp_port->up_flags & UPF_READ_SP)
72567                           {
72568                                   udp_port->up_flags &= ~(UPF_READ_SP|
72569                                           UPF_READ_IP);
72570                                   read_ip_packets(udp_port);
72571                           }
72572                           else
72573                                   udp_port->up_flags &= ~UPF_READ_IP;
72574                   }
72575                   else
72576                   {
72577   assert (!offset);       /* This isn't a valid assertion but ip sends only
72578                            * whole datagrams up */
72579                           process_inc_fragm(udp_port, data);
72580                   }
72581                   break;
72582           default:
72583                   ip_panic((
72584                   "udp_put_data(%d, 0x%x, 0x%x) called but up_state= 0x%x\n",
72585                                           fd, offset, data, udp_port->up_state ));
72586           }
72587           return NW_OK;
72588   }

72590   int udp_ioctl (fd, req)
72591   int fd;
72592   int req;
72593   {
72594           udp_fd_t *udp_fd;
72595           udp_port_t *udp_port;
72596           nwio_udpopt_t *udp_opt;
72597           acc_t *opt_acc;
72598           int type;
72599           int result;
72600
72601           udp_fd= &udp_fd_table[fd];
72602           type= req & IOCTYPE_MASK;
72603
72604   assert (udp_fd->uf_flags & UFF_INUSE);
72605
72606           udp_port= udp_fd->uf_port;
72607           udp_fd->uf_flags |= UFF_IOCTL_IP;
72608           udp_fd->uf_ioreq= req;
72609
72610           if (udp_port->up_state != UPS_MAIN)
72611                   return NW_SUSPEND;
72612
72613           switch(type)
72614           {
```

```
72615           case NWIOSUDPOPT & IOCTYPE_MASK:
72616                   if (req != NWIOSUDPOPT)
72617                   {
72618                           reply_thr_get (udp_fd, EBADIOCTL, TRUE);
72619                           result= NW_OK;
72620                           break;
72621                   }
72622                   result= udp_setopt(udp_fd);
72623                   break;
72624           case NWIOGUDPOPT & IOCTYPE_MASK:
72625                   if (req != NWIOGUDPOPT)
72626                   {
72627                           reply_thr_put(udp_fd, EBADIOCTL, TRUE);
72628                           result= NW_OK;
72629                           break;
72630                   }
72631                   opt_acc= bf_memreq(sizeof(*udp_opt));
72632   assert (opt_acc->acc_length == sizeof(*udp_opt));
72633                   udp_opt= (nwio_udpopt_t *)ptr2acc_data(opt_acc);
72634
72635                   *udp_opt= udp_fd->uf_udpopt;
72636                   udp_opt->nwuo_locaddr= udp_fd->uf_port->up_ipaddr;
72637                   result= (*udp_fd->uf_put_userdata)(udp_fd->uf_srfd, 0, opt_acc,
72638                           TRUE);
72639                   if (result == NW_OK)
72640                           reply_thr_put(udp_fd, NW_OK, TRUE);
72641                   break;
72642           default:
72643                   reply_thr_get(udp_fd, EBADIOCTL, TRUE);
72644                   result= NW_OK;
72645                   break;
72646           }
72647           if (result != NW_SUSPEND)
72648                   udp_fd->uf_flags &= ~UFF_IOCTL_IP;
72649           return result;
72650   }

72652   PRIVATE int udp_setopt(udp_fd)
72653   udp_fd_t *udp_fd;
72654   {
72655           udp_fd_t *fd_ptr;
72656           nwio_udpopt_t oldopt, newopt;
72657           acc_t *data;
72658           int result;
72659           udpport_t port;
72660           unsigned int new_en_flags, new_di_flags, old_en_flags, old_di_flags,
72661                   all_flags, flags;
72662           unsigned long new_flags;
72663           int i;
72664
72665   #if DEBUG & 256
72666    { where(); printf("in udp_setopt\n"); }
72667   #endif
72668           data= (*udp_fd->uf_get_userdata)(udp_fd->uf_srfd, 0,
72669                   sizeof(nwio_udpopt_t), TRUE);
72670
72671           if (!data)
72672                   return EFAULT;
72673
72674           data= bf_packIffLess(data, sizeof(nwio_udpopt_t));
```

```
72675   assert (data->acc_length == sizeof(nwio_udpopt_t));
72676
72677           newopt= *(nwio_udpopt_t *)ptr2acc_data(data);
72678           bf_afree(data);
72679           oldopt= udp_fd->uf_udpopt;
72680   #if DEBUG & 256
72681    { where(); printf("newopt.nwuo_flags= 0x%x, newopt.nwuo_locport= %d, newopt.nwuo_remport= %d\n",
72682           newopt.nwuo_flags, ntohs(newopt.nwuo_locport),
72683           ntohs(newopt.nwuo_remport)); }
72684   #endif
72685
72686           old_en_flags= oldopt.nwuo_flags & 0xffff;
72687           old_di_flags= (oldopt.nwuo_flags >> 16) & 0xffff;
72688
72689           new_en_flags= newopt.nwuo_flags & 0xffff;
72690           new_di_flags= (newopt.nwuo_flags >> 16) & 0xffff;
72691
72692           if (new_en_flags & new_di_flags)
72693           {
72694   #if DEBUG
72695    { where(); printf("returning EBADMODE\n"); }
72696   #endif
72697                   reply_thr_get(udp_fd, EBADMODE, TRUE);
72698                   return NW_OK;
72699           }
72700
72701           /* NWUO_ACC_MASK */
72702           if (new_di_flags & NWUO_ACC_MASK)
72703           {
72704   #if DEBUG
72705    { where(); printf("returning EBADMODE\n"); }
72706   #endif
72707                   reply_thr_get(udp_fd, EBADMODE, TRUE);
72708                   return NW_OK;
72709                   /* access modes can't be disabled */
72710           }
72711
72712           if (!(new_en_flags & NWUO_ACC_MASK))
72713                   new_en_flags |= (old_en_flags & NWUO_ACC_MASK);
72714
72715           /* NWUO_LOCPORT_MASK */
72716           if (new_di_flags & NWUO_LOCPORT_MASK)
72717           {
72718   #if DEBUG
72719    { where(); printf("returning EBADMODE\n"); }
72720   #endif
72721                   reply_thr_get(udp_fd, EBADMODE, TRUE);
72722                   return NW_OK;
72723                   /* the loc ports can't be disabled */
72724           }
72725           if (!(new_en_flags & NWUO_LOCPORT_MASK))
72726           {
72727                   new_en_flags |= (old_en_flags & NWUO_LOCPORT_MASK);
72728                   newopt.nwuo_locport= oldopt.nwuo_locport;
72729           }
72730           else if ((new_en_flags & NWUO_LOCPORT_MASK) == NWUO_LP_SEL)
72731           {
72732                   newopt.nwuo_locport= find_unused_port(udp_fd-udp_fd_table);
72733           }
72734           else if ((new_en_flags & NWUO_LOCPORT_MASK) == NWUO_LP_SET)
```

```
72735           {
72736                   if (!newopt.nwuo_locport)
72737                   {
72738   #if DEBUG
72739    { where(); printf("returning EBADMODE\n"); }
72740   #endif
72741                           reply_thr_get(udp_fd, EBADMODE, TRUE);
72742                           return NW_OK;
72743                   }
72744           }
72745
72746           /* NWUO_LOCADDR_MASK */
72747           if (!((new_en_flags | new_di_flags) & NWUO_LOCADDR_MASK))
72748           {
72749                   new_en_flags |= (old_en_flags & NWUO_LOCADDR_MASK);
72750                   new_di_flags |= (old_di_flags & NWUO_LOCADDR_MASK);
72751           }
72752
72753           /* NWUO_BROAD_MASK */
72754           if (!((new_en_flags | new_di_flags) & NWUO_BROAD_MASK))
72755           {
72756                   new_en_flags |= (old_en_flags & NWUO_BROAD_MASK);
72757                   new_di_flags |= (old_di_flags & NWUO_BROAD_MASK);
72758           }
72759
72760           /* NWUO_REMPORT_MASK */
72761           if (!((new_en_flags | new_di_flags) & NWUO_REMPORT_MASK))
72762           {
72763                   new_en_flags |= (old_en_flags & NWUO_REMPORT_MASK);
72764                   new_di_flags |= (old_di_flags & NWUO_REMPORT_MASK);
72765                   newopt.nwuo_remport= oldopt.nwuo_remport;
72766           }
72767   #if DEBUG & 256
72768    { where(); printf("newopt.nwuo_remport= %d\n", ntohs(newopt.nwuo_remport)); }
72769   #endif
72770
72771           /* NWUO_REMADDR_MASK */
72772           if (!((new_en_flags | new_di_flags) & NWUO_REMADDR_MASK))
72773           {
72774                   new_en_flags |= (old_en_flags & NWUO_REMADDR_MASK);
72775                   new_di_flags |= (old_di_flags & NWUO_REMADDR_MASK);
72776                   newopt.nwuo_remaddr= oldopt.nwuo_remaddr;
72777           }
72778
72779           /* NWUO_RW_MASK */
72780           if (!((new_en_flags | new_di_flags) & NWUO_RW_MASK))
72781           {
72782                   new_en_flags |= (old_en_flags & NWUO_RW_MASK);
72783                   new_di_flags |= (old_di_flags & NWUO_RW_MASK);
72784           }
72785
72786           /* NWUO_IPOPT_MASK */
72787           if (!((new_en_flags | new_di_flags) & NWUO_IPOPT_MASK))
72788           {
72789                   new_en_flags |= (old_en_flags & NWUO_IPOPT_MASK);
72790                   new_di_flags |= (old_di_flags & NWUO_IPOPT_MASK);
72791           }
72792
72793           new_flags= ((unsigned long)new_di_flags << 16) | new_en_flags;
72794           if ((new_flags & NWUO_RWDATONLY) &&
```

```
72795                   ((new_flags & NWUO_LOCPORT_MASK) == NWUO_LP_ANY ||
72796                   (new_flags & (NWUO_RP_ANY|NWUO_RA_ANY|NWUO_EN_IPOPT))))
72797           {
72798   #if DEBUG
72799    { where(); printf("returning EBADMODE\n"); }
72800   #endif
72801                   reply_thr_get(udp_fd, EBADMODE, TRUE);
72802                   return NW_OK;
72803           }
72804
72805           /* Let's check the access modes */
72806           if ((new_flags & NWUO_LOCPORT_MASK) == NWUO_LP_SEL ||
72807                   (new_flags & NWUO_LOCPORT_MASK) == NWUO_LP_SET)
72808           {
72809                   for (i= 0, fd_ptr= udp_fd_table; i<UDP_FD_NR; i++, fd_ptr++)
72810                   {
72811                           if (fd_ptr == udp_fd)
72812                                   continue;
72813                           if (!(fd_ptr->uf_flags & UFF_INUSE))
72814                                   continue;
72815                           flags= fd_ptr->uf_udpopt.nwuo_flags;
72816                           if ((flags & NWUO_LOCPORT_MASK) != NWUO_LP_SEL &&
72817                                   (flags & NWUO_LOCPORT_MASK) != NWUO_LP_SET)
72818                                   continue;
72819                           if (fd_ptr->uf_udpopt.nwuo_locport !=
72820                                   newopt.nwuo_locport)
72821                                   continue;
72822                           if ((flags & NWUO_ACC_MASK) !=
72823                                   (new_flags & NWUO_ACC_MASK))
72824                           {
72825   #if DEBUG
72826    { where(); printf("address inuse: new fd= %d, old_fd= %d, port= %u\n",
72827           udp_fd-udp_fd_table, fd_ptr-udp_fd_table, newopt.nwuo_locport); }
72828   #endif
72829                                   reply_thr_get(udp_fd, EADDRINUSE, TRUE);
72830                                   return NW_OK;
72831                           }
72832                   }
72833           }
72834
72835           newopt.nwuo_flags= new_flags;
72836           udp_fd->uf_udpopt= newopt;
72837
72838           all_flags= new_en_flags | new_di_flags;
72839   #if DEBUG & 256
72840    { where();
72841           printf("NWUO_ACC_MASK: %s set\n", all_flags & NWUO_ACC_MASK ? "" : "not");
72842           printf("NWUO_LOCADDR_MASK: %s set\n", all_flags & NWUO_LOCADDR_MASK ? "" : "not");
72843           printf("NWUO_BROAD_MASK: %s set\n", all_flags & NWUO_BROAD_MASK ? "" : "not");
72844           printf("NWUO_REMPORT_MASK: %s set\n", all_flags & NWUO_REMPORT_MASK ? "" : "not");
72845           printf("NWUO_REMADDR_MASK: %s set\n", all_flags & NWUO_REMADDR_MASK ? "" : "not");
72846           printf("NWUO_RW_MASK: %s set\n", all_flags & NWUO_RW_MASK ? "" : "not");
72847           printf("NWUO_IPOPT_MASK: %s set\n", all_flags & NWUO_IPOPT_MASK ? "" : "not");
72848    }
72849   #endif
72850           if ((all_flags & NWUO_ACC_MASK) && (all_flags & NWUO_LOCPORT_MASK) &&
72851                   (all_flags & NWUO_LOCADDR_MASK) &&
72852                   (all_flags & NWUO_BROAD_MASK) &&
72853                   (all_flags & NWUO_REMPORT_MASK) &&
72854                   (all_flags & NWUO_REMADDR_MASK) &&
```

```
72855                   (all_flags & NWUO_RW_MASK) &&
72856                   (all_flags & NWUO_IPOPT_MASK))
72857                   udp_fd->uf_flags |= UFF_OPTSET;
72858           else
72859           {
72860                   udp_fd->uf_flags &= ~UFF_OPTSET;
72861           }
72862
72863           reply_thr_get(udp_fd, NW_OK, TRUE);
72864           return NW_OK;
72865   }

72867   PRIVATE udpport_t find_unused_port(fd)
72868   int fd;
72869   {
72870           udpport_t port, nw_port;
72871
72872           for (port= 0x8000; port < 0xffff-UDP_FD_NR; port+= UDP_FD_NR)
72873           {
72874                   nw_port= htons(port);
72875                   if (is_unused_port(nw_port))
72876                           return nw_port;
72877           }
72878           for (port= 0x8000; port < 0xffff; port++)
72879           {
72880                   nw_port= htons(port);
72881                   if (is_unused_port(nw_port))
72882                           return nw_port;
72883           }
72884           ip_panic(( "unable to find unused port (shouldn't occur)" ));
72885           return 0;
72886   }

72888   /*
72889   reply_thr_put
72890   */
72891
72892   PRIVATE int reply_thr_put(udp_fd, reply, for_ioctl)
72893   udp_fd_t *udp_fd;
72894   int reply;
72895   int for_ioctl;
72896   {
72897   #if DEBUG
72898    { where(); printf("reply_thr_put(&udp_fd_table[%d], %d, %d) called\n",
72899           udp_fd-udp_fd_table, reply, for_ioctl); }
72900   #endif
72901   #if DEBUG & 2
72902    { where(); printf("calling 0x%x\n", udp_fd->uf_put_userdata); }
72903   #endif
72904   assert (udp_fd);
72905           return (*udp_fd->uf_put_userdata)(udp_fd->uf_srfd, reply,
72906                   (acc_t *)0, for_ioctl);
72907   }

72909   /*
72910   reply_thr_get
72911   */
72912
72913   PRIVATE void reply_thr_get(udp_fd, reply, for_ioctl)
72914   udp_fd_t *udp_fd;
```

```
72915   int reply;
72916   int for_ioctl;
72917   {
72918           acc_t *result;
72919   #if DEBUG & 256
72920    { where(); printf("reply_thr_get(&udp_fd_table[%d], %d, %d) called\n",
72921           udp_fd-udp_fd_table, reply, for_ioctl); }
72922   #endif
72923           result= (*udp_fd->uf_get_userdata)(udp_fd->uf_srfd, reply,
72924                   (size_t)0, for_ioctl);
72925           assert (!result);
72926   }

72928   PRIVATE int is_unused_port(port)
72929   udpport_t port;
72930   {
72931           int i;
72932           udp_fd_t *udp_fd;
72933
72934           for (i= 0, udp_fd= udp_fd_table; i<UDP_FD_NR; i++,
72935                   udp_fd++)
72936           {
72937                   if (!(udp_fd->uf_flags & UFF_OPTSET))
72938                           continue;
72939                   if (udp_fd->uf_udpopt.nwuo_locport == port)
72940                           return FALSE;
72941           }
72942           return TRUE;
72943   }

72945   PRIVATE void read_ip_packets(udp_port)
72946   udp_port_t *udp_port;
72947   {
72948           int result;
72949
72950           do
72951           {
72952                   udp_port->up_flags |= UPF_READ_IP;
72953   #if DEBUG & 256
72954    { where(); printf("doing ip_read\n"); }
72955   #endif
72956                   result= ip_read(udp_port->up_ipfd, UDP_MAX_DATAGRAM);
72957                   if (result == NW_SUSPEND)
72958                   {
72959                           udp_port->up_flags |= UPF_READ_SP;
72960                           return;
72961                   }
72962   assert(result == NW_OK);
72963                   udp_port->up_flags &= ~UPF_READ_IP;
72964           } while(!(udp_port->up_flags & UPF_READ_IP));
72965   }


72968   PUBLIC int udp_read (fd, count)
72969   int fd;
72970   size_t count;
72971   {
72972           udp_fd_t *udp_fd;
72973
72974           udp_fd= &udp_fd_table[fd];
```

```
72975           if (!(udp_fd->uf_flags & UFF_OPTSET))
72976                   return reply_thr_put(udp_fd, EBADMODE, FALSE);
72977
72978           udp_fd->uf_rd_count= count;
72979
72980           if (udp_fd->uf_rd_buf)
72981           {
72982                   if (get_time() <= udp_fd->uf_exp_tim)
72983                           return udp_packet2user (udp_fd);
72984                   bf_afree(udp_fd->uf_rd_buf);
72985                   udp_fd->uf_rd_buf= 0;
72986           }
72987           udp_fd->uf_flags |= UFF_READ_IP;
72988   #if DEBUG & 256
72989    { where(); printf("udp_fd_table[%d].uf_flags= 0x%x\n",
72990           udp_fd-udp_fd_table, udp_fd->uf_flags); }
72991   #endif
72992           return NW_SUSPEND;
72993   }

72995   PRIVATE int udp_packet2user (udp_fd)
72996   udp_fd_t *udp_fd;
72997   {
72998           acc_t *pack, *tmp_pack;
72999           udp_io_hdr_t *hdr;
73000           int result, hdr_len;
73001           size_t size, transf_size;
73002
73003           pack= udp_fd->uf_rd_buf;
73004           udp_fd->uf_rd_buf= 0;
73005
73006           size= bf_bufsize (pack);
73007
73008           if (udp_fd->uf_udpopt.nwuo_flags & NWUO_RWDATONLY)
73009           {
73010
73011                   pack= bf_packIffLess (pack, UDP_IO_HDR_SIZE);
73012   assert (pack->acc_length >= UDP_IO_HDR_SIZE);
73013
73014                   hdr= (udp_io_hdr_t *)ptr2acc_data(pack);
73015                   hdr_len= UDP_IO_HDR_SIZE+hdr->uih_ip_opt_len;
73016
73017   assert (size>= hdr_len);
73018                   size -= hdr_len;
73019                   tmp_pack= bf_cut(pack, hdr_len, size);
73020                   bf_afree(pack);
73021                   pack= tmp_pack;
73022           }
73023
73024           if (size>udp_fd->uf_rd_count)
73025           {
73026                   tmp_pack= bf_cut (pack, 0, udp_fd->uf_rd_count);
73027                   bf_afree(pack);
73028                   pack= tmp_pack;
73029                   transf_size= udp_fd->uf_rd_count;
73030           }
73031           else
73032                   transf_size= size;
73033
73034           result= (*udp_fd->uf_put_userdata)(udp_fd->uf_srfd,
```

```
73035                   (size_t)0, pack, FALSE);
73036
73037           if (result >= 0)
73038                   if (size > transf_size)
73039                           result= EPACKSIZE;
73040                   else
73041                           result= transf_size;
73042
73043           udp_fd->uf_flags &= ~UFF_READ_IP;
73044           result= (*udp_fd->uf_put_userdata)(udp_fd->uf_srfd, result,
73045                           (acc_t *)0, FALSE);
73046   assert (result == 0);
73047
73048           return result;
73049   }

73051   PRIVATE void process_inc_fragm(udp_port, pack)
73052   udp_port_t *udp_port;
73053   acc_t *pack;
73054   {
73055           udp_fd_t *udp_fd, *share_fd;
73056           acc_t *ip_acc, *udp_acc, *ipopt_pack, *no_ipopt_pack, *tmp_acc;
73057           ip_hdr_t *ip_hdr;
73058           udp_hdr_t *udp_hdr;
73059           udp_io_hdr_t *udp_io_hdr;
73060           size_t pack_size, ip_hdr_size;
73061           size_t udp_size;
73062           ipaddr_t src_addr, dst_addr;
73063           u8_t u16[2];
73064           u16_t chksum;
73065           unsigned long dst_type, flags;
73066           time_t  exp_tim;
73067           udpport_t src_port, dst_port;
73068           int i;
73069
73070   #if DEBUG & 256
73071    { where(); printf("in process_inc_fragm\n"); }
73072   #endif
73073           pack_size= bf_bufsize(pack);
73074
73075           pack= bf_packIffLess(pack, IP_MIN_HDR_SIZE);
73076   assert (pack->acc_length >= IP_MIN_HDR_SIZE);
73077           ip_hdr= (ip_hdr_t *)ptr2acc_data(pack);
73078           ip_hdr_size= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
73079           ip_acc= bf_cut(pack, (size_t)0, ip_hdr_size);
73080
73081           src_addr= ip_hdr->ih_src;
73082           dst_addr= ip_hdr->ih_dst;
73083
73084           udp_acc= bf_cut(pack, ip_hdr_size, pack_size-ip_hdr_size);
73085           bf_afree(pack);
73086           pack_size -= ip_hdr_size;
73087           if (pack_size < UDP_HDR_SIZE)
73088           {
73089   #if DEBUG
73090    { where(); printf("packet too small\n"); }
73091   #endif
73092                   bf_afree(ip_acc);
73093                   bf_afree(udp_acc);
73094                   return;
```

```
73095           }
73096           udp_acc= bf_packIffLess(udp_acc, UDP_HDR_SIZE);
73097           udp_hdr= (udp_hdr_t *)ptr2acc_data(udp_acc);
73098
73099           udp_size= ntohs(udp_hdr->uh_length);
73100           if (udp_size > pack_size)
73101           {
73102   #if DEBUG
73103    { where(); printf("packet too large\n"); }
73104   #endif
73105                   bf_afree(ip_acc);
73106                   bf_afree(udp_acc);
73107                   return;
73108           }
73109           if (udp_hdr->uh_chksum)
73110           {
73111                   u16[0]= 0;
73112                   u16[1]= ip_hdr->ih_proto;
73113                   chksum= pack_oneCsum(udp_acc);
73114                   chksum= oneC_sum(chksum, (u16_t *)&src_addr, sizeof(ipaddr_t));
73115                   chksum= oneC_sum(chksum, (u16_t *)&dst_addr, sizeof(ipaddr_t));
73116                   chksum= oneC_sum(chksum, (u16_t *)u16, sizeof(u16));
73117                   chksum= oneC_sum(chksum, (u16_t *)&udp_hdr->uh_length,
73118                           sizeof(udp_hdr->uh_length));
73119                   if (~chksum & 0xffff)
73120                   {
73121   #if DEBUG
73122    { where(); printf("udp chksum error\n"); }
73123   #endif
73124                           bf_afree(ip_acc);
73125                           bf_afree(udp_acc);
73126                           return;
73127                   }
73128           }
73129           exp_tim= get_time() + UDP_READ_EXP_TIME;
73130           src_port= udp_hdr->uh_src_port;
73131           dst_port= udp_hdr->uh_dst_port;
73132
73133           if (dst_addr == udp_port->up_ipaddr)
73134                   dst_type= NWUO_EN_LOC;
73135           else
73136                   dst_type= NWUO_EN_BROAD;
73137
73138           share_fd= 0;
73139           ipopt_pack= 0;
73140           no_ipopt_pack= 0;
73141
73142           for (i=0, udp_fd=udp_fd_table; i<UDP_FD_NR; i++, udp_fd++)
73143           {
73144                   if (!(udp_fd->uf_flags & UFF_INUSE))
73145                   {
73146   #if DEBUG & 256
73147    { where(); printf("%d: not inuse\n", i); }
73148   #endif
73149                           continue;
73150                   }
73151                   if (!(udp_fd->uf_flags & UFF_OPTSET))
73152                   {
73153   #if DEBUG
73154    { where(); printf("%d: options not set\n", i); }
```

```
73155   #endif
73156                           continue;
73157                   }
73158                   flags= udp_fd->uf_udpopt.nwuo_flags;
73159                   if (!(flags & dst_type))
73160                   {
73161   #if DEBUG & 256
73162    { where(); printf("%d: wrong type\n", i); }
73163   #endif
73164                           continue;
73165                   }
73166                   if ((flags & (NWUO_LP_SEL|NWUO_LP_SET)) &&
73167                           udp_fd->uf_udpopt.nwuo_locport != dst_port)
73168                   {
73169   #if DEBUG & 256
73170    { where(); printf("%d: wrong loc port, got %d, expected %d\n", i,
73171           dst_port, udp_fd->uf_udpopt.nwuo_locport); }
73172   #endif
73173                           continue;
73174                   }
73175                   if ((flags & NWUO_RP_SET) &&
73176                           udp_fd->uf_udpopt.nwuo_remport != src_port)
73177                   {
73178   #if DEBUG
73179    { where(); printf("%d: wrong rem port, I got %d, expected %d\n", i,
73180           ntohs(src_port), ntohs(udp_fd->uf_udpopt.nwuo_remport)); }
73181   #endif
73182                           continue;
73183                   }
73184                   if ((flags & NWUO_RA_SET) &&
73185                           udp_fd->uf_udpopt.nwuo_remaddr != src_addr)
73186                   {
73187   #if DEBUG
73188    { where(); printf("%d: wrong rem addr\n", i); }
73189   #endif
73190                           continue;
73191                   }
73192
73193                   if (!no_ipopt_pack)
73194                   {
73195                           no_ipopt_pack= bf_memreq(UDP_IO_HDR_SIZE);
73196                           udp_io_hdr= (udp_io_hdr_t *)ptr2acc_data(no_ipopt_pack);
73197                           udp_io_hdr->uih_src_addr= src_addr;
73198                           udp_io_hdr->uih_dst_addr= dst_addr;
73199                           udp_io_hdr->uih_src_port= src_port;
73200                           udp_io_hdr->uih_dst_port= dst_port;
73201                           udp_io_hdr->uih_ip_opt_len= 0;
73202                           udp_io_hdr->uih_data_len= udp_size-UDP_HDR_SIZE;
73203                           no_ipopt_pack->acc_next= bf_cut(udp_acc,
73204                                   UDP_HDR_SIZE, udp_io_hdr->uih_data_len);
73205                           if (ip_hdr_size == IP_MIN_HDR_SIZE)
73206                           {
73207                                   ipopt_pack= no_ipopt_pack;
73208                                   ipopt_pack->acc_linkC++;
73209                           }
73210                           else
73211                                   ipopt_pack= 0;
73212                   }
73213                   if (flags & NWUO_EN_IPOPT)
73214                   {
```

```
73215                           if (!ipopt_pack)
73216                           {
73217                                   ipopt_pack= bf_memreq(UDP_IO_HDR_SIZE);
73218                                   *(udp_io_hdr_t *)ptr2acc_data(ipopt_pack)=
73219                                           *udp_io_hdr;
73220                                   udp_io_hdr= (udp_io_hdr_t *)
73221                                           ptr2acc_data(ipopt_pack);
73222                                   udp_io_hdr->uih_ip_opt_len= ip_hdr_size -
73223                                           IP_MIN_HDR_SIZE;
73224                                   ipopt_pack->acc_next= bf_cut(ip_acc,
73225                                           (size_t)IP_MIN_HDR_SIZE,
73226                                           (size_t)udp_io_hdr->uih_ip_opt_len);
73227                                   for (tmp_acc= ipopt_pack; tmp_acc->acc_next;
73228                                           tmp_acc= tmp_acc->acc_next);
73229   assert (tmp_acc->acc_linkC == 1);
73230                                   tmp_acc->acc_next= no_ipopt_pack->acc_next;
73231                                   if (tmp_acc->acc_next)
73232                                           tmp_acc->acc_next->acc_linkC++;
73233                           }
73234                           pack= ipopt_pack;
73235                   }
73236                   else
73237                           pack= no_ipopt_pack;
73238
73239                   if (udp_fd->uf_rd_buf)
73240                   {
73241                           if ((flags & NWUO_ACC_MASK) == NWUO_SHARED)
73242                           {
73243                                   share_fd= udp_fd;
73244                                   continue;
73245                           }
73246   #if DEBUG
73247    { where(); printf("throwing away packet\n"); }
73248   #endif
73249                           bf_afree(udp_fd->uf_rd_buf);
73250                   }
73251
73252                   udp_fd->uf_rd_buf= pack;
73253                   pack->acc_linkC++;
73254                   udp_fd->uf_exp_tim= exp_tim;
73255
73256                   if ((flags & NWUO_ACC_MASK) == NWUO_SHARED ||
73257                           (flags & NWUO_ACC_MASK) ==  NWUO_EXCL)
73258                   {
73259                           if (ipopt_pack)
73260                           {
73261                                   bf_afree(ipopt_pack);
73262                                   ipopt_pack= 0;
73263                           }
73264   assert(no_ipopt_pack);
73265                           bf_afree(no_ipopt_pack);
73266                           no_ipopt_pack= 0;
73267                   }
73268
73269                   if (udp_fd->uf_flags & UFF_READ_IP)
73270                   {
73271   #if DEBUG & 256
73272    { where(); printf("%d calling packet2user\n", i); }
73273   #endif
73274                           udp_packet2user(udp_fd);
```

```
73275                   }
73276                   else
73277                   {
73278   #if DEBUG & 256
73279    { where(); printf("%d not READ_IP\n", i); }
73280   #endif
73281                   }
73282                   if ((flags & NWUO_ACC_MASK) == NWUO_SHARED ||
73283                           (flags & NWUO_ACC_MASK) ==  NWUO_EXCL)
73284                   {
73285                           break;
73286                   }
73287           }
73288           if (share_fd && no_ipopt_pack)
73289           {
73290                   bf_afree(share_fd->uf_rd_buf);
73291                   if (share_fd->uf_udpopt.nwuo_flags & NWUO_EN_IPOPT)
73292                           pack= ipopt_pack;
73293                   else
73294                           pack= no_ipopt_pack;
73295                   pack->acc_linkC++;
73296                   share_fd->uf_rd_buf= pack;
73297                   share_fd->uf_exp_tim= exp_tim;
73298                   if (ipopt_pack)
73299                   {
73300                           bf_afree(ipopt_pack);
73301                           ipopt_pack= 0;
73302                   }
73303   assert (no_ipopt_pack);
73304                   bf_afree(no_ipopt_pack);
73305                   no_ipopt_pack= 0;
73306           }
73307           else
73308           {
73309                   if (ipopt_pack)
73310                           bf_afree(ipopt_pack);
73311                   if (no_ipopt_pack)
73312                           bf_afree(no_ipopt_pack);
73313           }
73314   assert (ip_acc);
73315           bf_afree(ip_acc);
73316   assert (udp_acc);
73317           bf_afree(udp_acc);
73318   }

73320   PUBLIC void udp_close(fd)
73321   int fd;
73322   {
73323           udp_fd_t *udp_fd;
73324
73325   #if DEBUG
73326    { where(); printf("udp_close (%d)\n", fd); }
73327   #endif
73328           udp_fd= &udp_fd_table[fd];
73329
73330           assert (udp_fd->uf_flags & UFF_INUSE);
73331
73332           udp_fd->uf_flags= UFF_EMPTY;
73333           if (udp_fd->uf_rd_buf)
73334           {
```

```
73335                   bf_afree(udp_fd->uf_rd_buf);
73336                   udp_fd->uf_rd_buf= 0;
73337           }
73338   }

73340   PUBLIC int udp_write(fd, count)
73341   int fd;
73342   size_t count;
73343   {
73344           udp_fd_t *udp_fd;
73345           udp_port_t *udp_port;
73346
73347           udp_fd= &udp_fd_table[fd];
73348           udp_port= udp_fd->uf_port;
73349
73350           if (!(udp_fd->uf_flags & UFF_OPTSET))
73351           {
73352                   reply_thr_get (udp_fd, EBADMODE, FALSE);
73353                   return NW_OK;
73354           }
73355
73356   assert (!(udp_fd->uf_flags & UFF_WRITE_IP));
73357
73358           udp_fd->uf_wr_count= count;
73359
73360           udp_fd->uf_flags |= UFF_WRITE_IP;
73361
73362           restart_write_fd(udp_fd);
73363
73364           if (udp_fd->uf_flags & UFF_WRITE_IP)
73365           {
73366   #if DEBUG
73367    { where(); printf("replying NW_SUSPEND\n"); }
73368   #endif
73369                   return NW_SUSPEND;
73370           }
73371           else
73372           {
73373   #if DEBUG & 256
73374    { where(); printf("replying NW_OK\n"); }
73375   #endif
73376                   return NW_OK;
73377           }
73378   }

73380   PRIVATE void restart_write_fd(udp_fd)
73381   udp_fd_t *udp_fd;
73382   {
73383           udp_port_t *udp_port;
73384           acc_t *pack, *ip_hdr_pack, *udp_hdr_pack, *ip_opt_pack, *user_data;
73385           udp_hdr_t *udp_hdr;
73386           udp_io_hdr_t *udp_io_hdr;
73387           ip_hdr_t *ip_hdr;
73388           size_t ip_opt_size, user_data_size;
73389           unsigned long flags;
73390           u16_t chksum;
73391           u8_t u16[2];
73392           int result;
73393
73394           udp_port= udp_fd->uf_port;
```

```
73395
73396           if (udp_port->up_flags & UPF_WRITE_IP)
73397           {
73398                   udp_port->up_flags |= UPF_MORE2WRITE;
73399   #if DEBUG
73400    { where(); printf("\n"); }
73401   #endif
73402                   return;
73403           }
73404
73405   assert (udp_fd->uf_flags & UFF_WRITE_IP);
73406           udp_fd->uf_flags &= ~UFF_WRITE_IP;
73407
73408   assert (!udp_port->up_wr_pack);
73409
73410           pack= (*udp_fd->uf_get_userdata)(udp_fd->uf_srfd, 0,
73411                   udp_fd->uf_wr_count, FALSE);
73412           if (!pack)
73413           {
73414                   udp_fd->uf_flags &= ~UFF_WRITE_IP;
73415                   reply_thr_get (udp_fd, EFAULT, FALSE);
73416   #if DEBUG
73417    { where(); printf("\n"); }
73418   #endif
73419                   return;
73420           }
73421
73422           flags= udp_fd->uf_udpopt.nwuo_flags;
73423
73424           ip_hdr_pack= bf_memreq(IP_MIN_HDR_SIZE);
73425           ip_hdr= (ip_hdr_t *)ptr2acc_data(ip_hdr_pack);
73426
73427           udp_hdr_pack= bf_memreq(UDP_HDR_SIZE);
73428           udp_hdr= (udp_hdr_t *)ptr2acc_data(udp_hdr_pack);
73429
73430           if (flags & NWUO_RWDATALL)
73431           {
73432                   pack= bf_packIffLess(pack, UDP_IO_HDR_SIZE);
73433                   udp_io_hdr= (udp_io_hdr_t *)ptr2acc_data(pack);
73434                   ip_opt_size= udp_io_hdr->uih_ip_opt_len;
73435                   user_data_size= udp_io_hdr->uih_data_len;
73436                   if (UDP_IO_HDR_SIZE+ip_opt_size>udp_fd->uf_wr_count)
73437                   {
73438                           bf_afree(ip_hdr_pack);
73439                           bf_afree(udp_hdr_pack);
73440                           bf_afree(pack);
73441                           reply_thr_get (udp_fd, EINVAL, FALSE);
73442   #if DEBUG
73443    { where(); printf("\n"); }
73444   #endif
73445                           return;
73446                   }
73447                   if (ip_opt_size & 3)
73448                   {
73449                           bf_afree(ip_hdr_pack);
73450                           bf_afree(udp_hdr_pack);
73451                           bf_afree(pack);
73452                           reply_thr_get (udp_fd, EFAULT, FALSE);
73453   #if DEBUG
73454    { where(); printf("\n"); }
```

```
73455   #endif
73456                           return;
73457                   }
73458                   if (ip_opt_size)
73459                           ip_opt_pack= bf_cut(pack, UDP_IO_HDR_SIZE, ip_opt_size);
73460                   else
73461                           ip_opt_pack= 0;
73462                   user_data_size= udp_fd->uf_wr_count-UDP_IO_HDR_SIZE-
73463                           ip_opt_size;
73464                   user_data= bf_cut(pack, UDP_IO_HDR_SIZE+ip_opt_size,
73465                           user_data_size);
73466                   bf_afree(pack);
73467           }
73468           else
73469           {
73470                   udp_io_hdr= 0;
73471                   ip_opt_size= 0;
73472                   user_data_size= udp_fd->uf_wr_count;
73473                   ip_opt_pack= 0;
73474                   user_data= pack;
73475           }
73476
73477           ip_hdr->ih_vers_ihl= (IP_MIN_HDR_SIZE+ip_opt_size) >> 2;
73478           ip_hdr->ih_tos= UDP_TOS;
73479           ip_hdr->ih_flags_fragoff= HTONS(UDP_IP_FLAGS);
73480           ip_hdr->ih_ttl= UDP_TTL;
73481           ip_hdr->ih_proto= IPPROTO_UDP;
73482           if (flags & NWUO_RA_SET)
73483           {
73484   #if DEBUG
73485            { where(); printf("NWUO_RA_SET\n"); }
73486   #endif
73487                   ip_hdr->ih_dst= udp_fd->uf_udpopt.nwuo_remaddr;
73488           }
73489           else
73490           {
73491   assert (udp_io_hdr);
73492                   ip_hdr->ih_dst= udp_io_hdr->uih_dst_addr;
73493           }
73494   #if DEBUG & 256
73495            { where(); printf("ih_dst= "); writeIpAddr(ip_hdr->ih_dst); printf("\n"); }
73496   #endif
73497
73498           if ((flags & NWUO_LOCPORT_MASK) != NWUO_LP_ANY)
73499                   udp_hdr->uh_src_port= udp_fd->uf_udpopt.nwuo_locport;
73500           else
73501           {
73502   assert (udp_io_hdr);
73503                   udp_hdr->uh_src_port= udp_io_hdr->uih_src_port;
73504           }
73505
73506           if (flags & NWUO_RP_SET)
73507                   udp_hdr->uh_dst_port= udp_fd->uf_udpopt.nwuo_remport;
73508           else
73509           {
73510   assert (udp_io_hdr);
73511                   udp_hdr->uh_dst_port= udp_io_hdr->uih_dst_port;
73512           }
73513
73514           udp_hdr->uh_length= htons(UDP_HDR_SIZE+user_data_size);
```

```
73515           udp_hdr->uh_chksum= 0;
73516
73517           udp_hdr_pack->acc_next= user_data;
73518           chksum= pack_oneCsum(udp_hdr_pack);
73519           chksum= oneC_sum(chksum, (u16_t *)&udp_fd->uf_port->up_ipaddr,
73520                   sizeof(ipaddr_t));
73521           chksum= oneC_sum(chksum, (u16_t *)&ip_hdr->ih_dst, sizeof(ipaddr_t));
73522           u16[0]= 0;
73523           u16[1]= IPPROTO_UDP;
73524           chksum= oneC_sum(chksum, (u16_t *)u16, sizeof(u16));
73525           chksum= oneC_sum(chksum, (u16_t *)&udp_hdr->uh_length, sizeof(u16_t));
73526           if (~chksum)
73527                   chksum= ~chksum;
73528           udp_hdr->uh_chksum= chksum;
73529
73530           if (ip_opt_pack)
73531           {
73532                   ip_opt_pack= bf_packIffLess(ip_opt_pack, ip_opt_size);
73533                   ip_opt_pack->acc_next= udp_hdr_pack;
73534                   udp_hdr_pack= ip_opt_pack;
73535           }
73536           ip_hdr_pack->acc_next= udp_hdr_pack;
73537
73538   assert (!udp_port->up_wr_pack);
73539   assert (!(udp_port->up_flags & UPF_WRITE_IP));
73540
73541           udp_port->up_wr_pack= ip_hdr_pack;
73542           udp_port->up_flags |= UPF_WRITE_IP;
73543   #if DEBUG & 256
73544    { where(); printf("calling ip_write(%d, %d)\n", udp_port->up_ipfd,
73545           bf_bufsize(ip_hdr_pack)); }
73546   #endif
73547           result= ip_write(udp_port->up_ipfd, bf_bufsize(ip_hdr_pack));
73548   #if DEBUG & 256
73549    { where(); printf("ip_write done\n"); }
73550   #endif
73551           if (result == NW_SUSPEND)
73552           {
73553                   udp_port->up_flags |= UPF_WRITE_SP;
73554                   udp_fd->uf_flags |= UFF_WRITE_IP;
73555                   udp_port->up_write_fd= udp_fd;
73556           }
73557           else if (result<0)
73558                   reply_thr_get(udp_fd, result, FALSE);
73559           else
73560                   reply_thr_get (udp_fd, udp_fd->uf_wr_count, FALSE);
73561   #if DEBUG & 256
73562    { where(); printf("\n"); }
73563   #endif
73564   }

73566   PRIVATE u16_t pack_oneCsum(pack)
73567   acc_t *pack;
73568   {
73569           u16_t prev;
73570           int odd_byte;
73571           char *data_ptr;
73572           int length;
73573           char byte_buf[2];
73574
```

```
73575           assert (pack);
73576
73577           prev= 0;
73578
73579           odd_byte= FALSE;
73580           for (; pack; pack= pack->acc_next)
73581           {
73582
73583                   data_ptr= ptr2acc_data(pack);
73584                   length= pack->acc_length;
73585
73586                   if (!length)
73587                           continue;
73588                   if (odd_byte)
73589                   {
73590                           byte_buf[1]= *data_ptr;
73591                           prev= oneC_sum(prev, (u16_t *)byte_buf, 2);
73592                           data_ptr++;
73593                           length--;
73594                           odd_byte= FALSE;
73595                   }
73596                   if (length & 1)
73597                   {
73598                           odd_byte= TRUE;
73599                           length--;
73600                           byte_buf[0]= data_ptr[length];
73601                   }
73602                   if (!length)
73603                           continue;
73604                   prev= oneC_sum (prev, (u16_t *)data_ptr, length);
73605           }
73606           if (odd_byte)
73607                   prev= oneC_sum (prev, (u16_t *)byte_buf, 1);
73608           return prev;
73609   }

73611   PRIVATE void udp_restart_write_port(udp_port )
73612   udp_port_t *udp_port;
73613   {
73614           udp_fd_t *udp_fd;
73615           int i;
73616
73617   assert (!udp_port->up_wr_pack);
73618   assert (!(udp_port->up_flags & (UPF_WRITE_IP|UPF_WRITE_SP)));
73619
73620           while (udp_port->up_flags & UPF_MORE2WRITE)
73621           {
73622                   udp_port->up_flags &= ~UPF_MORE2WRITE;
73623
73624                   for (i= 0, udp_fd= udp_port->up_next_fd; i<UDP_FD_NR;
73625                           i++, udp_fd++)
73626                   {
73627                           if (udp_fd == &udp_fd_table[UDP_FD_NR])
73628                                   udp_fd= udp_fd_table;
73629
73630                           if (!(udp_fd->uf_flags & UFF_INUSE))
73631                                   continue;
73632                           if (!(udp_fd->uf_flags & UFF_WRITE_IP))
73633                                   continue;
73634                           if (udp_fd->uf_port != udp_port)
```

```
73635                                   continue;
73636                           restart_write_fd(udp_fd);
73637                           if (udp_port->up_flags & UPF_WRITE_IP)
73638                           {
73639                                   udp_port->up_next_fd= udp_fd+1;
73640                                   udp_port->up_flags |= UPF_MORE2WRITE;
73641                                   return;
73642                           }
73643                   }
73644           }
73645   }

73647   PUBLIC int udp_cancel(fd, which_operation)
73648   int fd;
73649   int which_operation;
73650   {
73651           udp_fd_t *udp_fd;
73652
73653   #if DEBUG
73654    { where(); printf("udp_cancel(%d, %d)\n", fd, which_operation); }
73655   #endif
73656           udp_fd= &udp_fd_table[fd];
73657
73658           switch (which_operation)
73659           {
73660           case SR_CANCEL_READ:
73661   assert (udp_fd->uf_flags & UFF_READ_IP);
73662                   udp_fd->uf_flags &= ~UFF_READ_IP;
73663                   reply_thr_put(udp_fd, EINTR, FALSE);
73664                   break;
73665           case SR_CANCEL_WRITE:
73666   assert (udp_fd->uf_flags & UFF_WRITE_IP);
73667                   udp_fd->uf_flags &= ~UFF_WRITE_IP;
73668                   if (udp_fd->uf_port->up_write_fd == udp_fd)
73669                           udp_fd->uf_port->up_write_fd= NULL;
73670                   reply_thr_get(udp_fd, EINTR, FALSE);
73671                   break;
73672           case SR_CANCEL_IOCTL:
73673   assert (udp_fd->uf_flags & UFF_IOCTL_IP);
73674                   udp_fd->uf_flags &= ~UFF_IOCTL_IP;
73675                   reply_thr_get(udp_fd, EINTR, TRUE);
73676                   break;
73677           default:
73678                   ip_panic(( "got unknown cancel request" ));
73679           }
73680           return NW_OK;
73681   }

73683   PRIVATE void udp_buffree (priority, reqsize)
73684   int priority;
73685   size_t reqsize;
73686   {
73687           int i;
73688           time_t curr_tim;
73689
73690
73691           if (priority <UDP_PRI_EXP_FDBUFS)
73692                   return;
73693
73694           curr_tim= get_time();
```

```
73695           for (i=0; i<UDP_FD_NR; i++)
73696           {
73697                   if (!(udp_fd_table[i].uf_flags & UFF_INUSE) )
73698                           continue;
73699                   if (udp_fd_table[i].uf_rd_buf &&
73700                           udp_fd_table[i].uf_exp_tim < curr_tim)
73701                   {
73702                           bf_afree(udp_fd_table[i].uf_rd_buf);
73703                           udp_fd_table[i].uf_rd_buf= 0;
73704                           if (bf_free_buffsize >= reqsize)
73705                                   return;
73706                   }
73707           }
73708
73709           if (priority <UDP_PRI_FDBUFS)
73710                   return;
73711
73712           for (i=0; i<UDP_FD_NR; i++)
73713           {
73714                   if (!(udp_fd_table[i].uf_flags & UFF_INUSE))
73715                           continue;
73716                   if (udp_fd_table[i].uf_rd_buf)
73717                   {
73718                           bf_afree(udp_fd_table[i].uf_rd_buf);
73719                           udp_fd_table[i].uf_rd_buf= 0;
73720                           if (bf_free_buffsize >= reqsize)
73721                                   return;
73722                   }
73723           }
73724   }
```