

```
# Makefile for all system servers.
```

```
#
```

```
MAKE = exec make -$(MAKEFLAGS)
```

usage:

```
@echo " " >&2
@echo "Makefile for all system servers." >&2
@echo "Usage:" >&2
@echo " make build   # Compile all system servers locally" >&2
@echo " make image   # Compile servers in boot image" >&2
@echo " make clean    # Remove local compiler results" >&2
@echo " make install  # Install servers to /etc/servers/" >&2
@echo "                (requires root privileges)" >&2
@echo " " >&2
```

build: all

```
all install depend clean:
```

```
cd ./pm && $(MAKE) $@
cd ./fs && $(MAKE) $@
cd ./rs && $(MAKE) $@
cd ./ds && $(MAKE) $@
cd ./is && $(MAKE) $@
cd ./init && $(MAKE) $@
cd ./inet && $(MAKE) $@
```

image:

```
cd ./pm && $(MAKE) EXTRA_OPTS=$(EXTRA_OPTS) build
cd ./fs && $(MAKE) EXTRA_OPTS=$(EXTRA_OPTS) build
cd ./rs && $(MAKE) EXTRA_OPTS=$(EXTRA_OPTS) build
cd ./ds && $(MAKE) EXTRA_OPTS=$(EXTRA_OPTS) build
cd ./init && $(MAKE) EXTRA_OPTS=$(EXTRA_OPTS) build
```

```
# Makefile for Data Store Server (DS)
SERVER = ds

# directories
u = /usr
i = $u/include
s = $i/sys
m = $i/minix
b = $i/ibm
k = $u/src/kernel
p = $u/src/servers/pm
f = $u/src/servers/fs

# programs, flags, etc.
CC =      exec cc
CFLAGS = -I$i
LDFLAGS = -i
LIBS = -lsys -lsysutil

OBJ = main.o store.o

# build local binary
all build:      $(SERVER)
$(SERVER):      $(OBJ)
                $(CC) -o $@ $(LDFLAGS) $(OBJ) $(LIBS)
                install -S 16k $@

# install with other servers
install: $(SERVER)
            install -o root -c $? /sbin/$(SERVER)
#           install -o root -cs $? $@

# clean up local files
clean:
            rm -f $(SERVER) *.o *.bak

depend:
            /usr/bin/mkdep "$(CC) -E $(CPPFLAGS)" *.c > .depend

# Include generated dependencies.
include .depend
```

```
/* Global variables. */

/* The parameters of the call are kept here. */
extern int who; /* caller's proc number */
extern int callnr; /* system call number */
extern int dont_reply; /* normally 0; set to 1 to inhibit reply */
```

```
/* Header file including all needed system headers. */

#define _SYSTEM 1 /* get OK and negative error codes */
#define _MINIX 1 /* tell headers to include MINIX stuff */

#include <ansi.h>
#include <sys/types.h>
#include <limits.h>
#include <errno.h>

#include <minix/callnr.h>
#include <minix/config.h>
#include <minix/type.h>
#include <minix/const.h>
#include <minix/com.h>
#include <minix/syslib.h>
#include <minix/sysutil.h>
#include <minix/keymap.h>
#include <minix/bitmap.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>

#include "proto.h"
#include "glo.h"
#include "store.h"
```

```

/* Data Store Server.
 * This service implements a little publish/subscribe data store that is
 * crucial for the system's fault tolerance. Components that require state
 * can store it here, for later retrieval, e.g., after a crash and subsequent
 * restart by the reincarnation server.
 *
 * Created:
 *   Oct 19, 2005          by Jorrit N. Herder
 */

#include "inc.h"          /* include master header file */

/* Allocate space for the global variables. */
int who_e;               /* caller's proc number */
int callnr;              /* system call number */
int sys_panic;           /* flag to indicate system-wide panic */

extern int errno;        /* error number set by system library */

/* Declare some local functions. */
FORWARD _PROTOTYPE(void init_server, (int argc, char **argv)           );
FORWARD _PROTOTYPE(void exit_server, (void)                           );
FORWARD _PROTOTYPE(void sig_handler, (void)                           );
FORWARD _PROTOTYPE(void get_work, (message *m_ptr)                     );
FORWARD _PROTOTYPE(void reply, (int whom, message *m_ptr)             );

/*=====
 *                               main
 *=====*/
PUBLIC int main(int argc, char **argv)
{
/* This is the main routine of this service. The main loop consists of
 * three major activities: getting new work, processing the work, and
 * sending the reply. The loop never terminates, unless a panic occurs.
 */
    message m;
    int result;
    sigset_t sigset;

    /* Initialize the server, then go to work. */
    init_server(argc, argv);

    /* Main loop - get work and do it, forever. */
    while (TRUE) {

        /* Wait for incoming message, sets 'callnr' and 'who'. */
        get_work(&m);

        switch (callnr) {
        case PROC_EVENT:
            sig_handler();
            continue;
        case DS_PUBLISH:
            result = do_publish(&m);
            break;
        case DS_RETRIEVE:
            result = do_retrieve(&m);
            break;
        case DS_SUBSCRIBE:
            result = do_subscribe(&m);
            break;
        case GETSYSINFO:
            result = do_getsysinfo(&m);
            break;
        default:
            report("DS", "warning, got illegal request from:", m.m_source);
            result = EINVAL;
        }

        /* Finally send reply message, unless disabled. */
        if (result != EDONTREPLY) {
            m.m_type = result;          /* build reply message */
            reply(who_e, &m);          /* send it away */
        }
    }
}

```

```

}
return(OK);                                /* shouldn't come here */
}

/*=====
*                               init_server                               *
*=====*/
PRIVATE void init_server(int argc, char **argv)
{
    /* Initialize the data store server. */
    int i, s;
    struct sigaction sigact;

    /* Install signal handler. Ask PM to transform signal into message. */
    sigact.sa_handler = SIG_MESS;
    sigact.sa_mask = ~0;                    /* block all other signals */
    sigact.sa_flags = 0;                    /* default behaviour */
    if (sigaction(SIGTERM, &sigact, NULL) < 0)
        report("DS", "warning, sigaction() failed", errno);
}

/*=====
*                               sig_handler                               *
*=====*/
PRIVATE void sig_handler()
{
    /* Signal handler. */
    sigset_t sigset;
    int sig;

    /* Try to obtain signal set from PM. */
    if (getsigset(&sigset) != 0) return;

    /* Check for known signals. */
    if (sigismember(&sigset, SIGTERM)) {
        exit_server();
    }
}

/*=====
*                               exit_server                               *
*=====*/
PRIVATE void exit_server()
{
    /* Shut down the information service. */

    /* Done. Now exit. */
    exit(0);
}

/*=====
*                               get_work                                  *
*=====*/
PRIVATE void get_work(m_ptr)
message *m_ptr;                            /* message buffer */
{
    int status = 0;
    status = receive(ANY, m_ptr);           /* this blocks until message arrives */
    if (OK != status)
        panic("DS", "failed to receive message!", status);
    who_e = m_ptr->m_source;                /* message arrived! set sender */
    callnr = m_ptr->m_type;                 /* set function call number */
}

/*=====
*                               reply                                    *
*=====*/
PRIVATE void reply(who_e, m_ptr)
int who_e;                                /* destination */
message *m_ptr;                           /* message buffer */
{
    int s;
    s = send(who_e, m_ptr);                /* send the message */
    if (OK != s)

```

```
}    panic( "DS", "unable to send reply!", s );
```

```
/* Function prototypes. */

/* main.c */
_PROTOTYPE(int main, (int argc, char **argv));

/* store.c */
_PROTOTYPE(int do_publish, (message *m_ptr));
_PROTOTYPE(int do_retrieve, (message *m_ptr));
_PROTOTYPE(int do_subscribe, (message *m_ptr));
_PROTOTYPE(int do_getsysinfo, (message *m_ptr));
```



```
/* Implementation of the Data Store. */
```

```
#include "inc.h"
```

```
/* Allocate space for the data store. */
```

```
PRIVATE struct data_store ds_store[NR_DS_KEYS];
```

```
PRIVATE int nr_in_use;
```

```
PRIVATE _PROTOTYPE(int find_key, (int key, struct data_store **dsp));
```

```
PRIVATE _PROTOTYPE(int set_owner, (struct data_store *dsp, void *auth_ptr));
```

```
PRIVATE _PROTOTYPE(int is_authorized, (struct data_store *dsp, void *auth_ptr));
```

```
PRIVATE int set_owner(dsp, ap)
```

```
struct data_store *dsp;
```

```
/* data store structure */
```

```
void *ap;
```

```
/* authorization pointer */
```

```
{
    /* Authorize the caller. */
    return(TRUE);
}
```

```
PRIVATE int is_authorized(dsp, ap)
```

```
struct data_store *dsp;
```

```
/* data store structure */
```

```
void *ap;
```

```
/* authorization pointer */
```

```
{
    /* Authorize the caller. */
    return(TRUE);
}
```

```
PRIVATE int find_key(key, dsp)
```

```
int key;
```

```
/* key to look up */
```

```
struct data_store **dsp;
```

```
/* store pointer here */
```

```
{
    register int i;

    *dsp = NULL;
    for (i=0; i<NR_DS_KEYS; i++) {
        if ((ds_store[i].ds_flags & DS_IN_USE) && ds_store[i].ds_key == key) {
            *dsp = &ds_store[i];
            return(TRUE);
        }
    }
    return(FALSE);
}
```

```
/* report success */
```

```
/* report not found */
```

```
PUBLIC int do_publish(m_ptr)
```

```
message *m_ptr;
```

```
/* request message */
```

```
{
    struct data_store *dsp;

    /* Store (key,value)-pair. First see if key already exists. If so,
     * check if the caller is allowed to overwrite the value. Otherwise
     * find a new slot and store the new value.
     */
    if (find_key(m_ptr->DS_KEY, &dsp)) {
        /* look up key */
        if (! is_authorized(dsp, m_ptr->DS_AUTH)) {
            /* check if owner */
            return(EPERM);
        }
    }
    else {
        /* find a new slot */
        if (nr_in_use >= NR_DS_KEYS) {
            return(EAGAIN);
        }
        else {
            dsp = &ds_store[nr_in_use];
            /* new slot found */
            dsp->ds_key = m_ptr->DS_KEY;
            if (! set_owner(dsp, m_ptr->DS_AUTH)) {
                /* associate owner */
                return(EINVAL);
            }
            dsp->ds_nr_subs = 0;
            /* nr of subscribers */
            dsp->ds_flags = DS_IN_USE;
            /* initialize slot */
            nr_in_use ++;
        }
    }
}
```

```

    }
}

/* At this point we have a data store pointer and know the caller is
 * authorized to write to it. Set all fields as requested.
 */
dsp->ds_val_l1 = m_ptr->DS_VAL_L1;          /* store all data */
dsp->ds_val_l2 = m_ptr->DS_VAL_L2;

/* If the data is public. Check if there are any subscribers to this key.
 * If so, notify all subscribers so that they can retrieve the data, if
 * they're still interested.
 */
if ((dsp->ds_flags & DS_PUBLIC) && dsp->ds_nr_subs > 0) {
    /* Subscriptions are not yet implemented. */
}

return(OK);
}

PUBLIC int do_retrieve(m_ptr)
message *m_ptr;                                /* request message */
{
    struct data_store *dsp;

    /* Retrieve data. Look up the key in the data store. Return an error if it
     * is not found. If this data is private, only the owner may retrieve it.
     */
    if (find_key(m_ptr->DS_KEY, &dsp)) {          /* look up key */

        /* If the data is not public, the caller must be authorized. */
        if (! dsp->ds_flags & DS_PUBLIC) {        /* check if private */
            if (! is_authorized(dsp, m_ptr->DS_AUTH)) { /* authorize call */
                return(EPERM);                    /* not allowed */
            }
        }

        /* Data is public or the caller is authorized to retrieve it. */
        printf("DS retrieves data: key %d (found %d), l1 %u, l2 %u\n",
               m_ptr->DS_KEY, dsp->ds_key, dsp->ds_val_l1, dsp->ds_val_l2);
        m_ptr->DS_VAL_L1 = dsp->ds_val_l1;        /* return value */
        m_ptr->DS_VAL_L2 = dsp->ds_val_l2;        /* return value */
        return(OK);                              /* report success */
    }
    return(ESRCH);                                /* key not found */
}

PUBLIC int do_subscribe(m_ptr)
message *m_ptr;                                /* request message */
{
    /* Subscribe to a key of interest. Only existing and public keys can be
     * subscribed to. All updates to the key will cause a notification message
     * to be sent to the subscribed. On success, directly return a copy of the
     * data for the given key.
     */
    return(ENOSYS);
}

/*=====
 *                               do_getsysinfo                               *
 *=====*/
PUBLIC int do_getsysinfo(m_ptr)
message *m_ptr;
{
    vir_bytes src_addr, dst_addr;
    int dst_proc;
    size_t len;
    int s;

    switch(m_ptr->m1_i1) {

```

```
case SI_DATA_STORE:
    src_addr = (vir_bytes) ds_store;
    len = sizeof(struct data_store) * NR_DS_KEYS;
    break;
default:
    return(EINVAL);
}

dst_proc = m_ptr->m_source;
dst_addr = (vir_bytes) m_ptr->m1_p1;
if (OK != (s=sys_datacopy(SELF, src_addr, dst_proc, dst_addr, len)))
    return(s);
return(OK);
}
```

```
/* Type definitions for the Data Store Server. */
struct data_store {
    int ds_flags;           /* flags for this store */
    int ds_key;             /* key to lookup information */
    long ds_val_l1;         /* data associated with key */
    long ds_val_l2;         /* data associated with key */
    long ds_auth;           /* secret given by owner of data */
    int ds_nr_subs;         /* number of subscribers for key */
};

/* Flag values. */
#define DS_IN_USE          0x01
#define DS_PUBLIC          0x02

/* Constants for the Data Store Server. */
#define NR_DS_KEYS         64    /* reserve space for so many items */
```

```
# Makefile for File System (FS)
SERVER = fs

# directories
u = /usr
i = $u/include
s = $i/sys
h = $i/minix

# programs, flags, etc.
CC =      exec cc
CFLAGS = -I$i $(EXTRA_OPTS)
LDFLAGS = -i
LIBS = -lsys -lsysutil -ltimers

OBJ =  main.o open.o read.o write.o pipe.o dmap.o \
       device.o exec.o path.o mount.o link.o super.o inode.o \
       cache.o cache2.o filedес.o stadir.o protect.o time.o \
       lock.o misc.o utility.o select.o timers.o table.o

# build local binary
install all build:      $(SERVER)
$(SERVER):             $(OBJ)
                       $(CC) -o $@ $(LDFLAGS) $(OBJ) $(LIBS)
                       install -S 512w $@

# clean up local files
clean:
      rm -f $(SERVER) *.o *.bak

depend:
      /usr/bin/mkdep "$(CC) -E $(CPPFLAGS)" *.c > .depend

# Include generated dependencies.
include .depend
```

```

/* Buffer (block) cache. To acquire a block, a routine calls get_block(),
 * telling which block it wants. The block is then regarded as "in use"
 * and has its 'b_count' field incremented. All the blocks that are not
 * in use are chained together in an LRU list, with 'front' pointing
 * to the least recently used block, and 'rear' to the most recently used
 * block. A reverse chain, using the field b_prev is also maintained.
 * Usage for LRU is measured by the time the put_block() is done. The second
 * parameter to put_block() can violate the LRU order and put a block on the
 * front of the list, if it will probably not be needed soon. If a block
 * is modified, the modifying routine must set b_dirt to DIRTY, so the block
 * will eventually be rewritten to the disk.
 */

#include <sys/dir.h>                /* need struct direct */
#include <dirent.h>

EXTERN struct buf {
    /* Data portion of the buffer. */
    union {
        char b__data[_MAX_BLOCK_SIZE];          /* ordinary user data */
/* directory block */
        struct direct b__dir[NR_DIR_ENTRIES(_MAX_BLOCK_SIZE)];
/* V1 indirect block */
        zone1_t b__v1_ind[V1_INDIRECTS];
/* V2 indirect block */
        zone_t b__v2_ind[V2_INDIRECTS(_MAX_BLOCK_SIZE)];
/* V1 inode block */
        d1_inode b__v1_ino[V1_INODES_PER_BLOCK];
/* V2 inode block */
        d2_inode b__v2_ino[V2_INODES_PER_BLOCK(_MAX_BLOCK_SIZE)];
/* bit map block */
        bitchunk_t b__bitmap[FS_BITMAP_CHUNKS(_MAX_BLOCK_SIZE)];
    } b;

    /* Header portion of the buffer. */
    struct buf *b_next;          /* used to link all free bufs in a chain */
    struct buf *b_prev;         /* used to link all free bufs the other way */
    struct buf *b_hash;         /* used to link bufs on hash chains */
    block_t b_blocknr;          /* block number of its (minor) device */
    dev_t b_dev;                /* major | minor device where block resides */
    char b_dirt;                /* CLEAN or DIRTY */
    char b_count;               /* number of users of this buffer */
} buf[NR_BUFS];

/* A block is free if b_dev == NO_DEV. */

#define NIL_BUF ((struct buf *) 0)    /* indicates absence of a buffer */

/* These defs make it possible to use to bp->b_data instead of bp->b.b__data */
#define b_data      b.b__data
#define b_dir       b.b__dir
#define b_v1_ind    b.b__v1_ind
#define b_v2_ind    b.b__v2_ind
#define b_v1_ino    b.b__v1_ino
#define b_v2_ino    b.b__v2_ino
#define b_bitmap    b.b__bitmap

EXTERN struct buf *buf_hash[NR_BUF_HASH];    /* the buffer hash table */

EXTERN struct buf *front;    /* points to least recently used free block */
EXTERN struct buf *rear;    /* points to most recently used free block */
EXTERN int bufs_in_use;    /* # bufs currently in use (not on free list) */

/* When a block is released, the type of usage is passed to put_block(). */
#define WRITE_IMMED    0100 /* block should be written to disk now */
#define ONE_SHOT       0200 /* set if block not likely to be needed soon */

#define INODE_BLOCK    0          /* inode block */
#define DIRECTORY_BLOCK 1        /* directory block */
#define INDIRECT_BLOCK 2        /* pointer block */
#define MAP_BLOCK      3        /* bit map */
#define FULL_DATA_BLOCK 5        /* data, fully used */
#define PARTIAL_DATA_BLOCK 6    /* data, partly used */

```

```
#define HASH_MASK (NR_BUF_HASH - 1)      /* mask for hashing block numbers */
```

```

/* The file system maintains a buffer cache to reduce the number of disk
 * accesses needed. Whenever a read or write to the disk is done, a check is
 * first made to see if the block is in the cache. This file manages the
 * cache.
 *
 * The entry points into this file are:
 *   get_block:   request to fetch a block for reading or writing from cache
 *   put_block:   return a block previously requested with get_block
 *   alloc_zone:  allocate a new zone (to increase the length of a file)
 *   free_zone:   release a zone (when a file is removed)
 *   invalidate:  remove all the cache blocks on some device
 *
 * Private functions:
 *   rw_block:    read or write a block from the disk itself
 */

#include "fs.h"
#include <minix/com.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "super.h"

FORWARD _PROTOTYPE( void rm_lru, (struct buf *bp) );
FORWARD _PROTOTYPE( int rw_block, (struct buf *, int) );

/*=====
 *                               get_block                               *
 *=====*/
PUBLIC struct buf *get_block(dev, block, only_search)
register dev_t dev;          /* on which device is the block? */
register block_t block;      /* which block is wanted? */
int only_search;             /* if NO_READ, don't read, else act normal */
{
/* Check to see if the requested block is in the block cache. If so, return
 * a pointer to it. If not, evict some other block and fetch it (unless
 * 'only_search' is 1). All the blocks in the cache that are not in use
 * are linked together in a chain, with 'front' pointing to the least recently
 * used block and 'rear' to the most recently used block. If 'only_search' is
 * 1, the block being requested will be overwritten in its entirety, so it is
 * only necessary to see if it is in the cache; if it is not, any free buffer
 * will do. It is not necessary to actually read the block in from disk.
 * If 'only_search' is PREFETCH, the block need not be read from the disk,
 * and the device is not to be marked on the block, so callers can tell if
 * the block returned is valid.
 * In addition to the LRU chain, there is also a hash chain to link together
 * blocks whose block numbers end with the same bit strings, for fast lookup.
 */

    int b;
    register struct buf *bp, *prev_ptr;

/* Search the hash chain for (dev, block). Do_read() can use
 * get_block(NO_DEV ...) to get an unnamed block to fill with zeros when
 * someone wants to read from a hole in a file, in which case this search
 * is skipped
 */
    if (dev != NO_DEV) {
        b = (int) block & HASH_MASK;
        bp = buf_hash[b];

        while (bp != NIL_BUF) {
            if (bp->b_blocknr == block && bp->b_dev == dev) {
                /* Block needed has been found. */
                if (bp->b_count == 0) rm_lru(bp);
                bp->b_count++; /* record that block is in use */

                return(bp);
            } else {
                /* This block is not the one sought. */
                bp = bp->b_hash; /* move to next block on hash chain */
            }
        }
    }
}

```



```

/* Desired block is not on available chain. Take oldest block ('front'). */
if ((bp = front) == NIL_BUF) panic(__FILE__, "all buffers in use", NR_BUFS);
rm_lru(bp);

/* Remove the block that was just taken from its hash chain. */
b = (int) bp->b_blocknr & HASH_MASK;
prev_ptr = buf_hash[b];
if (prev_ptr == bp) {
    buf_hash[b] = bp->b_hash;
} else {
    /* The block just taken is not on the front of its hash chain. */
    while (prev_ptr->b_hash != NIL_BUF)
    {
        if (prev_ptr->b_hash == bp) {
            prev_ptr->b_hash = bp->b_hash; /* found it */
            break;
        } else {
            prev_ptr = prev_ptr->b_hash; /* keep looking */
        }
    }
}

/* If the block taken is dirty, make it clean by writing it to the disk.
 * Avoid hysteresis by flushing all other dirty blocks for the same device.
 */
if (bp->b_dev != NO_DEV) {
    if (bp->b_dirt == DIRTY) flushall(bp->b_dev);
}
#ifdef ENABLE_CACHE2
    put_block2(bp);
#endif

/* Fill in block's parameters and add it to the hash chain where it goes. */
bp->b_dev = dev; /* fill in device number */
bp->b_blocknr = block; /* fill in block number */
bp->b_count++; /* record that block is being used */
b = (int) bp->b_blocknr & HASH_MASK;
bp->b_hash = buf_hash[b];
buf_hash[b] = bp; /* add to hash list */

/* Go get the requested block unless searching or prefetching. */
if (dev != NO_DEV) {
#ifdef ENABLE_CACHE2
    if (get_block2(bp, only_search)) /* in 2nd level cache */
    else
#endif
    if (only_search == PREFETCH) bp->b_dev = NO_DEV;
    else
    if (only_search == NORMAL) {
        rw_block(bp, READING);
    }
}
return(bp); /* return the newly acquired block */
}

/*=====
 *                               put_block                               *
 *=====*/
PUBLIC void put_block(bp, block_type)
register struct buf *bp; /* pointer to the buffer to be released */
int block_type; /* INODE_BLOCK, DIRECTORY_BLOCK, or whatever */
{
    /* Return a block to the list of available blocks. Depending on 'block_type'
     * it may be put on the front or rear of the LRU chain. Blocks that are
     * expected to be needed again shortly (e.g., partially full data blocks)
     * go on the rear; blocks that are unlikely to be needed again shortly
     * (e.g., full data blocks) go on the front. Blocks whose loss can hurt
     * the integrity of the file system (e.g., inode blocks) are written to
     * disk immediately if they are dirty.
     */
    if (bp == NIL_BUF) return; /* it is easier to check here than in caller */

    bp->b_count--; /* there is one use fewer now */

```

```

if (bp->b_count != 0) return; /* block is still in use */

bufs_in_use--; /* one fewer block buffers in use */

/* Put this block back on the LRU chain. If the ONE_SHOT bit is set in
 * 'block_type', the block is not likely to be needed again shortly, so put
 * it on the front of the LRU chain where it will be the first one to be
 * taken when a free buffer is needed later.
 */
if (bp->b_dev == DEV_RAM || (block_type & ONE_SHOT)) {
    /* Block probably won't be needed quickly. Put it on front of chain.
     * It will be the next block to be evicted from the cache.
     */
    bp->b_prev = NIL_BUF;
    bp->b_next = front;
    if (front == NIL_BUF)
        rear = bp; /* LRU chain was empty */
    else
        front->b_prev = bp;
    front = bp;
} else {
    /* Block probably will be needed quickly. Put it on rear of chain.
     * It will not be evicted from the cache for a long time.
     */
    bp->b_prev = rear;
    bp->b_next = NIL_BUF;
    if (rear == NIL_BUF)
        front = bp;
    else
        rear->b_next = bp;
    rear = bp;
}

/* Some blocks are so important (e.g., inodes, indirect blocks) that they
 * should be written to the disk immediately to avoid messing up the file
 * system in the event of a crash.
 */
if ((block_type & WRITE_IMMED) && bp->b_dirt == DIRTY && bp->b_dev != NO_DEV) {
    rw_block(bp, WRITING);
}
}

/*=====
 *                               alloc_zone                               *
 *=====*/
PUBLIC zone_t alloc_zone(dev, z)
dev_t dev; /* device where zone wanted */
zone_t z; /* try to allocate new zone near this one */
{
    /* Allocate a new zone on the indicated device and return its number. */

    int major, minor;
    bit_t b, bit;
    struct super_block *sp;

    /* Note that the routine alloc_bit() returns 1 for the lowest possible
     * zone, which corresponds to sp->s_firstdatazone. To convert a value
     * between the bit number, 'b', used by alloc_bit() and the zone number, 'z',
     * stored in the inode, use the formula:
     * z = b + sp->s_firstdatazone - 1
     * Alloc_bit() never returns 0, since this is used for NO_BIT (failure).
     */
    sp = get_super(dev);

    /* If z is 0, skip initial part of the map known to be fully in use. */
    if (z == sp->s_firstdatazone) {
        bit = sp->s_zsearch;
    } else {
        bit = (bit_t) z - (sp->s_firstdatazone - 1);
    }
    b = alloc_bit(sp, ZMAP, bit);
    if (b == NO_BIT) {
        err_code = ENOSPC;
        major = (int) (sp->s_dev >> MAJOR) & BYTE;
    }
}

```

```

        minor = (int) (sp->s_dev >> MINOR) & BYTE;
        printf("No space on %sdevice %d/%d\n",
                sp->s_dev == root_dev ? "root " : "", major, minor);
        return(NO_ZONE);
    }
    if (z == sp->s_firstdatazone) sp->s_zsearch = b;        /* for next time */
    return(sp->s_firstdatazone - 1 + (zone_t) b);
}

/*=====
 *                                free_zone                                *
 *=====*/
PUBLIC void free_zone(dev, numb)
dev_t dev;                /* device where zone located */
zone_t numb;              /* zone to be returned */
{
    /* Return a zone. */

    register struct super_block *sp;
    bit_t bit;

    /* Locate the appropriate super_block and return bit. */
    sp = get_super(dev);
    if (numb < sp->s_firstdatazone || numb >= sp->s_zones) return;
    bit = (bit_t) (numb - (sp->s_firstdatazone - 1));
    free_bit(sp, ZMAP, bit);
    if (bit < sp->s_zsearch) sp->s_zsearch = bit;
}

/*=====
 *                                rw_block                                *
 *=====*/
PRIVATE int rw_block(bp, rw_flag)
register struct buf *bp;    /* buffer pointer */
int rw_flag;               /* READING or WRITING */
{
    /* Read or write a disk block. This is the only routine in which actual disk
     * I/O is invoked. If an error occurs, a message is printed here, but the error
     * is not reported to the caller. If the error occurred while purging a block
     * from the cache, it is not clear what the caller could do about it anyway.
     */

    int r, op;
    off_t pos;
    dev_t dev;
    int block_size;

    block_size = get_block_size(bp->b_dev);

    if ( (dev = bp->b_dev) != NO_DEV) {
        pos = (off_t) bp->b_blocknr * block_size;
        op = (rw_flag == READING ? DEV_READ : DEV_WRITE);
        r = dev_bio(op, dev, FS_PROC_NR, bp->b_data, pos, block_size, 0);
        if (r != block_size) {
            if (r >= 0) r = END_OF_FILE;
            if (r != END_OF_FILE)
                printf("Unrecoverable disk error on device %d/%d, block %ld\n",
                        (dev>>MAJOR)&BYTE, (dev>>MINOR)&BYTE, bp->b_blocknr);
            bp->b_dev = NO_DEV;        /* invalidate block */

            /* Report read errors to interested parties. */
            if (rw_flag == READING) rdwt_err = r;
        }
    }

    bp->b_dirt = CLEAN;

    return OK;
}

/*=====
 *                                invalidate                                *
 *=====*/
PUBLIC void invalidate(device)

```

```

dev_t device;                /* device whose blocks are to be purged */
{
/* Remove all the blocks belonging to some device from the cache. */

    register struct buf *bp;

    for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++)
        if (bp->b_dev == device) bp->b_dev = NO_DEV;

#ifdef ENABLE_CACHE2
    invalidate2(device);
#endif
}

/*=====
*                               flushall                               *
*=====*/
PUBLIC void flushall(dev)
dev_t dev;                    /* device to flush */
{
/* Flush all dirty blocks for one device. */

    register struct buf *bp;
    static struct buf *dirty[NR_BUFS]; /* static so it isn't on stack */
    int ndirty;

    for (bp = &buf[0], ndirty = 0; bp < &buf[NR_BUFS]; bp++)
        if (bp->b_dirt == DIRTY && bp->b_dev == dev) dirty[ndirty++] = bp;
    rw_scattered(dev, dirty, ndirty, WRITING);
}

/*=====
*                               rw_scattered                           *
*=====*/
PUBLIC void rw_scattered(dev, bufq, bufqsize, rw_flag)
dev_t dev;                    /* major-minor device number */
struct buf **bufq;            /* pointer to array of buffers */
int bufqsize;                 /* number of buffers */
int rw_flag;                  /* READING or WRITING */
{
/* Read or write scattered data from a device. */

    register struct buf *bp;
    int gap;
    register int i;
    register iovec_t *iop;
    static iovec_t iovec[NR_IOREQS]; /* static so it isn't on stack */
    int j, r;
    int block_size;

    block_size = get_block_size(dev);

    /* (Shell) sort buffers on b_blocknr. */
    gap = 1;
    do
        gap = 3 * gap + 1;
    while (gap <= bufqsize);
    while (gap != 1) {
        gap /= 3;
        for (j = gap; j < bufqsize; j++) {
            for (i = j - gap;
                i >= 0 && bufq[i]->b_blocknr > bufq[i + gap]->b_blocknr;
                i -= gap) {
                bp = bufq[i];
                bufq[i] = bufq[i + gap];
                bufq[i + gap] = bp;
            }
        }
    }

    /* Set up I/O vector and do I/O. The result of dev_io is OK if everything
     * went fine, otherwise the error code for the first failed transfer.
     */
    while (bufqsize > 0) {

```

```

    for (j = 0, iop = iovec; j < NR_IOREQS && j < bufqsize; j++, iop++) {
        bp = bufq[j];
        if (bp->b_blocknr != bufq[0]->b_blocknr + j) break;
        iop->iov_addr = (vir_bytes) bp->b_data;
        iop->iov_size = block_size;
    }
    r = dev_bio(rw_flag == WRITING ? DEV_SCATTER : DEV_GATHER,
               dev, FS_PROC_NR, iovec,
               (off_t) bufq[0]->b_blocknr * block_size, j, 0);

    /* Harvest the results. Dev_io reports the first error it may have
     * encountered, but we only care if it's the first block that failed.
     */
    for (i = 0, iop = iovec; i < j; i++, iop++) {
        bp = bufq[i];
        if (iop->iov_size != 0) {
            /* Transfer failed. An error? Do we care? */
            if (r != OK && i == 0) {
                printf(
                    "fs: I/O error on device %d/%d, block %lu\n",
                    (dev>>MAJOR)&BYTE, (dev>>MINOR)&BYTE,
                    bp->b_blocknr);
                bp->b_dev = NO_DEV; /* invalidate block */
            }
            break;
        }
        if (rw_flag == READING) {
            bp->b_dev = dev; /* validate block */
            put_block(bp, PARTIAL_DATA_BLOCK);
        } else {
            bp->b_dirt = CLEAN;
        }
    }
    bufq += i;
    bufqsize -= i;
    if (rw_flag == READING) {
        /* Don't bother reading more than the device is willing to
         * give at this time. Don't forget to release those extras.
         */
        while (bufqsize > 0) {
            put_block(*bufq++, PARTIAL_DATA_BLOCK);
            bufqsize--;
        }
    }
    if (rw_flag == WRITING && i == 0) {
        /* We're not making progress, this means we might keep
         * looping. Buffers remain dirty if un-written. Buffers are
         * lost if invalidate()d or LRU-removed while dirty. This
         * is better than keeping unwritable blocks around forever..
         */
        break;
    }
}
}

/*=====
 *                               rm_lru                               *
 *=====*/
PRIVATE void rm_lru(bp)
struct buf *bp;
{
    /* Remove a block from its LRU chain. */
    struct buf *next_ptr, *prev_ptr;

    bufs_in_use++;
    next_ptr = bp->b_next; /* successor on LRU chain */
    prev_ptr = bp->b_prev; /* predecessor on LRU chain */
    if (prev_ptr != NIL_BUF)
    {
        prev_ptr->b_next = next_ptr;
    }
    else
        front = next_ptr; /* this block was at front of chain */
}

```

```

if (next_ptr != NIL_BUF)
{
    next_ptr->b_prev = prev_ptr;
}
else
    rear = prev_ptr;          /* this block was at rear of chain */
}

#if 0
PRIVATE void check_lru()
{
    int i;
    struct buf *bp, *nbp;

    for (i= 0; i<NR_BUFS; i++)
    {
        bp= &buf[i];
        nbp= bp->b_next;
        if (nbp != NULL && (nbp < buf || nbp >= &buf[NR_BUFS]))
        {
            stacktrace();
            panic(__FILE__, "check_lru: bad next", nbp);
        }
        nbp= bp->b_prev;
        if (nbp != NULL && (nbp < buf || nbp >= &buf[NR_BUFS]))
        {
            stacktrace();
            panic(__FILE__, "check_lru: bad next", nbp);
        }
    }
}

PRIVATE void check_buf(bp)
struct buf *bp;
{
    struct buf *nbp;

    if (bp < buf || bp >= &buf[NR_BUFS])
    {
        stacktrace();
        panic(__FILE__, "check_buf: bad buf", bp);
    }
    nbp= bp->b_next;
    if (nbp != NULL && (nbp < buf || nbp >= &buf[NR_BUFS]))
    {
        stacktrace();
        panic(__FILE__, "check_buf: bad next", nbp);
    }
    nbp= bp->b_prev;
    if (nbp != NULL && (nbp < buf || nbp >= &buf[NR_BUFS]))
    {
        stacktrace();
        panic(__FILE__, "check_buf: bad next", nbp);
    }
}

PRIVATE void check_hash_chains()
{
    int i;
    struct buf *bp;

    for (i= 0; i<NR_BUFS; i++)
    {
        bp= &buf[i];
        while (bp)
        {
            if (bp < buf || bp >= &buf[NR_BUFS])
            {
                panic(__FILE__, "check_hash_chains: bad buf",
                    bp);
            }
            bp= bp->b_hash;
        }
    }
}

```

```
}

PUBLIC void check_hash_chainsX(file, line)
char *file;
int line;
{
    int i;
    struct buf *bp;

    for (i= 0; i<NR_BUF_HASH; i++)
    {
        bp= buf_hash[i];
        while (bp)
        {
            if (bp < buf || bp >= &buf[NR_BUFS])
            {
                printf(
                    "check_hash_chainsX: called from %s, %d\n",
                    file, line);
                panic(__FILE__, "check_hash_chainsX: bad buf",
                    bp);
            }
            bp= bp->b_hash;
        }
    }
}

PRIVATE void check_hash_chain(bp)
struct buf *bp;
{
    while (bp)
    {
        if (bp < buf || bp >= &buf[NR_BUFS])
        {
            panic(__FILE__, "check_hash_chain: bad buf", bp);
        }
        bp= bp->b_hash;
    }
}

#endif
```

```

/* Second level block cache to supplement the file system cache. The block
 * cache of a 16-bit Minix system is very small, too small to prevent trashing.
 * A generic 32-bit system also doesn't have a very large cache to allow it
 * to run on systems with little memory. On a system with lots of memory one
 * can use the RAM disk as a read-only second level cache. Any blocks pushed
 * out of the primary cache are cached on the RAM disk. This code manages the
 * second level cache. The cache is a simple FIFO where old blocks are put
 * into and drop out at the other end. Must be searched backwards.
 *
 * The entry points into this file are:
 *   init_cache2: initialize the second level cache
 *   get_block2:  get a block from the 2nd level cache
 *   put_block2:  store a block in the 2nd level cache
 *   invalidate2: remove all the cache blocks on some device
 */

#include "fs.h"
#include <minix/com.h>
#include "buf.h"

#if ENABLE_CACHE2

#define MAX_BUF2      (256 * sizeof(char *))

PRIVATE struct buf2 { /* 2nd level cache per block administration */
    block_t b2_blocknr; /* block number */
    dev_t b2_dev; /* device number */
    ul6_t b2_count; /* count of in-cache block groups */
} buf2[MAX_BUF2];

PRIVATE unsigned nr_buf2; /* actual cache size */
PRIVATE unsigned buf2_idx; /* round-robin reuse index */

#define hash2(block) ((unsigned) ((block) & (MAX_BUF2 - 1)))

/*=====
 *                               init_cache2                               *
 *=====*/
PUBLIC void init_cache2(size)
unsigned long size;
{
/* Initialize the second level disk buffer cache of 'size' blocks. */

    nr_buf2 = size > MAX_BUF2 ? MAX_BUF2 : (unsigned) size;
}

/*=====
 *                               get_block2                               *
 *=====*/
PUBLIC int get_block2(bp, only_search)
struct buf *bp; /* buffer to get from the 2nd level cache */
int only_search; /* if NO_READ, do nothing, else act normal */
{
/* Fill a buffer from the 2nd level cache. Return true iff block acquired. */
    unsigned b;
    struct buf2 *bp2;

    /* If the block wanted is in the RAM disk then our game is over. */
    if (bp->b_dev == DEV_RAM) nr_buf2 = 0;

    /* Cache enabled? NO_READ? Any blocks with the same hash key? */
    if (nr_buf2 == 0 || only_search == NO_READ
        || buf2[hash2(bp->b_blocknr)].b2_count == 0) return(0);

    /* Search backwards (there may be older versions). */
    b = buf2_idx;
    for (;;) {
        if (b == 0) b = nr_buf2;
        bp2 = &buf2[--b];
        if (bp2->b2_blocknr == bp->b_blocknr && bp2->b2_dev == bp->b_dev) break;
        if (b == buf2_idx) return(0);
    }

    /* Block is in the cache, get it. */

```



```

    if (dev_io(DEV_READ, DEV_RAM, FS_PROC_NR, bp->b_data,
               (off_t) b * BLOCK_SIZE, BLOCK_SIZE, 0) == BLOCK_SIZE) {
        return(1);
    }
    return(0);
}

/*=====*
 *                               put_block2                               *
 *=====*/
PUBLIC void put_block2(bp)
struct buf *bp;                /* buffer to store in the 2nd level cache */
{
    /* Store a buffer into the 2nd level cache. */
    unsigned b;
    struct buf2 *bp2;

    if (nr_buf2 == 0) return;    /* no 2nd level cache */

    b = buf2_idx++;
    if (buf2_idx == nr_buf2) buf2_idx = 0;

    bp2 = &buf2[b];

    if (dev_io(DEV_WRITE, DEV_RAM, FS_PROC_NR, bp->b_data,
               (off_t) b * BLOCK_SIZE, BLOCK_SIZE, 0) == BLOCK_SIZE) {
        if (bp2->b2_dev != NO_DEV) buf2[hash2(bp2->b2_blocknr)].b2_count--;
        bp2->b2_dev = bp->b_dev;
        bp2->b2_blocknr = bp->b_blocknr;
        buf2[hash2(bp2->b2_blocknr)].b2_count++;
    }
}

/*=====*
 *                               invalidate2                               *
 *=====*/
PUBLIC void invalidate2(device)
dev_t device;
{
    /* Invalidate all blocks from a given device in the 2nd level cache. */
    unsigned b;
    struct buf2 *bp2;

    for (b = 0; b < nr_buf2; b++) {
        bp2 = &buf2[b];
        if (bp2->b2_dev == device) {
            bp2->b2_dev = NO_DEV;
            buf2[hash2(bp2->b2_blocknr)].b2_count--;
        }
    }
}
#endif /* ENABLE_CACHE2 */

```

```

/* Tables sizes */
#define V1_NR_DZONES      7      /* # direct zone numbers in a V1 inode */
#define V1_NR_TZONES      9      /* total # zone numbers in a V1 inode */
#define V2_NR_DZONES      7      /* # direct zone numbers in a V2 inode */
#define V2_NR_TZONES     10      /* total # zone numbers in a V2 inode */

#define NR_FILPS          256     /* # slots in filp table */
#define NR_INODES         256     /* # slots in "in core" inode table */
#define NR_SUPERS          12     /* # slots in super block table */
#define NR_LOCKS           8      /* # slots in the file locking table */

/* The type of sizeof may be (unsigned) long. Use the following macro for
 * taking the sizes of small objects so that there are no surprises like
 * (small) long constants being passed to routines expecting an int.
 */
#define usesizeof(t) ((unsigned) sizeof(t))

/* File system types. */
#define SUPER_MAGIC      0x137F   /* magic number contained in super-block */
#define SUPER_REV        0x7F13   /* magic # when 68000 disk read on PC or vv */
#define SUPER_V2         0x2468   /* magic # for V2 file systems */
#define SUPER_V2_REV     0x6824   /* V2 magic written on PC, read on 68K or vv */
#define SUPER_V3         0x4d5a   /* magic # for V3 file systems */

#define V1                1       /* version number of V1 file systems */
#define V2                2       /* version number of V2 file systems */
#define V3                3       /* version number of V3 file systems */

/* Miscellaneous constants */
#define SU_UID ((uid_t) 0)        /* super_user's uid_t */
#define SERVERS_UID ((uid_t) 11) /* who may do FSSIGNON */
#define SYS_UID ((uid_t) 0)       /* uid_t for processes MM and INIT */
#define SYS_GID ((gid_t) 0)       /* gid_t for processes MM and INIT */
#define NORMAL      0             /* forces get_block to do disk read */
#define NO_READ      1            /* prevents get_block from doing disk read */
#define PREFETCH     2            /* tells get_block not to read or mark dev */

#define XPIPE (-NR_TASKS-1)       /* used in fp_task when susp'd on pipe */
#define XLOCK (-NR_TASKS-2)       /* used in fp_task when susp'd on lock */
#define XOPEN (-NR_TASKS-3)       /* used in fp_task when susp'd on pipe open */
#define XSELECT (-NR_TASKS-4)     /* used in fp_task when susp'd on select */

#define NO_BIT ((bit_t) 0)        /* returned by alloc_bit() to signal failure */

#define DUP_MASK      0100        /* mask to distinguish dup2 from dup */

#define LOOK_UP        0          /* tells search_dir to lookup string */
#define ENTER          1          /* tells search_dir to make dir entry */
#define DELETE         2          /* tells search_dir to delete entry */
#define IS_EMPTY       3          /* tells search_dir to ret. OK or ENOTEMPTY */

/* write_map() args */
#define WMAP_FREE      (1 << 0)

#define PATH_TRANSPARENT 000      /* parse_path stops at final object */
#define PATH_PENULTIMATE 001      /* parse_path stops at last but one name */
#define PATH_OPAQUE      002      /* parse_path stops at final name */
#define PATH_NONSYMBOLIC 004      /* parse_path scans final name if symbolic */
#define PATH_STRIPDOT    010      /* parse_path strips /. from path */
#define EAT_PATH          PATH_TRANSPARENT
#define EAT_PATH_OPAQUE   PATH_OPAQUE
#define LAST_DIR          PATH_PENULTIMATE
#define LAST_DIR_NOTDOT   PATH_PENULTIMATE | PATH_STRIPDOT
#define LAST_DIR_EATSYM   PATH_NONSYMBOLIC
#define SYMLOOP           16

#define CLEAN            0         /* disk and memory copies identical */
#define DIRTY            1         /* disk and memory copies differ */
#define ATIME            002       /* set if atime field needs updating */
#define CTIME            004       /* set if ctime field needs updating */
#define MTIME            010       /* set if mtime field needs updating */

#define BYTE_SWAP        0         /* tells conv2/conv4 to swap bytes */

```

```
#define END_OF_FILE      (-104)      /* eof detected */

#define ROOT_INODE       1           /* inode number for root directory */
#define BOOT_BLOCK      ((block_t) 0) /* block number of boot block */
#define SUPER_BLOCK_BYTES (1024)     /* bytes offset */
#define START_BLOCK      2          /* first block of FS (not counting SB) */

#define DIR_ENTRY_SIZE   sizeof (struct direct) /* # bytes/dir entry */
#define NR_DIR_ENTRIES(b) ((b)/DIR_ENTRY_SIZE) /* # dir entries/blk */
#define SUPER_SIZE       sizeof (struct super_block) /* super_block size */
#define PIPE_SIZE(b)     (V1_NR_DZONES*(b)) /* pipe size in bytes */

#define FS_BITMAP_CHUNKS(b) ((b)/sizeof (bitchunk_t)) /* # map chunks/blk */
#define FS_BITCHUNK_BITS   (sizeof(bitchunk_t) * CHAR_BIT)
#define FS_BITS_PER_BLOCK(b) (FS_BITMAP_CHUNKS(b) * FS_BITCHUNK_BITS)

/* Derived sizes pertaining to the V1 file system. */
#define V1_ZONE_NUM_SIZE   sizeof (zone1_t) /* # bytes in V1 zone */
#define V1_INODE_SIZE      sizeof (d1_inode) /* bytes in V1 disk ino */

/* # zones/indir block */
#define V1_INDIRECTS (_STATIC_BLOCK_SIZE/V1_ZONE_NUM_SIZE)

/* # V1 disk inodes/blk */
#define V1_INODES_PER_BLOCK (_STATIC_BLOCK_SIZE/V1_INODE_SIZE)

/* Derived sizes pertaining to the V2 file system. */
#define V2_ZONE_NUM_SIZE   sizeof (zone_t) /* # bytes in V2 zone */
#define V2_INODE_SIZE      sizeof (d2_inode) /* bytes in V2 disk ino */
#define V2_INDIRECTS(b) ((b)/V2_ZONE_NUM_SIZE) /* # zones/indir block */
#define V2_INODES_PER_BLOCK(b) ((b)/V2_INODE_SIZE) /* # V2 disk inodes/blk */
```

```

/* When a needed block is not in the cache, it must be fetched from the disk.
 * Special character files also require I/O. The routines for these are here.
 *
 * The entry points in this file are:
 * dev_open: FS opens a device
 * dev_close: FS closes a device
 * dev_io: FS does a read or write on a device
 * dev_status: FS processes callback request alert
 * gen_opcl: generic call to a task to perform an open/close
 * gen_io: generic call to a task to perform an I/O operation
 * no_dev: open/close processing for devices that don't exist
 * no_dev_io: i/o processing for devices that don't exist
 * tty_opcl: perform tty-specific processing for open/close
 * cttty_opcl: perform controlling-tty-specific processing for open/close
 * cttty_io: perform controlling-tty-specific processing for I/O
 * do_ioctl: perform the IOCTL system call
 * pm_setsid: perform the SETSID system call (FS side)
 */

#include "fs.h"
#include <fcntl.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <minix/endpoint.h>
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

#define ELEMENTS(a) (sizeof(a)/sizeof((a)[0]))

extern int dmap_size;
PRIVATE int dummyproc;

/*=====
 *
 * dev_open
 *=====*/
PUBLIC int dev_open(dev, proc, flags)
dev_t dev; /* device to open */
int proc; /* process to open for */
int flags; /* mode bits and flags */
{
    int major, r;
    struct dmap *dp;

    /* Determine the major device number call the device class specific
     * open/close routine. (This is the only routine that must check the
     * device number for being in range. All others can trust this check.)
     */
    major = (dev >> MAJOR) & BYTE;
    if (major >= NR_DEVICES) major = 0;
    dp = &dmap[major];
    if (dp->dmap_driver == NONE)
        return ENXIO;
    r = (*dp->dmap_opcl)(DEV_OPEN, dev, proc, flags);
    if (r == SUSPEND) panic(__FILE__, "suspend on open from", dp->dmap_driver);
    return(r);
}

/*=====
 *
 * dev_close
 *=====*/
PUBLIC void dev_close(dev)
dev_t dev; /* device to close */
{
    /* See if driver is roughly valid. */
    if (dmap[(dev >> MAJOR)].dmap_driver == NONE) {
        return;
    }
    (void) (*dmap[(dev >> MAJOR) & BYTE].dmap_opcl)(DEV_CLOSE, dev, 0, 0);
}

/*=====

```

```

*                                     dev_status                                     *
*=====*/
PUBLIC void dev_status(message *m)
{
    message st;
    int d, get_more = 1;

    for(d = 0; d < NR_DEVICES; d++)
        if (dmap[d].dmap_driver != NONE &&
            dmap[d].dmap_driver == m->m_source)
            break;

    if (d >= NR_DEVICES)
        return;

    do {
        int r;
        st.m_type = DEV_STATUS;
        if ((r=sendrec(m->m_source, &st)) != OK) {
            printf("DEV_STATUS failed to %d: %d\n", m->m_source, r);
            if (r == EDEADSRCDST) return;
            if (r == EDSTDIED) return;
            if (r == ESRCDIED) return;
            panic(__FILE__, "couldn't sendrec for DEV_STATUS", r);
        }

        switch(st.m_type) {
            case DEV_REVIVE:
                revive(st.REP_ENDPT, st.REP_STATUS);
                break;
            case DEV_IO_READY:
                select_notified(d, st.DEV_MINOR, st.DEV_SEL_OPS);
                break;
            default:
                printf("FS: unrecognized reply %d to DEV_STATUS\n", st.m_type);
                /* Fall through. */
            case DEV_NO_STATUS:
                get_more = 0;
                break;
        }
    } while(get_more);

    return;
}

/*=====*/
*                                     dev_bio                                     *
*=====*/
PUBLIC int dev_bio(op, dev, proc_e, buf, pos, bytes, flags)
int op;                                     /* DEV_READ, DEV_WRITE, DEV_IOCTL, etc. */
dev_t dev;                                /* major-minor device number */
int proc_e;                               /* in whose address space is buf? */
void *buf;                                /* virtual address of the buffer */
off_t pos;                                /* byte position */
int bytes;                                /* how many bytes to transfer */
int flags;                                /* special flags, like O_NONBLOCK */
{
    /* Read or write from a device. The parameter 'dev' tells which one. */
    struct dmap *dp;
    int r;
    message m;

    /* Determine task dmap. */
    dp = &dmap[(dev >> MAJOR) & BYTE];

    for (;;)
    {
        /* See if driver is roughly valid. */
        if (dp->dmap_driver == NONE) {
            printf("FS: dev_io: no driver for dev %x\n", dev);
            return ENXIO;
        }

        /* Set up the message passed to task. */

```

```

    m.m_type    = op;
    m.DEVICE    = (dev >> MINOR) & BYTE;
    m.POSITION  = pos;
    m.IO_ENDPT  = proc_e;
    m.ADDRESS   = buf;
    m.COUNT     = bytes;
    m.TTY_FLAGS = flags;

    /* Call the task. */
    (*dp->dmap_io)(dp->dmap_driver, &m);

    if(dp->dmap_driver == NONE) {
        /* Driver has vanished. Wait for a new one. */
        for (;;)
        {
            r= receive(RS_PROC_NR, &m);
            if (r != OK)
            {
                panic(__FILE__,
                    "dev_bio: unable to receive from RS",
                    r);
            }
            if (m.m_type == DEVCTL)
            {
                r= fs_devctl(m.ctl_req, m.dev_nr, m.driver_nr,
                    m.dev_style, m.m_force);
            }
            else
            {
                panic(__FILE__,
                    "dev_bio: got message from RS, type",
                    m.m_type);
            }
            m.m_type= r;
            r= send(RS_PROC_NR, &m);
            if (r != OK)
            {
                panic(__FILE__,
                    "dev_bio: unable to send to RS",
                    r);
            }
            if (dp->dmap_driver != NONE)
                break;
        }
        printf("dev_bio: trying new driver\n");
        continue;
    }

    /* Task has completed. See if call completed. */
    if (m.REP_STATUS == SUSPEND) {
        panic(__FILE__, "dev_bio: driver returned SUSPEND", NO_NUM);
    }
    return(m.REP_STATUS);
}
}

/*=====
*                               dev_io                               *
*=====*/
PUBLIC int dev_io(op, dev, proc_e, buf, pos, bytes, flags)
int op;                               /* DEV_READ, DEV_WRITE, DEV_IOCTL, etc. */
dev_t dev;                           /* major-minor device number */
int proc_e;                          /* in whose address space is buf? */
void *buf;                           /* virtual address of the buffer */
off_t pos;                           /* byte position */
int bytes;                           /* how many bytes to transfer */
int flags;                           /* special flags, like O_NONBLOCK */
{
    /* Read or write from a device. The parameter 'dev' tells which one. */
    struct dmap *dp;
    message dev_mess;

    /* Determine task dmap. */
    dp = &dmap[(dev >> MAJOR) & BYTE];

```

```

/* See if driver is roughly valid. */
if (dp->dmap_driver == NONE) {
    printf("FS: dev_io: no driver for dev %x\n", dev);
    return ENXIO;
}

if(isokendpt(dp->dmap_driver, &dummysproc) != OK) {
    printf("FS: dev_io: old driver for dev %x (%d)\n",
        dev, dp->dmap_driver);
    return ENXIO;
}

/* Set up the message passed to task. */
dev_mess.m_type = op;
dev_mess.DEVICE = (dev >> MINOR) & BYTE;
dev_mess.POSITION = pos;
dev_mess.IO_ENDPT = proc_e;
dev_mess.ADDRESS = buf;
dev_mess.COUNT = bytes;
dev_mess.TTY_FLAGS = flags;

/* Call the task. */
(*dp->dmap_io)(dp->dmap_driver, &dev_mess);

if(dp->dmap_driver == NONE) {
    /* Driver has vanished. */
    return EIO;
}

/* Task has completed. See if call completed. */
if (dev_mess.REP_STATUS == SUSPEND) {
    if (flags & O_NONBLOCK) {
        /* Not supposed to block. */
        dev_mess.m_type = CANCEL;
        dev_mess.IO_ENDPT = proc_e;
        dev_mess.DEVICE = (dev >> MINOR) & BYTE;
        (*dp->dmap_io)(dp->dmap_driver, &dev_mess);
        if (dev_mess.REP_STATUS == EINTR) dev_mess.REP_STATUS = EAGAIN;
    } else {
        /* Suspend user. */
        suspend(dp->dmap_driver);
        return(SUSPEND);
    }
}
return(dev_mess.REP_STATUS);
}

/*=====
*                               gen_opcl                               *
*=====*/
PUBLIC int gen_opcl(op, dev, proc_e, flags)
int op;                /* operation, DEV_OPEN or DEV_CLOSE */
dev_t dev;            /* device to open or close */
int proc_e;           /* process to open/close for */
int flags;            /* mode bits and flags */
{
/* Called from the dmap struct in table.c on opens & closes of special files.*/
    struct dmap *dp;
    message dev_mess;

    /* Determine task dmap. */
    dp = &dmap[(dev >> MAJOR) & BYTE];

    dev_mess.m_type = op;
    dev_mess.DEVICE = (dev >> MINOR) & BYTE;
    dev_mess.IO_ENDPT = proc_e;
    dev_mess.COUNT = flags;

    if (dp->dmap_driver == NONE) {
        printf("FS: gen_opcl: no driver for dev %x\n", dev);
        return ENXIO;
    }
}

```

```

/* Call the task. */
(dp->dmap_io)(dp->dmap_driver, &dev_mess);

return(dev_mess.REP_STATUS);
}

/*=====
 *                               tty_opcl                               *
 *=====*/
PUBLIC int tty_opcl(op, dev, proc_e, flags)
int op;                               /* operation, DEV_OPEN or DEV_CLOSE */
dev_t dev;                           /* device to open or close */
int proc_e;                          /* process to open/close for */
int flags;                           /* mode bits and flags */
{
/* This procedure is called from the dmap struct on tty open/close. */

int r;
register struct fproc *rfp;

/* Add O_NOCTTY to the flags if this process is not a session leader, or
 * if it already has a controlling tty, or if it is someone elses
 * controlling tty.
 */
if (!fp->fp_sesldr || fp->fp_tty != 0) {
    flags |= O_NOCTTY;
} else {
    for (rfp = &fproc[0]; rfp < &fproc[NR_PROCS]; rfp++) {
        if(rfp->fp_pid == PID_FREE) continue;
        if (rfp->fp_tty == dev) flags |= O_NOCTTY;
    }
}

r = gen_opcl(op, dev, proc_e, flags);

/* Did this call make the tty the controlling tty? */
if (r == 1) {
    fp->fp_tty = dev;
    r = OK;
}
return(r);
}

/*=====
 *                               ctty_opcl                               *
 *=====*/
PUBLIC int ctty_opcl(op, dev, proc_e, flags)
int op;                               /* operation, DEV_OPEN or DEV_CLOSE */
dev_t dev;                           /* device to open or close */
int proc_e;                          /* process to open/close for */
int flags;                           /* mode bits and flags */
{
/* This procedure is called from the dmap struct in table.c on opening/closing
 * /dev/tty, the magic device that translates to the controlling tty.
 */

return(fp->fp_tty == 0 ? ENXIO : OK);
}

/*=====
 *                               pm_setsid                               *
 *=====*/
PUBLIC void pm_setsid(proc_e)
int proc_e;
{
/* Perform the FS side of the SETSID call, i.e. get rid of the controlling
 * terminal of a process, and make the process a session leader.
 */
register struct fproc *rfp;
int slot;

/* Make the process a session leader with no controlling tty. */
okendpt(proc_e, &slot);
rfp = &fproc[slot];

```



```

    rfp->fp_sesldr = TRUE;
    rfp->fp_tty = 0;
}

/*=====
 *                               do_ioctl                               *
 *=====*/
PUBLIC int do_ioctl()
{
    /* Perform the ioctl(ls_fd, request, argx) system call (uses m2 fmt). */

    struct filp *f;
    register struct inode *rip;
    dev_t dev;

    if ( (f = get_filp(m_in.ls_fd)) == NIL_FILP) return(err_code);
    rip = f->filp_ino;          /* get inode pointer */
    if ( (rip->i_mode & I_TYPE) != I_CHAR_SPECIAL
        && (rip->i_mode & I_TYPE) != I_BLOCK_SPECIAL) return(ENOTTY);
    dev = (dev_t) rip->i_zone[0];

#ifdef ENABLE_BINCOMPAT
    if ((m_in.TTY_REQUEST >> 8) == 't') {
        /* Obsolete sgtty ioctl, message contains more than is sane. */
        struct dmap *dp;
        message dev_mess;

        dp = &dmap[(dev >> MAJOR) & BYTE];

        dev_mess = m;          /* Copy full message with all the weird bits. */
        dev_mess.m_type = DEV_IOCTL;
        dev_mess.PROC_NR = who_e;
        dev_mess.TTY_LINE = (dev >> MINOR) & BYTE;

        /* Call the task. */

        if (dp->dmap_driver == NONE) {
            printf("FS: do_ioctl: no driver for dev %x\n", dev);
            return ENXIO;
        }

        if(isokendpt(dp->dmap_driver, &dummysproc) != OK) {
            printf("FS: do_ioctl: old driver for dev %x (%d)\n",
                dev, dp->dmap_driver);
            return ENXIO;
        }

        (*dp->dmap_io)(dp->dmap_driver, &dev_mess);

        m_out.TTY_SPEK = dev_mess.TTY_SPEK;          /* erase and kill */
        m_out.TTY_FLAGS = dev_mess.TTY_FLAGS;        /* flags */
        return(dev_mess.REP_STATUS);
    }
#endif

    return(dev_io(DEV_IOCTL, dev, who_e, m_in.ADDRESS, 0L,
        m_in.REQUEST, f->filp_flags));
}

/*=====
 *                               gen_io                               *
 *=====*/
PUBLIC int gen_io(task_nr, mess_ptr)
int task_nr;          /* which task to call */
message *mess_ptr;    /* pointer to message for task */
{
    /* All file system I/O ultimately comes down to I/O on major/minor device
     * pairs. These lead to calls on the following routines via the dmap table.
     */

    int r, proc_e;

    proc_e = mess_ptr->IO_ENDPT;

```

```

#if DEAD_CODE
    while ((r = sendrec(task_nr, mess_ptr)) == ELOCKED) {
        /* sendrec() failed to avoid deadlock. The task 'task_nr' is
         * trying to send a REVIVE message for an earlier request.
         * Handle it and go try again.
         */
        if ((r = receive(task_nr, &local_m)) != OK) {
            break;
        }

        /* If we're trying to send a cancel message to a task which has just
         * sent a completion reply, ignore the reply and abort the cancel
         * request. The caller will do the revive for the process.
         */
        if (mess_ptr->m_type == CANCEL && local_m.REP_ENDPT == proc_e) {
            return OK;
        }

        /* Otherwise it should be a REVIVE. */
        if (local_m.m_type != REVIVE) {
            printf(
                "fs: strange device reply from %d, type = %d, proc = %d (1)\n",
                local_m.m_source,
                local_m.m_type, local_m.REP_ENDPT);
            continue;
        }

        revive(local_m.REP_ENDPT, local_m.REP_STATUS);
    }
#endif

    /* The message received may be a reply to this call, or a REVIVE for some
     * other process.
     */
    r = sendrec(task_nr, mess_ptr);
    for(;;) {
        if (r != OK) {
            if (r == EDEADSRCDST || r == EDSTDIED || r == ESRCDIED) {
                printf("fs: dead driver %d\n", task_nr);
                dmap_unmap_by_endpt(task_nr);
                return r;
            }
            if (r == ELOCKED) {
                printf("fs: ELOCKED talking to %d\n", task_nr);
                return r;
            }
            panic(__FILE__, "call_task: can't send/receive", r);
        }

        /* Did the process we did the sendrec() for get a result? */
        if (mess_ptr->REP_ENDPT == proc_e) {
            break;
        } else if (mess_ptr->m_type == REVIVE) {
            /* Otherwise it should be a REVIVE. */
            revive(mess_ptr->REP_ENDPT, mess_ptr->REP_STATUS);
        } else {
            printf(
                "fs: strange device reply from %d, type = %d, proc = %d (2) ignored\n",
                mess_ptr->m_source,
                mess_ptr->m_type, mess_ptr->REP_ENDPT);
        }
        r = receive(task_nr, mess_ptr);
    }

    return OK;
}

/*=====
 *
 *                               ctty_io
 *=====*/
PUBLIC int ctty_io(task_nr, mess_ptr)
int task_nr;          /* not used - for compatibility with dmap_t */
message *mess_ptr;    /* pointer to message for task */
{

```

```

/* This routine is only called for one device, namely /dev/tty. Its job
 * is to change the message to use the controlling terminal, instead of the
 * major/minor pair for /dev/tty itself.
 */

struct dmap *dp;

if (fp->fp_tty == 0) {
    /* No controlling tty present anymore, return an I/O error. */
    mess_ptr->REP_STATUS = EIO;
} else {
    /* Substitute the controlling terminal device. */
    dp = &dmap[(fp->fp_tty >> MAJOR) & BYTE];
    mess_ptr->DEVICE = (fp->fp_tty >> MINOR) & BYTE;

    if (dp->dmap_driver == NONE) {
        printf("FS: cty_io: no driver for dev\n");
        return EIO;
    }

    if(isokendpt(dp->dmap_driver, &dummysproc) != OK) {
        printf("FS: cty_io: old driver %d\n",
            dp->dmap_driver);
        return EIO;
    }

    (*dp->dmap_io)(dp->dmap_driver, mess_ptr);
}
return OK;
}

/*=====
 *                               no_dev                               *
 *=====*/
PUBLIC int no_dev(op, dev, proc, flags)
int op;                /* operation, DEV_OPEN or DEV_CLOSE */
dev_t dev;             /* device to open or close */
int proc;              /* process to open/close for */
int flags;             /* mode bits and flags */
{
    /* Called when opening a nonexistent device. */
    return(ENODEV);
}

/*=====
 *                               no_dev_io                           *
 *=====*/
PUBLIC int no_dev_io(int proc, message *m)
{
    /* Called when doing i/o on a nonexistent device. */
    printf("FS: I/O on unmapped device number\n");
    return EIO;
}

/*=====
 *                               clone_opcl                           *
 *=====*/
PUBLIC int clone_opcl(op, dev, proc_e, flags)
int op;                /* operation, DEV_OPEN or DEV_CLOSE */
dev_t dev;             /* device to open or close */
int proc_e;            /* process to open/close for */
int flags;             /* mode bits and flags */
{
    /* Some devices need special processing upon open. Such a device is "cloned",
     * i.e. on a succesful open it is replaced by a new device with a new unique
     * minor device number. This new device number identifies a new object (such
     * as a new network connection) that has been allocated within a task.
     */
    struct dmap *dp;
    int r, minor;
    message dev_mess;

    /* Determine task dmap. */
    dp = &dmap[(dev >> MAJOR) & BYTE];

```

```

minor = (dev >> MINOR) & BYTE;

dev_mess.m_type    = op;
dev_mess.DEVICE    = minor;
dev_mess.IO_ENDPT  = proc_e;
dev_mess.COUNT     = flags;

if (dp->dmap_driver == NONE) {
    printf("FS: clone_opcl: no driver for dev %x\n", dev);
    return ENXIO;
}

if(isokendpt(dp->dmap_driver, &dummysproc) != OK) {
    printf("FS: clone_opcl: old driver for dev %x (%d)\n",
        dev, dp->dmap_driver);
    return ENXIO;
}

/* Call the task. */
r = (*dp->dmap_io)(dp->dmap_driver, &dev_mess);
if (r != OK)
    return r;

if (op == DEV_OPEN && dev_mess.REP_STATUS >= 0) {
    if (dev_mess.REP_STATUS != minor) {
        /* A new minor device number has been returned. Create a
         * temporary device file to hold it.
         */
        struct inode *ip;

        /* Device number of the new device. */
        dev = (dev & ~(BYTE << MINOR)) | (dev_mess.REP_STATUS << MINOR);

        ip = alloc_inode(root_dev, ALL_MODES | I_CHAR_SPECIAL);
        if (ip == NIL_INODE) {
            /* Oops, that didn't work. Undo open. */
            (void) clone_opcl(DEV_CLOSE, dev, proc_e, 0);
            return(err_code);
        }
        ip->i_zone[0] = dev;

        put_inode(fp->fp_filp[m_in.fd]->filp_ino);
        fp->fp_filp[m_in.fd]->filp_ino = ip;
    }
    dev_mess.REP_STATUS = OK;
}
return(dev_mess.REP_STATUS);
}

/*=====
 *                               dev_up                               *
 *=====*/
PUBLIC void dev_up(int maj)
{
    /* A new device driver has been mapped in. This function
     * checks if any filesystems are mounted on it, and if so,
     * dev_open()s them so the filesystem can be reused.
     */
    struct super_block *sb;
    struct filp *fp;
    int r;

    /* Open a device once for every filp that's opened on it,
     * and once for every filesystem mounted from it.
     */

    for(sb = super_block; sb < &super_block[NR_SUPERS]; sb++) {
        int minor;
        if(sb->s_dev == NO_DEV)
            continue;
        if(((sb->s_dev >> MAJOR) & BYTE) != maj)
            continue;
        minor = ((sb->s_dev >> MINOR) & BYTE);

```

```
    printf("FS: remounting dev %d/%d\n", maj, minor);
    if((r = dev_open(sb->s_dev, FS_PROC_NR,
        sb->s_rd_only ? R_BIT : (R_BIT|W_BIT))) != OK) {
        printf("FS: mounted dev %d/%d re-open failed: %d.\n",
            maj, minor, r);
    }
}

for(fp = filp; fp < &filp[NR_FILPS]; fp++) {
    struct inode *in;
    int minor;

    if(fp->filp_count < 1 || !(in=fp->filp_ino)) continue;
    if(((in->i_zone[0] >> MAJOR) & BYTE) != maj) continue;
    if(!(in->i_mode & (I_BLOCK_SPECIAL|I_CHAR_SPECIAL))) continue;

    minor = ((in->i_zone[0] >> MINOR) & BYTE);

    printf("FS: reopening special %d/%d.\n", maj, minor);

    if((r = dev_open(in->i_zone[0], FS_PROC_NR,
        in->i_mode & (R_BIT|W_BIT))) != OK) {
        int n;
        /* This function will set the fp_filp[]s of processes
         * holding that fp to NULL, but _not_ clear
         * fp_filp_inuse, so that fd can't be recycled until
         * it's close()d.
         */
        n = inval_filp(fp);
        if(n != fp->filp_count)
            printf("FS: warning: invalidate/count "
                "discrepancy (%d,%d)\n", n, fp->filp_count);
        fp->filp_count = 0;
        printf("FS: file on dev %d/%d re-open failed: %d; "
            "invalidated %d fd's.\n", maj, minor, r, n);
    }
}

return;
}
```

```

/* This file contains the table with device <-> driver mappings. It also
 * contains some routines to dynamically add and/ or remove device drivers
 * or change mappings.
 */

#include "fs.h"
#include "fproc.h"
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include <minix/com.h>
#include "param.h"

/* Some devices may or may not be there in the next table. */
#define DT(enable, opcl, io, driver, flags) \
    { (enable?(opcl):no_dev), (enable?(io):0), \
      (enable?(driver):0), (flags) },
#define NC(x) (NR_CTRLRS >= (x))

/* The order of the entries here determines the mapping between major device
 * numbers and tasks. The first entry (major device 0) is not used. The
 * next entry is major device 1, etc. Character and block devices can be
 * intermixed at random. The ordering determines the device numbers in /dev/.
 * Note that FS knows the device number of /dev/ram/ to load the RAM disk.
 * Also note that the major device numbers used in /dev/ are NOT the same as
 * the process numbers of the device drivers.
 */
/*
Driver enabled      Open/Cls  I/O      Driver #      Flags Device  File
-----
*/
struct dmap dmap[NR_DEVICES]; /* actual map */
PRIVATE struct dmap init_dmap[] = {
    DT(1, no_dev, 0, 0, 0) /* 0 = not used */
    DT(1, gen_opcl, gen_io, MEM_PROC_NR, 0) /* 1 = /dev/mem */
    DT(0, no_dev, 0, 0, DMAP_MUTABLE) /* 2 = /dev/fd0 */
    DT(0, no_dev, 0, 0, DMAP_MUTABLE) /* 3 = /dev/c0 */
    DT(1, tty_opcl, gen_io, TTY_PROC_NR, 0) /* 4 = /dev/tty00 */
    DT(1, cttty_opcl, cttty_io, TTY_PROC_NR, 0) /* 5 = /dev/tty */
    DT(0, no_dev, 0, NONE, DMAP_MUTABLE) /* 6 = /dev/lp */

#ifdef MACHINE == IBM_PC
    DT(1, no_dev, 0, 0, DMAP_MUTABLE) /* 7 = /dev/ip */
    DT(0, no_dev, 0, NONE, DMAP_MUTABLE) /* 8 = /dev/c1 */
    DT(0, 0, 0, 0, DMAP_MUTABLE) /* 9 = not used */
    DT(0, no_dev, 0, 0, DMAP_MUTABLE) /* 10 = /dev/c2 */
    DT(0, 0, 0, 0, DMAP_MUTABLE) /* 11 = not used */
    DT(0, no_dev, 0, NONE, DMAP_MUTABLE) /* 12 = /dev/c3 */
    DT(0, no_dev, 0, NONE, DMAP_MUTABLE) /* 13 = /dev/audio */
    DT(0, no_dev, 0, NONE, DMAP_MUTABLE) /* 14 = /dev/mixer */
    DT(1, gen_opcl, gen_io, LOG_PROC_NR, 0) /* 15 = /dev/klog */
    DT(0, no_dev, 0, NONE, DMAP_MUTABLE) /* 16 = /dev/random */
    DT(0, no_dev, 0, NONE, DMAP_MUTABLE) /* 17 = /dev/cmos */
#endif
#endif /* IBM_PC */
};

/*=====
 * do_devctl
 *=====*/
PUBLIC int do_devctl()
{
    if (!super_user)
    {
        printf("FS: unauthorized call of do_devctl by proc %d\n",
            who_e);
        return(EPERM); /* only su (should be only RS or some drivers)
                        * may call do_devctl.
                        */
    }
    return fs_devctl(m_in.ctl_req, m_in.dev_nr, m_in.driver_nr,
        m_in.dev_style, m_in.m_force);
}

```

```

/*=====
 *
 *                               fs_devctl
 *=====*/
PUBLIC int fs_devctl(req, dev, proc_nr_e, style, force)
int req;
int dev;
int proc_nr_e;
int style;
int force;
{
    int result, proc_nr_n;

    switch(req) {
    case DEV_MAP:
        if (!force)
        {
            /* Check process number of new driver. */
            if (isokendpt(proc_nr_e, &proc_nr_n) != OK)
                return(EINVAL);
        }

        /* Try to update device mapping. */
        result = map_driver(dev, proc_nr_e, style, force);
        if (result == OK)
        {
            /* If a driver has completed its exec(), it can be announced to be
             * up.
             */
            if(force || fproc[proc_nr_n].fp_execeed) {
                dev_up(dev);
            } else {
                dmap[dev].dmap_flags |= DMAP_BABY;
            }
        }
        break;
    case DEV_UNMAP:
        result = map_driver(dev, NONE, 0, 0);
        break;
    default:
        result = EINVAL;
    }
    return(result);
}

/*=====
 *
 *                               map_driver
 *=====*/
PUBLIC int map_driver(major, proc_nr_e, style, force)
int major;
int proc_nr_e;
int style;
int force;
{
    /* Set a new device driver mapping in the dmap table. Given that correct
     * arguments are given, this only works if the entry is mutable and the
     * current driver is not busy. If the proc_nr is set to NONE, we're supposed
     * to unmap it.
     *
     * Normal error codes are returned so that this function can be used from
     * a system call that tries to dynamically install a new driver.
     */
    struct dmap *dp;
    int proc_nr_n;

    /* Get pointer to device entry in the dmap table. */
    if (major < 0 || major >= NR_DEVICES) return(ENODEV);
    dp = &dmap[major];

    /* Check if we're supposed to unmap it. If so, do it even
     * if busy or unmutable, as unmap is called when driver has
     * exited.
     */
    if(proc_nr_e == NONE) {
        dp->dmap_opcl = no_dev;
    }
}

```

```

    dp->dmap_io = no_dev_io;
    dp->dmap_driver = NONE;
    dp->dmap_flags = DMAP_MUTABLE; /* When gone, not busy or reserved. */
    return(OK);
}

/* See if updating the entry is allowed. */
if (! (dp->dmap_flags & DMAP_MUTABLE)) return(EPERM);
if (dp->dmap_flags & DMAP_BUSY) return(EBUSY);

if (!force)
{
    /* Check process number of new driver. */
    if (isokendpt(proc_nr_e, &proc_nr_n) != OK)
        return(EINVAL);
}

/* Try to update the entry. */
switch (style) {
case STYLE_DEV:      dp->dmap_opcl = gen_opcl;      break;
case STYLE_TTY:      dp->dmap_opcl = tty_opcl;      break;
case STYLE_CLONE:    dp->dmap_opcl = clone_opcl;    break;
default:             return(EINVAL);
}
dp->dmap_io = gen_io;
dp->dmap_driver = proc_nr_e;

return(OK);
}

/*=====
 *                               dmap_unmap_by_endpt                               *
 *=====*/
PUBLIC void dmap_unmap_by_endpt(int proc_nr_e)
{
    int i, r;
    for (i=0; i<NR_DEVICES; i++)
        if(dmap[i].dmap_driver && dmap[i].dmap_driver == proc_nr_e)
            if((r=map_driver(i, NONE, 0, 0)) != OK)
                printf("FS: unmap of p %d / d %d failed: %d\n", proc_nr_e, i, r);

    return;
}

/*=====
 *                               build_dmap                               *
 *=====*/
PUBLIC void build_dmap()
{
    /* Initialize the table with all device <-> driver mappings. Then, map
    * the boot driver to a controller and update the dmap table to that
    * selection. The boot driver and the controller it handles are set at
    * the boot monitor.
    */
    int i;
    struct dmap *dp;

    /* Build table with device <-> driver mappings. */
    for (i=0; i<NR_DEVICES; i++) {
        dp = &dmap[i];
        if (i < sizeof(init_dmap)/sizeof(struct dmap) &&
            init_dmap[i].dmap_opcl != no_dev) { /* a preset driver */
            dp->dmap_opcl = init_dmap[i].dmap_opcl;
            dp->dmap_io = init_dmap[i].dmap_io;
            dp->dmap_driver = init_dmap[i].dmap_driver;
            dp->dmap_flags = init_dmap[i].dmap_flags;
        } else { /* no default */
            dp->dmap_opcl = no_dev;
            dp->dmap_io = no_dev_io;
            dp->dmap_driver = NONE;
            dp->dmap_flags = DMAP_MUTABLE;
        }
    }
}

```



```

#if 0
/* Get settings of 'controller' and 'driver' at the boot monitor. */
if ((s = env_get_param("label", driver, sizeof(driver))) != OK)
    panic(__FILE__, "couldn't get boot monitor parameter 'driver'", s);
if ((s = env_get_param("controller", controller, sizeof(controller))) != OK)
    panic(__FILE__, "couldn't get boot monitor parameter 'controller'", s);

/* Determine major number to map driver onto. */
if (controller[0] == 'f' && controller[1] == 'd') {
    major = FLOPPY_MAJOR;
}
else if (controller[0] == 'c' && isdigit(controller[1])) {
    if ((nr = (unsigned) atoi(&controller[1])) > NR_CTRLRS)
        panic(__FILE__, "monitor 'controller' maximum 'c#' is", NR_CTRLRS);
    major = CTRLR(nr);
}
else {
    panic(__FILE__, "monitor 'controller' syntax is 'c#' of 'fd'", NO_NUM);
}

/* Now try to set the actual mapping and report to the user. */
if ((s=map_driver(major, DRVR_PROC_NR, STYLE_DEV)) != OK)
    panic(__FILE__, "map_driver failed", s);
printf("Boot medium driver: %s driver mapped onto controller %s.\n",
    driver, controller);
#endif
}

/*=====
*                               dmap_driver_match                               *
*=====*/
PUBLIC int dmap_driver_match(int proc, int major)
{
    if (major < 0 || major >= NR_DEVICES) return(0);
    if(dmap[major].dmap_driver != NONE && dmap[major].dmap_driver == proc)
        return 1;
    return 0;
}

/*=====
*                               dmap_endpt_up                               *
*=====*/
PUBLIC void dmap_endpt_up(int proc_e)
{
    int i;
    for (i=0; i<NR_DEVICES; i++) {
        if(dmap[i].dmap_driver != NONE
            && dmap[i].dmap_driver == proc_e
            && (dmap[i].dmap_flags & DMAP_BABY)) {
            dmap[i].dmap_flags &= ~DMAP_BABY;
            dev_up(i);
        }
    }
    return;
}

```

```
/* This is the filp table. It is an intermediary between file descriptors and
 * inodes. A slot is free if filp_count == 0.
 */

EXTERN struct filp {
    mode_t filp_mode;           /* RW bits, telling how file is opened */
    int filp_flags;             /* flags from open and fcntl */
    int filp_count;             /* how many file descriptors share this slot? */
    struct inode *filp_ino;      /* pointer to the inode */
    off_t filp_pos;             /* file position */

    /* the following fields are for select() and are owned by the generic
     * select() code (i.e., fd-type-specific select() code can't touch these).
     */
    int filp_selectors;         /* select()ing processes blocking on this fd */
    int filp_select_ops;        /* interested in these SEL_* operations */

    /* following are for fd-type-specific select() */
    int filp_pipe_select_ops;
} filp[NR_FILPS];

#define FILP_CLOSED      0      /* filp_mode: associated device closed */

#define NIL_FILP (struct filp *) 0 /* indicates absence of a filp slot */
```

```

/* This file contains the procedures that manipulate file descriptors.
 *
 * The entry points into this file are
 *   get_fd:    look for free file descriptor and free filp slots
 *   get_filp:  look up the filp entry for a given file descriptor
 *   find_filp: find a filp slot that points to a given inode
 *   inval_filp: invalidate a filp and associated fd's, only let close()
 *                happen on it
 */

#include <sys/select.h>

#include "fs.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"

/*=====
 *                               get_fd                               *
 *=====*/
PUBLIC int get_fd(int start, mode_t bits, int *k, struct filp **fpt)
{
/* Look for a free file descriptor and a free filp slot.  Fill in the mode word
 * in the latter, but don't claim either one yet, since the open() or creat()
 * may yet fail.
 */

    register struct filp *f;
    register int i;

    *k = -1;                                /* we need a way to tell if file desc found */

    /* Search the fproc fp_filp table for a free file descriptor. */
    for (i = start; i < OPEN_MAX; i++) {
        if (fp->fp_filp[i] == NIL_FILP && !FD_ISSET(i, &fp->fp_filp_inuse)) {
            /* A file descriptor has been located. */
            *k = i;
            break;
        }
    }

    /* Check to see if a file descriptor has been found. */
    if (*k < 0) return(EMFILE); /* this is why we initialized k to -1 */

    /* Now that a file descriptor has been found, look for a free filp slot. */
    for (f = &filp[0]; f < &filp[NR_FILPS]; f++) {
        if (f->filp_count == 0) {
            f->filp_mode = bits;
            f->filp_pos = 0L;
            f->filp_selectors = 0;
            f->filp_select_ops = 0;
            f->filp_pipe_select_ops = 0;
            f->filp_flags = 0;
            *fpt = f;
            return(OK);
        }
    }

    /* If control passes here, the filp table must be full.  Report that back. */
    return(ENFILE);
}

/*=====
 *                               get_filp                               *
 *=====*/
PUBLIC struct filp *get_filp(fild)
int fild;                                /* file descriptor */
{
/* See if 'fild' refers to a valid file descr.  If so, return its filp ptr. */

    return get_filp2(fp, fild);
}

/*=====

```

```

*                               get_filp2                               *
*=====*/
PUBLIC struct filp *get_filp2(rfp, fild)
register struct fproc *rfp;
int fild;                               /* file descriptor */
{
/* See if 'fild' refers to a valid file descr.  If so, return its filp ptr. */

    err_code = EBADF;
    if (fild < 0 || fild >= OPEN_MAX ) return(NIL_FILP);
    return(rfp->fp_filp[fild]); /* may also be NIL_FILP */
}

/*=====*
*                               find_filp                               *
*=====*/
PUBLIC struct filp *find_filp(register struct inode *rip, mode_t bits)
{
/* Find a filp slot that refers to the inode 'rip' in a way as described
 * by the mode bit 'bits'. Used for determining whether somebody is still
 * interested in either end of a pipe. Also used when opening a FIFO to
 * find partners to share a filp field with (to shared the file position).
 * Like 'get_fd' it performs its job by linear search through the filp table.
*/

    register struct filp *f;

    for (f = &filp[0]; f < &filp[NR_FILPS]; f++) {
        if (f->filp_count != 0 && f->filp_ino == rip && (f->filp_mode & bits)){
            return(f);
        }
    }

    /* If control passes here, the filp wasn't there. Report that back. */
    return(NIL_FILP);
}

/*=====*
*                               inval_filp                               *
*=====*/
PUBLIC int inval_filp(struct filp *fp)
{
    int f, fd, n = 0;
    for(f = 0; f < NR_PROCS; f++) {
        if(fproc[f].fp_pid == PID_FREE) continue;
        for(fd = 0; fd < OPEN_MAX; fd++) {
            if(fproc[f].fp_filp[fd] && fproc[f].fp_filp[fd] == fp) {
                fproc[f].fp_filp[fd] = NIL_FILP;
                n++;
            }
        }
    }

    return n;
}

```

```

#include <sys/select.h>

/* This is the per-process information. A slot is reserved for each potential
 * process. Thus NR_PROCS must be the same as in the kernel. It is not
 * possible or even necessary to tell when a slot is free here.
 */
EXTERN struct fproc {
    mode_t fp_umask;           /* mask set by umask system call */
    struct inode *fp_workdir;  /* pointer to working directory's inode */
    struct inode *fp_rootdir;  /* pointer to current root dir (see chroot) */
    struct filp *fp_filp[OPEN_MAX]; /* the file descriptor table */
    fd_set fp_filp_inuse;      /* which fd's are in use? */
    uid_t fp_realuid;          /* real user id */
    uid_t fp_effuid;           /* effective user id */
    gid_t fp_realgid;          /* real group id */
    gid_t fp_effgid;           /* effective group id */
    dev_t fp_tty;              /* major/minor of controlling tty */
    int fp_fd;                 /* place to save fd if rd/wr can't finish */
    char *fp_buffer;           /* place to save buffer if rd/wr can't finish */
    int fp_nbytes;             /* place to save bytes if rd/wr can't finish */
    int fp_cum_io_partial;     /* partial byte count if rd/wr can't finish */
    char fp_suspended;         /* set to indicate process hanging */
    char fp_revived;           /* set to indicate process being revived */
    int fp_task;               /* which task is proc suspended on */
    char fp_sesldr;            /* true if proc is a session leader */
    char fp_execcd;            /* true if proc has exec()ced after fork */
    pid_t fp_pid;              /* process id */
    long fp_cloexec;           /* bit map for POSIX Table 6-2 FD_CLOEXEC */
    int fp_endpoint;           /* kernel endpoint number of this process */
} fproc[NR_PROCS];

/* Field values. */
#define NOT_SUSPENDED 0 /* process is not suspended on pipe or task */
#define SUSPENDED 1 /* process is suspended on pipe or task */
#define NOT_REVIVING 0 /* process is not being revived */
#define REVIVING 1 /* process is being revived from suspension */
#define PID_FREE 0 /* process slot free */

/* Check if process number is acceptable - includes system processes. */
#define isokprocnr(n) ((unsigned)((n)+NR_TASKS) < NR_PROCS + NR_TASKS)

```

```
/* This is the master header for fs. It includes some other files
 * and defines the principal constants.
 */
#define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
#define _MINIX             1    /* tell headers to include MINIX stuff */
#define _SYSTEM            1    /* tell headers that this is the kernel */

#define VERBOSE            0    /* show messages during initialization? */

/* The following are so basic, all the *.c files get them automatically. */
#include <minix/config.h>      /* MUST be first */
#include <ansi.h>              /* MUST be second */
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>
#include <minix/dmap.h>

#include <limits.h>
#include <errno.h>

#include <minix/syslib.h>
#include <minix/sysutil.h>

#include "const.h"
#include "type.h"
#include "proto.h"
#include "glo.h"
```

```
/* EXTERN should be extern except for the table file */
#ifdef _TABLE
#undef EXTERN
#define EXTERN
#endif

/* File System global variables */
EXTERN struct fproc *fp; /* pointer to caller's fproc struct */
EXTERN int super_user; /* 1 if caller is super_user, else 0 */
EXTERN int susp_count; /* number of procs suspended on pipe */
EXTERN int nr_locks; /* number of locks currently in place */
EXTERN int reviving; /* number of pipe processes to be revived */
EXTERN off_t rdahedpos; /* position to read ahead */
EXTERN struct inode *rdahed_inode; /* pointer to inode to read ahead */
EXTERN Dev_t root_dev; /* device number of the root device */
EXTERN time_t boottime; /* time in seconds at system boot */

/* The parameters of the call are kept here. */
EXTERN message m_in; /* the input message itself */
EXTERN message m_out; /* the output message used for reply */
EXTERN int who_p, who_e; /* caller's proc number, endpoint */
EXTERN int call_nr; /* system call number */
EXTERN char user_path[PATH_MAX]; /* storage for user path name */

/* The following variables are used for returning results to the caller. */
EXTERN int err_code; /* temporary storage for error number */
EXTERN int rdwt_err; /* status of last disk i/o request */

/* Data initialized elsewhere. */
extern _PROTOTYPE (int (*call_vec[]), (void) ); /* sys call table */
extern char dot1[2]; /* dot1 (&dot1[0]) and dot2 (&dot2[0]) have a special */
extern char dot2[3]; /* meaning to search_dir: no access permission check. */
```

```

/* This file manages the inode table. There are procedures to allocate and
 * deallocate inodes, acquire, erase, and release them, and read and write
 * them from the disk.
 *
 * The entry points into this file are
 * get_inode: search inode table for a given inode; if not there,
 * read it
 * put_inode: indicate that an inode is no longer needed in memory
 * alloc_inode: allocate a new, unused inode
 * wipe_inode: erase some fields of a newly allocated inode
 * free_inode: mark an inode as available for a new file
 * update_times: update atime, ctime, and mtime
 * rw_inode: read a disk block and extract an inode, or corresp. write
 * old_icopy: copy to/from in-core inode struct and disk inode (V1.x)
 * new_icopy: copy to/from in-core inode struct and disk inode (V2.x)
 * dup_inode: indicate that someone else is using an inode table entry
 */

#include "fs.h"
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "super.h"

FORWARD _PROTOTYPE( void old_icopy, (struct inode *rip, dl_inode *dip,
                                     int direction, int norm));
FORWARD _PROTOTYPE( void new_icopy, (struct inode *rip, d2_inode *dip,
                                     int direction, int norm));

/*=====
 *
 * get_inode
 *=====*/
PUBLIC struct inode *get_inode(dev, numb)
dev_t dev; /* device on which inode resides */
int numb; /* inode number (ANSI: may not be unshort) */
{
/* Find a slot in the inode table, load the specified inode into it, and
 * return a pointer to the slot. If 'dev' == NO_DEV, just return a free slot.
 */

register struct inode *rip, *xp;

/* Search the inode table both for (dev, numb) and a free slot. */
xp = NIL_INODE;
for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++) {
    if (rip->i_count > 0) { /* only check used slots for (dev, numb) */
        if (rip->i_dev == dev && rip->i_num == numb) {
            /* This is the inode that we are looking for. */
            rip->i_count++;
            return(rip); /* (dev, numb) found */
        }
    } else {
        xp = rip; /* remember this free slot for later */
    }
}

/* Inode we want is not currently in use. Did we find a free slot? */
if (xp == NIL_INODE) { /* inode table completely full */
    err_code = ENFILE;
    return(NIL_INODE);
}

/* A free inode slot has been located. Load the inode into it. */
xp->i_dev = dev;
xp->i_num = numb;
xp->i_count = 1;
if (dev != NO_DEV) rw_inode(xp, READING); /* get inode from disk */
xp->i_update = 0; /* all the times are initially up-to-date */

return(xp);
}

/*=====

```



```

*                                     put_inode                                     *
*=====*/
PUBLIC void put_inode(rip)
register struct inode *rip;      /* pointer to inode to be released */
{
/* The caller is no longer using this inode.  If no one else is using it either
* write it back to the disk immediately.  If it has no links, truncate it and
* return it to the pool of available inodes.
*/

if (rip == NIL_INODE) return; /* checking here is easier than in caller */
if (--rip->i_count == 0) {      /* i_count == 0 means no one is using it now */
    if (rip->i_nlinks == 0) {
        /* i_nlinks == 0 means free the inode. */
        truncate_inode(rip, 0); /* return all the disk blocks */
        rip->i_mode = I_NOT_ALLOC; /* clear I_TYPE field */
        rip->i_dirt = DIRTY;
        free_inode(rip->i_dev, rip->i_num);
    } else {
        if (rip->i_pipe == I_PIPE) truncate_inode(rip, 0);
    }
    rip->i_pipe = NO_PIPE; /* should always be cleared */
    if (rip->i_dirt == DIRTY) rw_inode(rip, WRITING);
}
}

/*=====*/
*                                     alloc_inode                                 *
*=====*/
PUBLIC struct inode *alloc_inode(dev_t dev, mode_t bits)
{
/* Allocate a free inode on 'dev', and return a pointer to it. */

register struct inode *rip;
register struct super_block *sp;
int major, minor, inumb;
bit_t b;

sp = get_super(dev); /* get pointer to super_block */
if (sp->s_rd_only) { /* can't allocate an inode on a read only device. */
    err_code = EROFS;
    return(NIL_INODE);
}

/* Acquire an inode from the bit map. */
b = alloc_bit(sp, IMAP, sp->s_isearch);
if (b == NO_BIT) {
    err_code = ENFILE;
    major = (int) (sp->s_dev >> MAJOR) & BYTE;
    minor = (int) (sp->s_dev >> MINOR) & BYTE;
    printf("Out of i-nodes on %sdevice %d/%d\n",
        sp->s_dev == root_dev ? "root" : "", major, minor);
    return(NIL_INODE);
}
sp->s_isearch = b; /* next time start here */
inumb = (int) b; /* be careful not to pass unshort as param */

/* Try to acquire a slot in the inode table. */
if ((rip = get_inode(NO_DEV, inumb)) == NIL_INODE) {
    /* No inode table slots available. Free the inode just allocated. */
    free_bit(sp, IMAP, b);
} else {
    /* An inode slot is available. Put the inode just allocated into it. */
    rip->i_mode = bits; /* set up RWX bits */
    rip->i_nlinks = 0; /* initial no links */
    rip->i_uid = fp->fp_effuid; /* file's uid is owner's */
    rip->i_gid = fp->fp_effgid; /* ditto group id */
    rip->i_dev = dev; /* mark which device it is on */
    rip->i_ndzones = sp->s_ndzones; /* number of direct zones */
    rip->i_nindirs = sp->s_nindirs; /* number of indirect zones per blk */
    rip->i_sp = sp; /* pointer to super block */

    /* Fields not cleared already are cleared in wipe_inode(). They have
    * been put there because truncate() needs to clear the same fields if

```

```

        * the file happens to be open while being truncated. It saves space
        * not to repeat the code twice.
        */
        wipe_inode(rip);
    }

    return(rip);
}

/*=====
 *                               wipe_inode                               *
 *=====*/
PUBLIC void wipe_inode(rip)
register struct inode *rip;    /* the inode to be erased */
{
    /* Erase some fields in the inode. This function is called from alloc_inode()
     * when a new inode is to be allocated, and from truncate(), when an existing
     * inode is to be truncated.
     */

    register int i;

    rip->i_size = 0;
    rip->i_update = ATIME | CTIME | MTIME;    /* update all times later */
    rip->i_dirt = DIRTY;
    for (i = 0; i < V2_NR_TZONES; i++) rip->i_zone[i] = NO_ZONE;
}

/*=====
 *                               free_inode                               *
 *=====*/
PUBLIC void free_inode(dev, inumb)
dev_t dev;    /* on which device is the inode */
ino_t inumb;    /* number of inode to be freed */
{
    /* Return an inode to the pool of unallocated inodes. */

    register struct super_block *sp;
    bit_t b;

    /* Locate the appropriate super_block. */
    sp = get_super(dev);
    if (inumb <= 0 || inumb > sp->s_ninodes) return;
    b = inumb;
    free_bit(sp, IMAP, b);
    if (b < sp->s_isearch) sp->s_isearch = b;
}

/*=====
 *                               update_times                               *
 *=====*/
PUBLIC void update_times(rip)
register struct inode *rip;    /* pointer to inode to be read/written */
{
    /* Various system calls are required by the standard to update atime, ctime,
     * or mtime. Since updating a time requires sending a message to the clock
     * task--an expensive business--the times are marked for update by setting
     * bits in i_update. When a stat, fstat, or sync is done, or an inode is
     * released, update_times() may be called to actually fill in the times.
     */

    time_t cur_time;
    struct super_block *sp;

    sp = rip->i_sp;    /* get pointer to super block. */
    if (sp->s_rd_only) return;    /* no updates for read-only file systems */

    cur_time = clock_time();
    if (rip->i_update & ATIME) rip->i_atime = cur_time;
    if (rip->i_update & CTIME) rip->i_ctime = cur_time;
    if (rip->i_update & MTIME) rip->i_mtime = cur_time;
    rip->i_update = 0;    /* they are all up-to-date now */
}

```

```

/*=====
 *
 *                               rw_inode
 *=====*/
PUBLIC void rw_inode(rip, rw_flag)
register struct inode *rip;    /* pointer to inode to be read/written */
int rw_flag;                  /* READING or WRITING */
{
/* An entry in the inode table is to be copied to or from the disk. */

register struct buf *bp;
register struct super_block *sp;
dl_inode *dip;
d2_inode *dip2;
block_t b, offset;

/* Get the block where the inode resides. */
sp = get_super(rip->i_dev);    /* get pointer to super block */
rip->i_sp = sp;                 /* inode must contain super block pointer */
offset = sp->s_imap_blocks + sp->s_zmap_blocks + 2;
b = (block_t) (rip->i_num - 1)/sp->s_inodes_per_block + offset;
bp = get_block(rip->i_dev, b, NORMAL);
dip = bp->b_v1_ino + (rip->i_num - 1) % V1_INODES_PER_BLOCK;
dip2 = bp->b_v2_ino + (rip->i_num - 1) %
        V2_INODES_PER_BLOCK(sp->s_block_size);

/* Do the read or write. */
if (rw_flag == WRITING) {
    if (rip->i_update) update_times(rip);    /* times need updating */
    if (sp->s_rd_only == FALSE) bp->b_dirt = DIRTY;
}

/* Copy the inode from the disk block to the in-core table or vice versa.
 * If the fourth parameter below is FALSE, the bytes are swapped.
 */
if (sp->s_version == V1)
    old_icopy(rip, dip, rw_flag, sp->s_native);
else
    new_icopy(rip, dip2, rw_flag, sp->s_native);

put_block(bp, INODE_BLOCK);
rip->i_dirt = CLEAN;
}

/*=====
 *
 *                               old_icopy
 *=====*/
PRIVATE void old_icopy(rip, dip, direction, norm)
register struct inode *rip;    /* pointer to the in-core inode struct */
register dl_inode *dip;        /* pointer to the dl_inode inode struct */
int direction;                /* READING (from disk) or WRITING (to disk) */
int norm;                      /* TRUE = do not swap bytes; FALSE = swap */
{
/* The V1.x IBM disk, the V1.x 68000 disk, and the V2 disk (same for IBM and
 * 68000) all have different inode layouts. When an inode is read or written
 * this routine handles the conversions so that the information in the inode
 * table is independent of the disk structure from which the inode came.
 * The old_icopy routine copies to and from V1 disks.
 */

int i;

if (direction == READING) {
    /* Copy V1.x inode to the in-core table, swapping bytes if need be. */
    rip->i_mode = conv2(norm, (int) dip->dl_mode);
    rip->i_uid = conv2(norm, (int) dip->dl_uid);
    rip->i_size = conv4(norm, dip->dl_size);
    rip->i_mtime = conv4(norm, dip->dl_mtime);
    rip->i_atime = rip->i_mtime;
    rip->i_ctime = rip->i_mtime;
    rip->i_nlinks = dip->dl_nlinks;    /* 1 char */
    rip->i_gid = dip->dl_gid;          /* 1 char */
    rip->i_ndzones = V1_NR_DZONES;
    rip->i_nindirs = V1_INDIRECTS;

```

```

    for (i = 0; i < V1_NR_TZONES; i++)
        rip->i_zone[i] = conv2(norm, (int) dip->d1_zone[i]);
} else {
    /* Copying V1.x inode to disk from the in-core table. */
    dip->d1_mode = conv2(norm, (int) rip->i_mode);
    dip->d1_uid = conv2(norm, (int) rip->i_uid);
    dip->d1_size = conv4(norm, rip->i_size);
    dip->d1_mtime = conv4(norm, rip->i_mtime);
    dip->d1_nlinks = rip->i_nlinks; /* 1 char */
    dip->d1_gid = rip->i_gid; /* 1 char */
    for (i = 0; i < V1_NR_TZONES; i++)
        dip->d1_zone[i] = conv2(norm, (int) rip->i_zone[i]);
}

/*=====
*
* new_icopy
*=====*/
PRIVATE void new_icopy(rip, dip, direction, norm)
register struct inode *rip; /* pointer to the in-core inode struct */
register struct d2_inode *dip; /* pointer to the d2_inode struct */
int direction; /* READING (from disk) or WRITING (to disk) */
int norm; /* TRUE = do not swap bytes; FALSE = swap */

{
    /* Same as old_icopy, but to/from V2 disk layout. */

    int i;

    if (direction == READING) {
        /* Copy V2.x inode to the in-core table, swapping bytes if need be. */
        rip->i_mode = conv2(norm, dip->d2_mode);
        rip->i_uid = conv2(norm, dip->d2_uid);
        rip->i_nlinks = conv2(norm, dip->d2_nlinks);
        rip->i_gid = conv2(norm, dip->d2_gid);
        rip->i_size = conv4(norm, dip->d2_size);
        rip->i_atime = conv4(norm, dip->d2_atime);
        rip->i_ctime = conv4(norm, dip->d2_ctime);
        rip->i_mtime = conv4(norm, dip->d2_mtime);
        rip->i_ndzones = V2_NR_DZONES;
        rip->i_nindirs = V2_INDIRECTS(rip->i_sp->s_block_size);
        for (i = 0; i < V2_NR_TZONES; i++)
            rip->i_zone[i] = conv4(norm, (long) dip->d2_zone[i]);
    } else {
        /* Copying V2.x inode to disk from the in-core table. */
        dip->d2_mode = conv2(norm, rip->i_mode);
        dip->d2_uid = conv2(norm, rip->i_uid);
        dip->d2_nlinks = conv2(norm, rip->i_nlinks);
        dip->d2_gid = conv2(norm, rip->i_gid);
        dip->d2_size = conv4(norm, rip->i_size);
        dip->d2_atime = conv4(norm, rip->i_atime);
        dip->d2_ctime = conv4(norm, rip->i_ctime);
        dip->d2_mtime = conv4(norm, rip->i_mtime);
        for (i = 0; i < V2_NR_TZONES; i++)
            dip->d2_zone[i] = conv4(norm, (long) rip->i_zone[i]);
    }
}

/*=====
*
* dup_inode
*=====*/
PUBLIC void dup_inode(ip)
struct inode *ip; /* The inode to be duplicated. */
{
    /* This routine is a simplified form of get_inode() for the case where
    * the inode pointer is already known.
    */

    ip->i_count++;
}

```

```

/* Inode table. This table holds inodes that are currently in use. In some
 * cases they have been opened by an open() or creat() system call, in other
 * cases the file system itself needs the inode for one reason or another,
 * such as to search a directory for a path name.
 * The first part of the struct holds fields that are present on the
 * disk; the second part holds fields not present on the disk.
 * The disk inode part is also declared in "type.h" as 'd1_inode' for V1
 * file systems and 'd2_inode' for V2 file systems.
 */

EXTERN struct inode {
    mode_t i_mode; /* file type, protection, etc. */
    nlink_t i_nlinks; /* how many links to this file */
    uid_t i_uid; /* user id of the file's owner */
    gid_t i_gid; /* group number */
    off_t i_size; /* current file size in bytes */
    time_t i_atime; /* time of last access (V2 only) */
    time_t i_mtime; /* when was file data last changed */
    time_t i_ctime; /* when was inode itself changed (V2 only) */
    zone_t i_zone[V2_NR_TZONES]; /* zone numbers for direct, ind, and dbl ind */

    /* The following items are not present on the disk. */
    dev_t i_dev; /* which device is the inode on */
    ino_t i_num; /* inode number on its (minor) device */
    int i_count; /* # times inode used; 0 means slot is free */
    int i_ndzones; /* # direct zones (Vx_NR_DZONES) */
    int i_nindirs; /* # indirect zones per indirect block */
    struct super_block *i_sp; /* pointer to super block for inode's device */
    char i_dirt; /* CLEAN or DIRTY */
    char i_pipe; /* set to I_PIPE if pipe */
    char i_mount; /* this bit is set if file mounted on */
    char i_seek; /* set on LSEEK, cleared on READ/WRITE */
    char i_update; /* the ATIME, CTIME, and MTIME bits are here */
} inode[NR_INODES];

#define NIL_INODE (struct inode *) 0 /* indicates absence of inode slot */

/* Field values. Note that CLEAN and DIRTY are defined in "const.h" */
#define NO_PIPE 0 /* i_pipe is NO_PIPE if inode is not a pipe */
#define I_PIPE 1 /* i_pipe is I_PIPE if inode is a pipe */
#define NO_MOUNT 0 /* i_mount is NO_MOUNT if file not mounted on */
#define I_MOUNT 1 /* i_mount is I_MOUNT if file mounted on */
#define NO_SEEK 0 /* i_seek = NO_SEEK if last op was not SEEK */
#define ISEEK 1 /* i_seek = ISEEK if last op was SEEK */

```

```

/* This file handles the LINK and UNLINK system calls.  It also deals with
 * deallocating the storage used by a file when the last UNLINK is done to a
 * file and the blocks must be returned to the free block pool.
 *
 * The entry points into this file are
 * do_link:      perform the LINK system call
 * do_unlink:    perform the UNLINK and RMDIR system calls
 * do_rename:    perform the RENAME system call
 * do_truncate:  perform the TRUNCATE system call
 * do_ftruncate: perform the FTRUNCATE system call
 * truncate_inode: release the blocks associated with an inode up to a size
 * freesp_inode:  release a range of blocks without setting the size
 */

#include "fs.h"
#include <sys/stat.h>
#include <string.h>
#include <minix/com.h>
#include <minix/callnr.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

#define SAME 1000

FORWARD _PROTOTYPE( int remove_dir, (struct inode *rldirp, struct inode *rip,
                                     char dir_name[NAME_MAX]) );
FORWARD _PROTOTYPE( int unlink_file, (struct inode *dirp, struct inode *rip,
                                     char file_name[NAME_MAX]) );
FORWARD _PROTOTYPE( off_t nextblock, (off_t pos, int zonesize) );
FORWARD _PROTOTYPE( void zeroblock_half, (struct inode *i, off_t p, int l));
FORWARD _PROTOTYPE( void zeroblock_range, (struct inode *i, off_t p, off_t h));

/* Args to zeroblock_half() */
#define FIRST_HALF 0
#define LAST_HALF 1

/*=====
 *                               do_link                               *
 *=====*/
PUBLIC int do_link()
{
    /* Perform the link(name1, name2) system call. */

    struct inode *ip, *rip;
    register int r;
    char string[NAME_MAX];
    struct inode *new_ip;

    /* See if 'name' (file to be linked) exists. */
    if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
    if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);

    /* Check to see if the file has maximum number of links already. */
    r = OK;
    if (rip->i_nlinks >= (rip->i_sp->s_version == V1 ? CHAR_MAX : SHRT_MAX))
        r = EMLINK;

    /* Only super_user may link to directories. */
    if (r == OK)
        if ( (rip->i_mode & I_TYPE) == I_DIRECTORY && !super_user) r = EPERM;

    /* If error with 'name', return the inode. */
    if (r != OK) {
        put_inode(rip);
        return(r);
    }

    /* Does the final directory of 'name2' exist? */
    if (fetch_name(m_in.name2, m_in.name2_length, M1) != OK) {
        put_inode(rip);

```

```

        return(err_code);
    }
    if ( (ip = last_dir(user_path, string)) == NIL_INODE) r = err_code;

    /* If 'name2' exists in full (even if no space) set 'r' to error. */
    if (r == OK) {
        if ( (new_ip = advance(&ip, string)) == NIL_INODE) {
            r = err_code;
            if (r == ENOENT) r = OK;
        } else {
            put_inode(new_ip);
            r = EEXIST;
        }
    }

    /* Check for links across devices. */
    if (r == OK)
        if (rip->i_dev != ip->i_dev) r = EXDEV;

    /* Try to link. */
    if (r == OK)
        r = search_dir(ip, string, &rip->i_num, ENTER);

    /* If success, register the linking. */
    if (r == OK) {
        rip->i_nlinks++;
        rip->i_update |= CTIME;
        rip->i_dirt = DIRTY;
    }

    /* Done. Release both inodes. */
    put_inode(rip);
    put_inode(ip);
    return(r);
}

/*=====
 *                               do_unlink                               *
 *=====*/
PUBLIC int do_unlink()
{
    /* Perform the unlink(name) or rmdir(name) system call. The code for these two
    * is almost the same. They differ only in some condition testing. Unlink()
    * may be used by the superuser to do dangerous things; rmdir() may not.
    */

    register struct inode *rip;
    struct inode *rldirp;
    int r;
    char string[NAME_MAX];

    /* Get the last directory in the path. */
    if (fetch_name(m_in.name, m_in.name_length, M3) != OK) return(err_code);
    if ( (rldirp = last_dir(user_path, string)) == NIL_INODE)
        return(err_code);

    /* The last directory exists. Does the file also exist? */
    r = OK;
    if ( (rip = advance(&rldirp, string)) == NIL_INODE) r = err_code;

    /* If error, return inode. */
    if (r != OK) {
        put_inode(rldirp);
        return(r);
    }

    /* Do not remove a mount point. */
    if (rip->i_num == ROOT_INODE) {
        put_inode(rldirp);
        put_inode(rip);
        return(EBUSY);
    }

    /* Now test if the call is allowed, separately for unlink() and rmdir(). */

```

```

if (call_nr == UNLINK) {
    /* Only the su may unlink directories, but the su can unlink any dir. */
    if ( (rip->i_mode & I_TYPE) == I_DIRECTORY && !super_user) r = EPERM;

    /* Don't unlink a file if it is the root of a mounted file system. */
    if (rip->i_num == ROOT_INODE) r = EBUSY;

    /* Actually try to unlink the file; fails if parent is mode 0 etc. */
    if (r == OK) r = unlink_file(rldirp, rip, string);

} else {
    r = remove_dir(rldirp, rip, string); /* call is RMDIR */
}

/* If unlink was possible, it has been done, otherwise it has not. */
put_inode(rip);
put_inode(rldirp);
return(r);
}

/*=====
*                                     do_rename                                     *
*=====*/
PUBLIC int do_rename()
{
    /* Perform the rename(name1, name2) system call. */

    struct inode *old_dirp, *old_ip;          /* ptrs to old dir, file inodes */
    struct inode *new_dirp, *new_ip;          /* ptrs to new dir, file inodes */
    struct inode *new_superdirp, *next_new_superdirp;
    int r = OK;                               /* error flag; initially no error */
    int odir, ndir;                           /* TRUE iff {old/new} file is dir */
    int same_pdir;                            /* TRUE iff parent dirs are the same */
    char old_name[NAME_MAX], new_name[NAME_MAX];
    ino_t numb;
    int r1;

    /* See if 'name1' (existing file) exists. Get dir and file inodes. */
    if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
    if ( (old_dirp = last_dir(user_path, old_name)) == NIL_INODE) return(err_code);

    if ( (old_ip = advance(&old_dirp, old_name)) == NIL_INODE) r = err_code;

    /* See if 'name2' (new name) exists. Get dir and file inodes. */
    if (fetch_name(m_in.name2, m_in.name2_length, M1) != OK) r = err_code;
    if ( (new_dirp = last_dir(user_path, new_name)) == NIL_INODE) r = err_code;
    new_ip = advance(&new_dirp, new_name); /* not required to exist */

    if (old_ip != NIL_INODE)
        odir = ((old_ip->i_mode & I_TYPE) == I_DIRECTORY); /* TRUE iff dir */

    /* If it is ok, check for a variety of possible errors. */
    if (r == OK) {
        same_pdir = (old_dirp == new_dirp);

        /* The old inode must not be a superdirectory of the new last dir. */
        if (odir && !same_pdir) {
            dup_inode(new_superdirp = new_dirp);
            while (TRUE) { /* may hang in a file system loop */
                if (new_superdirp == old_ip) {
                    r = EINVAL;
                    break;
                }
                next_new_superdirp = advance(&new_superdirp, dot2);
                put_inode(new_superdirp);
                if (next_new_superdirp == new_superdirp)
                    break; /* back at system root directory */
                new_superdirp = next_new_superdirp;
                if (new_superdirp == NIL_INODE) {
                    /* Missing ".." entry. Assume the worst. */
                    r = EINVAL;
                    break;
                }
            }
        }
    }
}

```



```

        put_inode(new_superdirp);
    }

    /* The old or new name must not be . or .. */
    if (strcmp(old_name, ".")==0 || strcmp(old_name, "..")==0 ||
        strcmp(new_name, ".")==0 || strcmp(new_name, "..")==0) r = EINVAL;

    /* Both parent directories must be on the same device. */
    if (old_dirp->i_dev != new_dirp->i_dev) r = EXDEV;

    /* Parent dirs must be writable, searchable and on a writable device */
    if ((r1 = forbidden(old_dirp, W_BIT | X_BIT)) != OK ||
        (r1 = forbidden(new_dirp, W_BIT | X_BIT)) != OK) r = r1;

    /* Some tests apply only if the new path exists. */
    if (new_ip == NIL_INODE) {
        /* don't rename a file with a file system mounted on it. */
        if (old_ip->i_dev != old_dirp->i_dev) r = EXDEV;
        if (odir && new_dirp->i_nlinks >=
            (new_dirp->i_sp->s_version == V1 ? CHAR_MAX : SHRT_MAX) &&
            !same_pdir && r == OK) r = EMLINK;
    } else {
        if (old_ip == new_ip) r = SAME; /* old=new */

        /* has the old file or new file a file system mounted on it? */
        if (old_ip->i_dev != new_ip->i_dev) r = EXDEV;

        ndir = ((new_ip->i_mode & I_TYPE) == I_DIRECTORY); /* dir ? */
        if (odir == TRUE && ndir == FALSE) r = ENOTDIR;
        if (odir == FALSE && ndir == TRUE) r = EISDIR;
    }
}

/* If a process has another root directory than the system root, we might
 * "accidentally" be moving it's working directory to a place where it's
 * root directory isn't a super directory of it anymore. This can make
 * the function chroot useless. If chroot will be used often we should
 * probably check for it here.
 */

/* The rename will probably work. Only two things can go wrong now:
 * 1. being unable to remove the new file. (when new file already exists)
 * 2. being unable to make the new directory entry. (new file doesn't exists)
 * [directory has to grow by one block and cannot because the disk
 * is completely full].
 */
if (r == OK) {
    if (new_ip != NIL_INODE) {
        /* There is already an entry for 'new'. Try to remove it. */
        if (odir)
            r = remove_dir(new_dirp, new_ip, new_name);
        else
            r = unlink_file(new_dirp, new_ip, new_name);
    }
    /* if r is OK, the rename will succeed, while there is now an
     * unused entry in the new parent directory.
     */
}

if (r == OK) {
    /* If the new name will be in the same parent directory as the old one,
     * first remove the old name to free an entry for the new name,
     * otherwise first try to create the new name entry to make sure
     * the rename will succeed.
     */
    numb = old_ip->i_num; /* inode number of old file */

    if (same_pdir) {
        r = search_dir(old_dirp, old_name, (ino_t *) 0, DELETE);
        /* shouldn't go wrong. */
        if (r==OK) (void) search_dir(old_dirp, new_name, &numb, ENTER);
    } else {
        r = search_dir(new_dirp, new_name, &numb, ENTER);
        if (r == OK)

```

```

        (void) search_dir(old_dirp, old_name, (ino_t *) 0, DELETE);
    }
}
/* If r is OK, the ctime and mtime of old_dirp and new_dirp have been marked
 * for update in search_dir.
 */

if (r == OK && odir && !same_pdir) {
    /* Update the .. entry in the directory (still points to old_dirp). */
    numb = new_dirp->i_num;
    (void) unlink_file(old_ip, NIL_INODE, dot2);
    if (search_dir(old_ip, dot2, &numb, ENTER) == OK) {
        /* New link created. */
        new_dirp->i_nlinks++;
        new_dirp->i_dirt = DIRTY;
    }
}

/* Release the inodes. */
put_inode(old_dirp);
put_inode(old_ip);
put_inode(new_dirp);
put_inode(new_ip);
return(r == SAME ? OK : r);
}

/*=====
 *                               do_truncate                               *
 *=====*/
PUBLIC int do_truncate()
{
    /* truncate_inode() does the actual work of do_truncate() and do_ftruncate().
     * do_truncate() and do_ftruncate() have to get hold of the inode, either
     * by name or fd, do checks on it, and call truncate_inode() to do the
     * work.
     */
    int r;
    struct inode *rip;      /* pointer to inode to be truncated */

    if (fetch_name(m_in.m2_pl, m_in.m2_il, M1) != OK)
        return err_code;
    if ( (rip = eat_path(user_path)) == NIL_INODE)
        return err_code;
    if ( (rip->i_mode & I_TYPE) != I_REGULAR)
        r = EINVAL;
    else
        r = truncate_inode(rip, m_in.m2_ll);
    put_inode(rip);

    return r;
}

/*=====
 *                               do_ftruncate                               *
 *=====*/
PUBLIC int do_ftruncate()
{
    /* As with do_truncate(), truncate_inode() does the actual work. */
    struct filp *rfilp;
    if ( (rfilp = get_filp(m_in.m2_il)) == NIL_FILP)
        return err_code;
    if ( (rfilp->filp_ino->i_mode & I_TYPE) != I_REGULAR)
        return EINVAL;
    return truncate_inode(rfilp->filp_ino, m_in.m2_ll);
}

/*=====
 *                               truncate_inode                               *
 *=====*/
PUBLIC int truncate_inode(rip, newsize)
register struct inode *rip;      /* pointer to inode to be truncated */
off_t newsize;                  /* inode must become this size */
{
    /* Set inode to a certain size, freeing any zones no longer referenced

```

```

* and updating the size in the inode. If the inode is extended, the
* extra space is a hole that reads as zeroes.
*
* Nothing special has to happen to file pointers if inode is opened in
* O_APPEND mode, as this is different per fd and is checked when
* writing is done.
*/
zone_t zone_size;
int scale, file_type, waspipe;
dev_t dev;

file_type = rip->i_mode & I_TYPE; /* check to see if file is special */
if (file_type == I_CHAR_SPECIAL || file_type == I_BLOCK_SPECIAL)
    return EINVAL;
if(newsize > rip->i_sp->s_max_size) /* don't let inode grow too big */
    return EFBIG;

dev = rip->i_dev; /* device on which inode resides */
scale = rip->i_sp->s_log_zone_size;
zone_size = (zone_t) rip->i_sp->s_block_size << scale;

/* Pipes can shrink, so adjust size to make sure all zones are removed. */
waspipe = rip->i_pipe == I_PIPE; /* TRUE if this was a pipe */
if (waspipe) {
    if(newsize != 0)
        return EINVAL; /* Only truncate pipes to 0. */
    rip->i_size = PIPE_SIZE(rip->i_sp->s_block_size);
}

/* Free the actual space if relevant. */
if(newsize < rip->i_size)
    freesp_inode(rip, newsize, rip->i_size);

/* Next correct the inode size. */
if(!waspipe) rip->i_size = newsize;
else wipe_inode(rip); /* Pipes can only be truncated to 0. */
rip->i_dirt = DIRTY;

return OK;
}

/*=====
*                               freesp_inode                               *
*=====*/
PUBLIC int freesp_inode(rip, start, end)
register struct inode *rip; /* pointer to inode to be partly freed */
off_t start, end; /* range of bytes to free (end uninclusive) */
{
/* Cut an arbitrary hole in an inode. The caller is responsible for checking
* the reasonableness of the inode type of rip. The reason is this is that
* this function can be called for different reasons, for which different
* sets of inode types are reasonable. Adjusting the final size of the inode
* is to be done by the caller too, if wished.
*
* Consumers of this function currently are truncate_inode() (used to
* free indirect and data blocks for any type of inode, but also to
* implement the ftruncate() and truncate() system calls) and the F_FREESP
* fcntl().
*/
    off_t p, e;
    int zone_size, dev;

    if(end > rip->i_size) /* freeing beyond end makes no sense */
        end = rip->i_size;
    if(end <= start) /* end is uninclusive, so start<end */
        return EINVAL;
    zone_size = rip->i_sp->s_block_size << rip->i_sp->s_log_zone_size;
    dev = rip->i_dev; /* device on which inode resides */

    /* If freeing doesn't cross a zone boundary, then we may only zero
    * a range of the block.
    */
    if(start/zone_size == (end-1)/zone_size) {
        zeroblock_range(rip, start, end-start);
    }
}

```

```

    } else {
        /* First zero unused part of partly used blocks. */
        if(start%zone_size)
            zeroblock_half(rip, start, LAST_HALF);
        if(end%zone_size && end < rip->i_size)
            zeroblock_half(rip, end, FIRST_HALF);
    }

    /* Now completely free the completely unused blocks.
     * write_map() will free unused (double) indirect
     * blocks too. Converting the range to zone numbers avoids
     * overflow on p when doing e.g. 'p += zone_size'.
     */
    e = end/zone_size;
    if(end == rip->i_size && (end % zone_size)) e++;
    for(p = nextblock(start, zone_size)/zone_size; p < e; p++)
        write_map(rip, p*zone_size, NO_ZONE, WMAP_FREE);

    return OK;
}

/*=====
 *                               nextblock                               *
 *=====*/
PRIVATE off_t nextblock(pos, zone_size)
off_t pos;
int zone_size;
{
    /* Return the first position in the next block after position 'pos'
     * (unless this is the first position in the current block).
     * This can be done in one expression, but that can overflow pos.
     */
    off_t p;
    p = (pos/zone_size)*zone_size;
    if((pos % zone_size)) p += zone_size; /* Round up. */
    return p;
}

/*=====
 *                               zeroblock_half                          *
 *=====*/
PRIVATE void zeroblock_half(rip, pos, half)
struct inode *rip;
off_t pos;
int half;
{
    /* Zero the upper or lower 'half' of a block that holds position 'pos'.
     * half can be FIRST_HALF or LAST_HALF.
     *
     * FIRST_HALF: 0..pos-1 will be zeroed
     * LAST_HALF: pos..blocksize-1 will be zeroed
     */
    int offset, len;

    /* Offset of zeroing boundary. */
    offset = pos % rip->i_sp->s_block_size;

    if(half == LAST_HALF) {
        len = rip->i_sp->s_block_size - offset;
    } else {
        len = offset;
        pos -= offset;
        offset = 0;
    }

    zeroblock_range(rip, pos, len);
}

/*=====
 *                               zeroblock_range                          *
 *=====*/
PRIVATE void zeroblock_range(rip, pos, len)
struct inode *rip;
off_t pos;

```

```

off_t len;
{
/* Zero a range in a block.
 * This function is used to zero a segment of a block, either
 * FIRST_HALF of LAST_HALF.
 *
 */
    block_t b;
    struct buf *bp;
    off_t offset;

    if(!len) return; /* no zeroing to be done. */
    if( (b = read_map(rip, pos)) == NO_BLOCK) return;
    if( (bp = get_block(rip->i_dev, b, NORMAL)) == NIL_BUF)
        panic(__FILE__, "zeroblock_range: no block", NO_NUM);
    offset = pos % rip->i_sp->s_block_size;
    if(offset + len > rip->i_sp->s_block_size)
        panic(__FILE__, "zeroblock_range: len too long", len);
    memset(bp->b_data + offset, 0, len);
    bp->b_dirt = DIRTY;
    put_block(bp, FULL_DATA_BLOCK);
}

/*=====
 *
 *                               remove_dir
 *=====*/
PRIVATE int remove_dir(rldirp, rip, dir_name)
struct inode *rldirp;          /* parent directory */
struct inode *rip;              /* directory to be removed */
char dir_name[NAME_MAX];       /* name of directory to be removed */
{
/* A directory file has to be removed. Five conditions have to met:
 * - The file must be a directory
 * - The directory must be empty (except for . and ..)
 * - The final component of the path must not be . or ..
 * - The directory must not be the root of a mounted file system
 * - The directory must not be anybody's root/working directory
 */

    int r;
    register struct fproc *rfp;

/* search_dir checks that rip is a directory too. */
    if ((r = search_dir(rip, "", (ino_t *) 0, IS_EMPTY)) != OK) return r;

    if (strcmp(dir_name, ".") == 0 || strcmp(dir_name, "..") == 0) return(EINVAL);
    if (rip->i_num == ROOT_INODE) return(EBUSY); /* can't remove 'root' */

    for (rfp = &fproc[INIT_PROC_NR + 1]; rfp < &fproc[NR_PROCS]; rfp++)
        if (rfp->fp_pid != PID_FREE &&
            (rfp->fp_workdir == rip || rfp->fp_rootdir == rip))
            return(EBUSY); /* can't remove anybody's working dir */

/* Actually try to unlink the file; fails if parent is mode 0 etc. */
    if ((r = unlink_file(rldirp, rip, dir_name)) != OK) return r;

/* Unlink . and .. from the dir. The super user can link and unlink any dir,
 * so don't make too many assumptions about them.
 */
    (void) unlink_file(rip, NIL_INODE, dot1);
    (void) unlink_file(rip, NIL_INODE, dot2);
    return(OK);
}

/*=====
 *
 *                               unlink_file
 *=====*/
PRIVATE int unlink_file(dirp, rip, file_name)
struct inode *dirp;          /* parent directory of file */
struct inode *rip;           /* inode of file, may be NIL_INODE too. */
char file_name[NAME_MAX];    /* name of file to be removed */
{
/* Unlink 'file_name'; rip must be the inode of 'file_name' or NIL_INODE. */

```

```
ino_t numb;                /* inode number */
int r;

/* If rip is not NIL_INODE, it is used to get faster access to the inode. */
if (rip == NIL_INODE) {
    /* Search for file in directory and try to get its inode. */
    err_code = search_dir(dirp, file_name, &numb, LOOK_UP);
    if (err_code == OK) rip = get_inode(dirp->i_dev, (int) numb);
    if (err_code != OK || rip == NIL_INODE) return(err_code);
} else {
    dup_inode(rip);         /* inode will be returned with put_inode */
}

r = search_dir(dirp, file_name, (ino_t *) 0, DELETE);

if (r == OK) {
    rip->i_nlinks--;        /* entry deleted from parent's dir */
    rip->i_update |= CTIME;
    rip->i_dirt = DIRTY;
}

put_inode(rip);
return(r);
}
```

```

/* This file handles advisory file locking as required by POSIX.
 *
 * The entry points into this file are
 *   lock_op:   perform locking operations for FCNTL system call
 *   lock_revive: revive processes when a lock is released
 */

#include "fs.h"
#include <minix/com.h>
#include <fcntl.h>
#include <unistd.h>
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "lock.h"
#include "param.h"

/*=====
 *                               lock_op                               *
 *=====*/
PUBLIC int lock_op(f, req)
struct filp *f;
int req;
/* either F_SETLK or F_SETLKW */
{
/* Perform the advisory locking required by POSIX. */

    int r, ltype, i, conflict = 0, unlocking = 0;
    mode_t mo;
    off_t first, last;
    struct flock flock;
    vir_bytes user_flock;
    struct file_lock *flp, *flp2, *empty;

    /* Fetch the flock structure from user space. */
    user_flock = (vir_bytes) m_in.namel;
    r = sys_datacopy(who_e, (vir_bytes) user_flock,
        FS_PROC_NR, (vir_bytes) &flock, (phys_bytes) sizeof(flock));
    if (r != OK) return(EINVAL);

    /* Make some error checks. */
    ltype = flock.l_type;
    mo = f->filp_mode;
    if (ltype != F_UNLCK && ltype != F_RDLCK && ltype != F_WRLCK) return(EINVAL);
    if (req == F_GETLK && ltype == F_UNLCK) return(EINVAL);
    if ((f->filp_ino->i_mode & I_TYPE) != I_REGULAR) return(EINVAL);
    if (req != F_GETLK && ltype == F_RDLCK && (mo & R_BIT) == 0) return(EBADF);
    if (req != F_GETLK && ltype == F_WRLCK && (mo & W_BIT) == 0) return(EBADF);

    /* Compute the first and last bytes in the lock region. */
    switch (flock.l_whence) {
        case SEEK_SET: first = 0; break;
        case SEEK_CUR: first = f->filp_pos; break;
        case SEEK_END: first = f->filp_ino->i_size; break;
        default: return(EINVAL);
    }
    /* Check for overflow. */
    if (((long)flock.l_start > 0) && ((first + flock.l_start) < first))
        return(EINVAL);
    if (((long)flock.l_start < 0) && ((first + flock.l_start) > first))
        return(EINVAL);
    first = first + flock.l_start;
    last = first + flock.l_len - 1;
    if (flock.l_len == 0) last = MAX_FILE_POS;
    if (last < first) return(EINVAL);

    /* Check if this region conflicts with any existing lock. */
    empty = (struct file_lock *) 0;
    for (flp = &file_lock[0]; flp < &file_lock[NR_LOCKS]; flp++) {
        if (flp->lock_type == 0) {
            if (empty == (struct file_lock *) 0) empty = flp;
            continue; /* 0 means unused slot */
        }
        if (flp->lock_inode != f->filp_ino) continue; /* different file */
        if (last < flp->lock_first) continue; /* new one is in front */
    }
}

```

```

if (first > flp->lock_last) continue; /* new one is afterwards */
if (ltype == F_RDLCK && flp->lock_type == F_RDLCK) continue;
if (ltype != F_UNLCK && flp->lock_pid == fp->fp_pid) continue;

/* There might be a conflict. Process it. */
conflict = 1;
if (req == F_GETLK) break;

/* If we are trying to set a lock, it just failed. */
if (ltype == F_RDLCK || ltype == F_WRLCK) {
    if (req == F_SETLK) {
        /* For F_SETLK, just report back failure. */
        return(EAGAIN);
    } else {
        /* For F_SETLKW, suspend the process. */
        suspend(XLOCK);
        return(SUSPEND);
    }
}

/* We are clearing a lock and we found something that overlaps. */
unlocking = 1;
if (first <= flp->lock_first && last >= flp->lock_last) {
    flp->lock_type = 0; /* mark slot as unused */
    nr_locks--; /* number of locks is now 1 less */
    continue;
}

/* Part of a locked region has been unlocked. */
if (first <= flp->lock_first) {
    flp->lock_first = last + 1;
    continue;
}

if (last >= flp->lock_last) {
    flp->lock_last = first - 1;
    continue;
}

/* Bad luck. A lock has been split in two by unlocking the middle. */
if (nr_locks == NR_LOCKS) return(ENOLCK);
for (i = 0; i < NR_LOCKS; i++)
    if (file_lock[i].lock_type == 0) break;
flp2 = &file_lock[i];
flp2->lock_type = flp->lock_type;
flp2->lock_pid = flp->lock_pid;
flp2->lock_inode = flp->lock_inode;
flp2->lock_first = last + 1;
flp2->lock_last = flp->lock_last;
flp->lock_last = first - 1;
nr_locks++;
}
if (unlocking) lock_revive();

if (req == F_GETLK) {
    if (conflict) {
        /* GETLK and conflict. Report on the conflicting lock. */
        flock.l_type = flp->lock_type;
        flock.l_whence = SEEK_SET;
        flock.l_start = flp->lock_first;
        flock.l_len = flp->lock_last - flp->lock_first + 1;
        flock.l_pid = flp->lock_pid;
    } else {
        /* It is GETLK and there is no conflict. */
        flock.l_type = F_UNLCK;
    }
}

/* Copy the flock structure back to the caller. */
r = sys_datacopy(FS_PROC_NR, (vir_bytes) &flock,
    who_e, (vir_bytes) user_flock, (phys_bytes) sizeof(flock));
return(r);
}

```



```

if (ltype == F_UNLCK) return(OK);      /* unlocked a region with no locks */

/* There is no conflict. If space exists, store new lock in the table. */
if (empty == (struct file_lock *) 0) return(ENOLCK); /* table full */
empty->lock_type = ltype;
empty->lock_pid = fp->fp_pid;
empty->lock_inode = f->filp_ino;
empty->lock_first = first;
empty->lock_last = last;
nr_locks++;
return(OK);
}

/*=====
*                               lock_revive                               *
*=====*/
PUBLIC void lock_revive()
{
/* Go find all the processes that are waiting for any kind of lock and
* revive them all. The ones that are still blocked will block again when
* they run. The others will complete. This strategy is a space-time
* tradeoff. Figuring out exactly which ones to unblock now would take
* extra code, and the only thing it would win would be some performance in
* extremely rare circumstances (namely, that somebody actually used
* locking).
*/

int task;
struct fproc *fptr;

for (fptr = &fproc[INIT_PROC_NR + 1]; fptr < &fproc[NR_PROCS]; fptr++){
    if(fptr->fp_pid == PID_FREE) continue;
    task = -fptr->fp_task;
    if (fptr->fp_suspended == SUSPENDED && task == XLOCK) {
        revive(fptr->fp_endpoint, 0);
    }
}
}

```

```
/* This is the file locking table. Like the filp table, it points to the
 * inode table, however, in this case to achieve advisory locking.
 */
EXTERN struct file_lock {
    short lock_type;          /* F_RDLCK or F_WRLCK; 0 means unused slot */
    pid_t lock_pid;          /* pid of the process holding the lock */
    struct inode *lock_inode; /* pointer to the inode locked */
    off_t lock_first;        /* offset of first byte locked */
    off_t lock_last;         /* offset of last byte locked */
} file_lock[NR_LOCKS];
```

```

/* This file contains the main program of the File System.  It consists of
 * a loop that gets messages requesting work, carries out the work, and sends
 * replies.
 *
 * The entry points into this file are:
 *   main:      main program of the File System
 *   reply:     send a reply to a process after the requested work is done
 */

struct super_block;          /* proto.h needs to know this */

#include "fs.h"
#include <fcntl.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <sys/ioc_memory.h>
#include <sys/svrctl.h>
#include <sys/select.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <minix/keymap.h>
#include <minix/const.h>
#include <minix/endpoint.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

FORWARD _PROTOTYPE( void fs_init, (void) ) ;
FORWARD _PROTOTYPE( void get_work, (void) ) ;
FORWARD _PROTOTYPE( void init_root, (void) ) ;
FORWARD _PROTOTYPE( void service_pm, (void) ) ;

/*=====
 *                               main
 *=====*/
PUBLIC int main()
{
/* This is the main program of the file system.  The main loop consists of
 * three major activities: getting new work, processing the work, and sending
 * the reply.  This loop never terminates as long as the file system runs.
 */
    int error;

    fs_init();

    /* This is the main loop that gets work, processes it, and sends replies. */
    while (TRUE) {
        get_work();          /* sets who and call_nr */
        fp = &fproc[who_p];  /* pointer to proc table struct */
        super_user = (fp->fp_effuid == SU_UID ? TRUE : FALSE);  /* su? */

        if (who_e == PM_PROC_NR && call_nr != PROC_EVENT)
            printf("FS: strange, got message %d from PM\n", call_nr);

        /* Check for special control messages first. */
        if ((call_nr & NOTIFY_MESSAGE)) {
            if (call_nr == PROC_EVENT)
            {
                /* PM tries to get FS to do something */
                service_pm();
            }
            else if (call_nr == SYN_ALARM)
            {
                /* Alarm timer expired.  Used only for select().
                 * Check it.
                 */
                fs_expire_timers(m_in.NOTIFY_TIMESTAMP);
            }
        }
    }
}

```

```

        else
        {
            /* Device notifies us of an event. */
            dev_status(&m_in);
        }
        continue;
    }

    switch(call_nr)
    {
    case DEVCTL:
        error= do_devctl();
        if (error != SUSPEND) reply(who_e, error);
        break;

    default:
        /* Call the internal function that does the work. */
        if (call_nr < 0 || call_nr >= NCALLS) {
            error = ENOSYS;
            /* Not supposed to happen. */
            printf("FS, warning illegal %d system call by %d\n",
                call_nr, who_e);
        } else if (fp->fp_pid == PID_FREE) {
            error = ENOSYS;
            printf(
                "FS, bad process, who = %d, call_nr = %d, endpt1 = %d\n",
                who_e, call_nr, m_in.endpt1);
        } else {
            error = (*call_vec[call_nr])();
        }

        /* Copy the results back to the user and send reply. */
        if (error != SUSPEND) { reply(who_e, error); }
        if (rdahed_inode != NIL_INODE) {
            read_ahead(); /* do block read ahead */
        }
    }
}

return(OK); /* shouldn't come here */
}

/*=====
 *                               get_work                               *
 *=====*/
PRIVATE void get_work()
{
    /* Normally wait for new input.  However, if 'reviving' is
     * nonzero, a suspended process must be awakened.
     */
    register struct fproc *rp;

    if (reviving != 0) {
        /* Revive a suspended process. */
        for (rp = &fproc[0]; rp < &fproc[NR_PROCS]; rp++)
            if (rp->fp_pid != PID_FREE && rp->fp_revived == REVIVING) {
                who_p = (int)(rp - fproc);
                who_e = rp->fp_endpoint;
                call_nr = rp->fp_fd & BYTE;
                m_in.fd = (rp->fp_fd >>8) & BYTE;
                m_in.buffer = rp->fp_buffer;
                m_in.nbytes = rp->fp_nbytes;
                rp->fp_suspended = NOT_SUSPENDED; /*no longer hanging*/
                rp->fp_revived = NOT_REVIVING;
                reviving--;
                return;
            }
        panic(__FILE__, "get_work couldn't revive anyone", NO_NUM);
    }

    for(;;) {
        /* Normal case.  No one to revive. */
        if (receive(ANY, &m_in) != OK)
            panic(__FILE__, "fs receive error", NO_NUM);
        who_e = m_in.m_source;
    }
}

```

```

    who_p = _ENDPOINT_P(who_e);

    if(who_p < -NR_TASKS || who_p >= NR_PROCS)
        panic(__FILE__, "receive process out of range", who_p);
    if(who_p >= 0 && fproc[who_p].fp_endpoint == NONE) {
        printf("FS: ignoring request from %d, endpointless slot %d (%d)\n",
            m_in.m_source, who_p, m_in.m_type);
        continue;
    }
    if(who_p >= 0 && fproc[who_p].fp_endpoint != who_e) {
        printf("FS: receive endpoint inconsistent (%d, %d, %d).\n",
            who_e, fproc[who_p].fp_endpoint, who_e);
        panic(__FILE__, "FS: inconsistent endpoint ", NO_NUM);
        continue;
    }
    call_nr = m_in.m_type;
    return;
}
}

/*=====
 *                               buf_pool                               *
 *=====*/
PRIVATE void buf_pool(void)
{
    /* Initialize the buffer pool. */

    register struct buf *bp;

    bufs_in_use = 0;
    front = &buf[0];
    rear = &buf[NR_BUFS - 1];

    for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++) {
        bp->b_blocknr = NO_BLOCK;
        bp->b_dev = NO_DEV;
        bp->b_next = bp + 1;
        bp->b_prev = bp - 1;
    }
    buf[0].b_prev = NIL_BUF;
    buf[NR_BUFS - 1].b_next = NIL_BUF;

    for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++) bp->b_hash = bp->b_next;
    buf_hash[0] = front;
}

/*=====
 *                               reply                               *
 *=====*/
PUBLIC void reply(whom, result)
int whom; /* process to reply to */
int result; /* result of the call (usually OK or error #) */
{
    /* Send a reply to a user process. If the send fails, just ignore it. */
    int s;
    m_out.reply_type = result;
    s = send(whom, &m_out);
    if (s != OK) printf("FS: couldn't send reply %d to %d: %d\n",
        result, whom, s);
}

/*=====
 *                               fs_init                               *
 *=====*/
PRIVATE void fs_init()
{
    /* Initialize global variables, tables, etc. */
    register struct inode *rip;
    register struct fproc *rfp;
    message mess;
    int s;

    /* Initialize the process table with help of the process manager messages.

```

```

* Expect one message for each system process with its slot number and pid.
* When no more processes follow, the magic process number NONE is sent.
* Then, stop and synchronize with the PM.
*/
do {
    if (OK != (s=receive(PM_PROC_NR, &mess)))
        panic(__FILE__, "FS couldn't receive from PM", s);
    if (NONE == mess.PR_ENDPT) break;

    rfp = &fproc[mess.PR_SLOT];
    rfp->fp_pid = mess.PR_PID;
    rfp->fp_endpoint = mess.PR_ENDPT;
    rfp->fp_realuid = (uid_t) SYS_UID;
    rfp->fp_effuid = (uid_t) SYS_UID;
    rfp->fp_realgid = (gid_t) SYS_GID;
    rfp->fp_effgid = (gid_t) SYS_GID;
    rfp->fp_umask = ~0;

} while (TRUE);                                /* continue until process NONE */
mess.m_type = OK;                               /* tell PM that we succeeded */
s=send(PM_PROC_NR, &mess);                     /* send synchronization message */

/* All process table entries have been set. Continue with FS initialization.
* Certain relations must hold for the file system to work at all. Some
* extra block_size requirements are checked at super-block-read-in time.
*/
if (OPEN_MAX > 127) panic(__FILE__, "OPEN_MAX > 127", NO_NUM);
if (NR_BUFS < 6) panic(__FILE__, "NR_BUFS < 6", NO_NUM);
if (V1_INODE_SIZE != 32) panic(__FILE__, "V1 inode size != 32", NO_NUM);
if (V2_INODE_SIZE != 64) panic(__FILE__, "V2 inode size != 64", NO_NUM);
if (OPEN_MAX > 8 * sizeof(long))
    panic(__FILE__, "Too few bits in fp_cloexec", NO_NUM);

/* The following initializations are needed to let dev_opcl succeed.*/
fp = (struct fproc *) NULL;
who_e = who_p = FS_PROC_NR;

buf_pool();                                    /* initialize buffer pool */
build_dmap();                                  /* build device table and map boot driver */
init_root();                                  /* init root device and load super block */
init_select();                                /* init select() structures */

/* The root device can now be accessed; set process directories. */
for (rfp=&fproc[0]; rfp < &fproc[NR_PROCS]; rfp++) {
    FD_ZERO(&(rfp->fp_filp_inuse));
    if (rfp->fp_pid != PID_FREE) {
        rip = get_inode(root_dev, ROOT_INODE);
        dup_inode(rip);
        rfp->fp_rootdir = rip;
        rfp->fp_workdir = rip;
    } else rfp->fp_endpoint = NONE;
}
}

/*=====
*                               init_root                               *
*=====*/
PRIVATE void init_root()
{
    int bad;
    register struct super_block *sp;
    register struct inode *rip = NIL_INODE;
    int s;

    /* Open the root device. */
    root_dev = DEV_IMGRD;
    if ((s=dev_open(root_dev, FS_PROC_NR, R_BIT|W_BIT)) != OK)
        panic(__FILE__, "Cannot open root device", s);

#ifdef ENABLE_CACHE2
    /* The RAM disk is a second level block cache while not otherwise used. */
    init_cache2(ram_size);
#endif
}

```

```

/* Initialize the super_block table. */
for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
    sp->s_dev = NO_DEV;

/* Read in super_block for the root file system. */
sp = &super_block[0];
sp->s_dev = root_dev;

/* Check super_block for consistency. */
bad = (read_super(sp) != OK);
if (!bad) {
    rip = get_inode(root_dev, ROOT_INODE); /* inode for root dir */
    if ( (rip->i_mode & I_TYPE) != I_DIRECTORY || rip->i_nlinks < 3) bad++;
}
if (bad) panic(__FILE__, "Invalid root file system", NO_NUM);

sp->s_imount = rip;
dup_inode(rip);
sp->s_isup = rip;
sp->s_rd_only = 0;
return;
}

/*=====
*                                     service_pm                                     *
*=====*/
PRIVATE void service_pm()
{
    int r, call;
    message m;

    /* Ask PM for work until there is nothing left to do */
    for (;;)
    {
        m.m_type= PM_GET_WORK;
        r= sendrec(PM_PROC_NR, &m);
        if (r != OK)
        {
            panic("fs", "service_pm: sendrec failed", r);
        }
        if (m.m_type == PM_IDLE)
            break;
        call= m.m_type;
        switch(call)
        {
            case PM_STIME:
                boottime= m.PM_STIME_TIME;

                /* No need to report status to PM */
                break;
            case PM_SETSID:
                pm_setsid(m.PM_SETSID_PROC);

                /* No need to report status to PM */
                break;
            case PM_SETGID:
                pm_setgid(m.PM_SETGID_PROC, m.PM_SETGID_EGID,
                           m.PM_SETGID_RGID);

                /* No need to report status to PM */
                break;
            case PM_SETUID:
                pm_setuid(m.PM_SETUID_PROC, m.PM_SETUID_EGID,
                           m.PM_SETUID_RGID);

                /* No need to report status to PM */
                break;
            case PM_FORK:
                pm_fork(m.PM_FORK_PPROC, m.PM_FORK_CPROC,
                        m.PM_FORK_CPID);

```

```

        /* No need to report status to PM */
        break;

case PM_EXIT:
case PM_EXIT_TR:
    pm_exit(m.PM_EXIT_PROC);

    /* Reply dummy status to PM for synchronization */
    m.m_type= (call == PM_EXIT_TR ? PM_EXIT_REPLY_TR :
               PM_EXIT_REPLY);
    /* Keep m.PM_EXIT_PROC */

    r= send(PM_PROC_NR, &m);
    if (r != OK)
        panic(__FILE__, "service_pm: send failed", r);
    break;

case PM_UNPAUSE:
case PM_UNPAUSE_TR:
    unpause(m.PM_UNPAUSE_PROC);

    /* No need to report status to PM */
    break;

case PM_REBOOT:
    pm_reboot();

    /* Reply dummy status to PM for synchronization */
    m.m_type= PM_REBOOT_REPLY;
    r= send(PM_PROC_NR, &m);
    if (r != OK)
        panic(__FILE__, "service_pm: send failed", r);
    break;

case PM_EXEC:
    r= pm_exec(m.PM_EXEC_PROC, m.PM_EXEC_PATH,
               m.PM_EXEC_PATH_LEN, m.PM_EXEC_FRAME,
               m.PM_EXEC_FRAME_LEN);

    /* Reply status to PM */
    m.m_type= PM_EXEC_REPLY;
    /* Keep m.PM_EXEC_PROC */
    m.PM_EXEC_STATUS= r;

    r= send(PM_PROC_NR, &m);
    if (r != OK)
        panic(__FILE__, "service_pm: send failed", r);
    break;

case PM_DUMPCORE:
    r= pm_dumpcore(m.PM_CORE_PROC,
                   (struct mem_map *)m.PM_CORE_SEGPTR);

    /* Reply status to PM */
    m.m_type= PM_CORE_REPLY;
    /* Keep m.PM_CORE_PROC */
    m.PM_CORE_STATUS= r;

    r= send(PM_PROC_NR, &m);
    if (r != OK)
        panic(__FILE__, "service_pm: send failed", r);
    break;

default:
    panic("fs", "service_pm: unknown call", m.m_type);
}
}

```



```

/* This file contains a collection of miscellaneous procedures.  Some of them
 * perform simple system calls.  Some others do a little part of system calls
 * that are mostly performed by the Memory Manager.
 *
 * The entry points into this file are
 * do_dup:      perform the DUP system call
 * do_fcntl:    perform the FCNTL system call
 * do_sync:     perform the SYNC system call
 * do_fsync:    perform the FSYNC system call
 * pm_reboot:   sync disks and prepare for shutdown
 * pm_fork:     adjust the tables after MM has performed a FORK system call
 * do_exec:     handle files with FD_CLOEXEC on after MM has done an EXEC
 * do_exit:     a process has exited; note that in the tables
 * pm_setgid:   set group ids for some process
 * pm_setuid:   set user ids for some process
 * do_svrctl:   file system control
 * do_getsysinfo: request copy of FS data structure
 * pm_dumpcore: create a core dump
 */

#include "fs.h"
#include <fcntl.h>
#include <unistd.h>      /* cc runs out of memory with unistd.h :-( */
#include <minix/callnr.h>
#include <minix/endpoint.h>
#include <minix/com.h>
#include <sys/ptrace.h>
#include <sys/svrctl.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

#define CORE_NAME      "core"
#define CORE_MODE      0777      /* mode to use on core image files */

FORWARD _PROTOTYPE( void free_proc, (struct fproc *freed, int flags));
FORWARD _PROTOTYPE( int dumpcore, (int proc_e, struct mem_map *seg_ptr));
FORWARD _PROTOTYPE( int write_bytes, (struct inode *rip, off_t off,
    char *buf, size_t bytes));
FORWARD _PROTOTYPE( int write_seg, (struct inode *rip, off_t off, int proc_e,
    int seg, off_t seg_off, phys_bytes seg_bytes));

#define FP_EXITING      1

/*=====
 *
 * do_getsysinfo
 *
 *=====*/
PUBLIC int do_getsysinfo()
{
    struct fproc *proc_addr;
    vir_bytes src_addr, dst_addr;
    size_t len;
    int s;

    if (!super_user)
    {
        printf("FS: unauthorized call of do_getsysinfo by proc %d\n", who_e);
        return(EPERM); /* only su may call do_getsysinfo. This call may leak
            * information (and is not stable enough to be part
            * of the API/ABI).
            */
    }

    switch(m_in.info_what) {
    case SI_PROC_ADDR:
        proc_addr = &fproc[0];
        src_addr = (vir_bytes) &proc_addr;
        len = sizeof(struct fproc *);
        break;
    case SI_PROC_TAB:
        src_addr = (vir_bytes) fproc;

```

```

        len = sizeof(struct fproc) * NR_PROCS;
        break;
    case SI_DMAP_TAB:
        src_addr = (vir_bytes) dmap;
        len = sizeof(struct dmap) * NR_DEVICES;
        break;
    default:
        return(EINVAL);
}

dst_addr = (vir_bytes) m_in.info_where;
if (OK != (s=sys_datacopy(SELFS, src_addr, who_e, dst_addr, len)))
    return(s);
return(OK);
}

/*=====
 *
 *                               do_dup
 *=====*/
PUBLIC int do_dup()
{
    /* Perform the dup(fd) or dup2(fd,fd2) system call. These system calls are
    * obsolete. In fact, it is not even possible to invoke them using the
    * current library because the library routines call fcntl(). They are
    * provided to permit old binary programs to continue to run.
    */

    register int rfd;
    register struct filp *f;
    struct filp *dummy;
    int r;

    /* Is the file descriptor valid? */
    rfd = m_in.fd & ~DUP_MASK; /* kill off dup2 bit, if on */
    if ((f = get_filp(rfd)) == NIL_FILP) return(err_code);

    /* Distinguish between dup and dup2. */
    if (m_in.fd == rfd) { /* bit not on */
        /* dup(fd) */
        if ((r = get_fd(0, 0, &m_in.fd2, &dummy)) != OK) return(r);
    } else {
        /* dup2(fd, fd2) */
        if (m_in.fd2 < 0 || m_in.fd2 >= OPEN_MAX) return(EBADF);
        if (rfd == m_in.fd2) return(m_in.fd2); /* ignore the call: dup2(x, x) */
        m_in.fd = m_in.fd2; /* prepare to close fd2 */
        (void) do_close(); /* cannot fail */
    }

    /* Success. Set up new file descriptors. */
    f->filp_count++;
    fp->fp_filp[m_in.fd2] = f;
    FD_SET(m_in.fd2, &fp->fp_filp_inuse);
    return(m_in.fd2);
}

/*=====
 *
 *                               do_fcntl
 *=====*/
PUBLIC int do_fcntl()
{
    /* Perform the fcntl(fd, request, ...) system call. */

    register struct filp *f;
    int new_fd, r, fl;
    long cloexec_mask; /* bit map for the FD_CLOEXEC flag */
    long clo_value; /* FD_CLOEXEC flag in proper position */
    struct filp *dummy;

    /* Is the file descriptor valid? */
    if ((f = get_filp(m_in.fd)) == NIL_FILP) return(err_code);

    switch (m_in.request) {
        case F_DUPFD:

```

```

/* This replaces the old dup() system call. */
if (m_in.addr < 0 || m_in.addr >= OPEN_MAX) return(EINVAL);
if ((r = get_fd(m_in.addr, 0, &new_fd, &dummy)) != OK) return(r);
f->filp_count++;
fp->fp_filp[new_fd] = f;
return(new_fd);

case F_GETFD:
/* Get close-on-exec flag (FD_CLOEXEC in POSIX Table 6-2). */
return( ((fp->fp_cloexec >> m_in.fd) & 01) ? FD_CLOEXEC : 0);

case F_SETFD:
/* Set close-on-exec flag (FD_CLOEXEC in POSIX Table 6-2). */
cloexec_mask = 1L << m_in.fd; /* singleton set position ok */
clo_value = (m_in.addr & FD_CLOEXEC ? cloexec_mask : 0L);
fp->fp_cloexec = (fp->fp_cloexec & ~cloexec_mask) | clo_value;
return(OK);

case F_GETFL:
/* Get file status flags (O_NONBLOCK and O_APPEND). */
fl = f->filp_flags & (O_NONBLOCK | O_APPEND | O_ACCMODE);
return(fl);

case F_SETFL:
/* Set file status flags (O_NONBLOCK and O_APPEND). */
fl = O_NONBLOCK | O_APPEND;
f->filp_flags = (f->filp_flags & ~fl) | (m_in.addr & fl);
return(OK);

case F_GETLK:
case F_SETLK:
case F_SETLKW:
/* Set or clear a file lock. */
r = lock_op(f, m_in.request);
return(r);

case F_FREESP:
{
/* Free a section of a file. Preparation is done here,
 * actual freeing in freesp_inode().
 */
off_t start, end;
struct flock flock_arg;
signed long offset;

/* Check if it's a regular file. */
if((f->filp_ino->i_mode & I_TYPE) != I_REGULAR) {
    return EINVAL;
}

/* Copy flock data from userspace. */
if((r = sys_datacopy(who_e, (vir_bytes) m_in.name1,
    SELF, (vir_bytes) &flock_arg,
    (phys_bytes) sizeof(flock_arg))) != OK)
    return r;

/* Convert starting offset to signed. */
offset = (signed long) flock_arg.l_start;

/* Figure out starting position base. */
switch(flock_arg.l_whence) {
    case SEEK_SET: start = 0; if(offset < 0) return EINVAL; break;
    case SEEK_CUR: start = f->filp_pos; break;
    case SEEK_END: start = f->filp_ino->i_size; break;
    default: return EINVAL;
}

/* Check for overflow or underflow. */
if(offset > 0 && start + offset < start) { return EINVAL; }
if(offset < 0 && start + offset > start) { return EINVAL; }
start += offset;
if(flock_arg.l_len > 0) {
    end = start + flock_arg.l_len;
    if(end <= start) {

```

```

        return EINVAL;
    }
    r = freesp_inode(f->filp_ino, start, end);
} else {
    r = truncate_inode(f->filp_ino, start);
}
return r;
}

default:
    return(EINVAL);
}
}

/*=====
 *                               do_sync                               *
 *=====*/
PUBLIC int do_sync()
{
    /* Perform the sync() system call. Flush all the tables.
     * The order in which the various tables are flushed is critical. The
     * blocks must be flushed last, since rw_inode() leaves its results in
     * the block cache.
     */
    register struct inode *rip;
    register struct buf *bp;

    /* Write all the dirty inodes to the disk. */
    for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++)
        if (rip->i_count > 0 && rip->i_dirt == DIRTY) rw_inode(rip, WRITING);

    /* Write all the dirty blocks to the disk, one drive at a time. */
    for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++)
        if (bp->b_dev != NO_DEV && bp->b_dirt == DIRTY) flushall(bp->b_dev);

    return(OK);          /* sync() can't fail */
}

/*=====
 *                               do_fsync                               *
 *=====*/
PUBLIC int do_fsync()
{
    /* Perform the fsync() system call. For now, don't be unnecessarily smart. */

    do_sync();

    return(OK);
}

/*=====
 *                               pm_reboot                               *
 *=====*/
PUBLIC void pm_reboot()
{
    /* Perform the FS side of the reboot call. */
    int i;
    struct super_block *sp;
    struct inode dummy;

    /* Do exit processing for all leftover processes and servers,
     * but don't actually exit them (if they were really gone, PM
     * will tell us about it).
     */
    for (i = 0; i < NR_PROCS; i++)
        if ((m_in.endpt1 = fproc[i].fp_endpoint) != NONE)
            free_proc(&fproc[i], 0);

    /* The root file system is mounted onto itself, which keeps it from being
     * unmounted. Pull an inode out of thin air and put the root on it.
     */
    put_inode(super_block[0].s_imount);
    super_block[0].s_imount = &dummy;
    dummy.i_count = 2;
    /* expect one "put" */

```

```

/* Unmount all filesystems. File systems are mounted on other file systems,
 * so you have to pull off the loose bits repeatedly to get it all undone.
 */
for (i= 0; i < NR_SUPERS; i++) {
    /* Unmount at least one. */
    for (sp= &super_block[0]; sp < &super_block[NR_SUPERS]; sp++) {
        if (sp->s_dev != NO_DEV) (void) unmount(sp->s_dev);
    }
}

/* Sync any unwritten buffers. */
do_sync();
}

/*=====
 *                               pm_fork                               *
 *=====*/
PUBLIC void pm_fork(pproc, cproc, cpid)
int pproc;    /* Parent process */
int cproc;    /* Child process */
int cpid;     /* Child process id */
{
/* Perform those aspects of the fork() system call that relate to files.
 * In particular, let the child inherit its parent's file descriptors.
 * The parent and child parameters tell who forked off whom. The file
 * system uses the same slot numbers as the kernel.
 */

register struct fproc *cp;
int i, parentno, childno;

/* Check up-to-dateness of fproc. */
okendpt(pproc, &parentno);

/* PM gives child endpoint, which implies process slot information.
 * Don't call isokendpt, because that will verify if the endpoint
 * number is correct in fproc, which it won't be.
 */
childno = _ENDPOINT_P(cproc);
if(childno < 0 || childno >= NR_PROCS)
    panic(__FILE__, "FS: bogus child for forking", m_in.child_endpt);
if(fproc[childno].fp_pid != PID_FREE)
    panic(__FILE__, "FS: forking on top of in-use child", childno);

/* Copy the parent's fproc struct to the child. */
fproc[childno] = fproc[parentno];

/* Increase the counters in the 'filp' table. */
cp = &fproc[childno];
for (i = 0; i < OPEN_MAX; i++)
    if (cp->fp_filp[i] != NIL_FILP) cp->fp_filp[i]->filp_count++;

/* Fill in new process and endpoint id. */
cp->fp_pid = cpid;
cp->fp_endpoint = cproc;

/* A child is not a process leader. */
cp->fp_sesldr = 0;

/* This child has not exec()ced yet. */
cp->fp_execcd = 0;

/* Record the fact that both root and working dir have another user. */
dup_inode(cp->fp_rootdir);
dup_inode(cp->fp_workdir);
}

/*=====
 *                               free_proc                               *
 *=====*/
PRIVATE void free_proc(struct fproc *exiter, int flags)
{
    int i, task;

```

```

register struct fproc *rfp;
register struct filp *rfilp;
register struct inode *rip;
dev_t dev;

fp = exiter;          /* get_filp() needs 'fp' */

if (fp->fp_suspended == SUSPENDED) {
    task = -fp->fp_task;
    if (task == XPIPE || task == XOPEN) susp_count--;
    unpause(fp->fp_endpoint);
    fp->fp_suspended = NOT_SUSPENDED;
}

/* Loop on file descriptors, closing any that are open. */
for (i = 0; i < OPEN_MAX; i++) {
    (void) close_fd(fp, i);
}

/* Release root and working directories. */
put_inode(fp->fp_rootdir);
put_inode(fp->fp_workdir);
fp->fp_rootdir = NIL_INODE;
fp->fp_workdir = NIL_INODE;

/* Check if any process is SUSPENDED on this driver.
 * If a driver exits, unmap its entries in the dmap table.
 * (unmapping has to be done after the first step, because the
 * dmap table is used in the first step.)
 */
unsuspend_by_endpt(fp->fp_endpoint);

/* The rest of these actions is only done when processes actually
 * exit.
 */
if (!(flags & FP_EXITING))
    return;

/* Invalidate endpoint number for error and sanity checks. */
fp->fp_endpoint = NONE;

/* If a session leader exits and it has a controlling tty, then revoke
 * access to its controlling tty from all other processes using it.
 */
if (fp->fp_sesldr && fp->fp_tty != 0) {
    dev = fp->fp_tty;

    for (rfp = &fproc[0]; rfp < &fproc[NR_PROCS]; rfp++) {
        if (rfp->fp_pid == PID_FREE) continue;
        if (rfp->fp_tty == dev) rfp->fp_tty = 0;

        for (i = 0; i < OPEN_MAX; i++) {
            if ((rfilp = rfp->fp_filp[i]) == NIL_FILP) continue;
            if (rfilp->filp_mode == FILP_CLOSED) continue;
            rip = rfilp->filp_ino;
            if ((rip->i_mode & I_TYPE) != I_CHAR_SPECIAL) continue;
            if ((dev_t) rip->i_zone[0] != dev) continue;
            dev_close(dev);
            rfilp->filp_mode = FILP_CLOSED;
        }
    }
}

/* Exit done. Mark slot as free. */
fp->fp_pid = PID_FREE;
}

/*=====
 *                               pm_exit                               *
 *=====*/
PUBLIC void pm_exit(proc)
int proc;
{

```

```

    int exitee_p;
/* Perform the file system portion of the exit(status) system call. */

    /* Nevertheless, pretend that the call came from the user. */
    okendpt(proc, &exitee_p);
    free_proc(&fproc[exitee_p], FP_EXITING);
}

/*=====
 *                               pm_setgid                               *
 *=====*/
PUBLIC void pm_setgid(proc_e, egid, rgid)
int proc_e;
int egid;
int rgid;
{
    register struct fproc *tfp;
    int slot;

    okendpt(proc_e, &slot);
    tfp = &fproc[slot];

    tfp->fp_effgid = egid;
    tfp->fp_realgid = rgid;
}

/*=====
 *                               pm_setuid                               *
 *=====*/
PUBLIC void pm_setuid(proc_e, euid, ruid)
int proc_e;
int euid;
int ruid;
{
    register struct fproc *tfp;
    int slot;

    okendpt(proc_e, &slot);
    tfp = &fproc[slot];

    tfp->fp_effuid = euid;
    tfp->fp_realuid = ruid;
}

/*=====
 *                               do_svrctl                               *
 *=====*/
PUBLIC int do_svrctl()
{
    switch (m_in.svrctl_req) {
    case FSSIGNON: {
        /* A server in user space calls in to manage a device. */
        struct fssignon device;
        int r, major, proc_nr_n;

        if (fp->fp_effuid != SU_UID && fp->fp_effuid != SERVERS_UID)
            return(EPERM);

        /* Try to copy request structure to FS. */
        if ((r = sys_datacopy(who_e, (vir_bytes) m_in.svrctl_argp,
            FS_PROC_NR, (vir_bytes) &device,
            (phys_bytes) sizeof(device))) != OK)
            return(r);

        if (isokendpt(who_e, &proc_nr_n) != OK)
            return(EINVAL);

        /* Try to update device mapping. */
        major = (device.dev >> MAJOR) & BYTE;
        r=map_driver(major, who_e, device.style, 0 /* !force */);
        if (r == OK)
            {

```

```

        /* If a driver has completed its exec(), it can be announced
        * to be up.
        */
        if(fproc[proc_nr_n].fp_execced) {
            dev_up(major);
        } else {
            dmap[major].dmap_flags |= DMAP_BABY;
        }
    }

    return(r);
}

case FSDEVUNMAP: {
    struct fsdevunmap fdu;
    int r, major;
    /* Try to copy request structure to FS. */
    if ((r = sys_datacopy(who_e, (vir_bytes) m_in.svrctl_argp,
        FS_PROC_NR, (vir_bytes) &fdu,
        (phys_bytes) sizeof(fdu))) != OK)
        return(r);
    major = (fdu.dev >> MAJOR) & BYTE;
    r=map_driver(major, NONE, 0, 0);
    return(r);
}

default:
    return(EINVAL);
}
}

/*=====
 *                               pm_dumpcore                               *
 *=====*/
PUBLIC int pm_dumpcore(proc_e, seg_ptr)
int proc_e;
struct mem_map *seg_ptr;
{
    int r, proc_s;

    r= dumpcore(proc_e, seg_ptr);

    /* Terminate the process */
    okendpt(proc_e, &proc_s);
    free_proc(&fproc[proc_s], FP_EXITING);

    return r;
}

/*=====
 *                               dumpcore                               *
 *=====*/
PRIVATE int dumpcore(proc_e, seg_ptr)
int proc_e;
struct mem_map *seg_ptr;
{
    int r, seg, proc_s, exists;
    mode_t omode;
    vir_bytes len;
    off_t off, seg_off;
    long trace_off, trace_data;
    struct fproc *rfp;
    struct inode *rip, *ldirp;
    struct mem_map segs[NR_LOCAL_SEGS];

    okendpt(proc_e, &proc_s);
    rfp= fp= &fproc[proc_s];
    who_e= proc_e;
    who_p= proc_s;
    super_user = (fp->fp_effuid == SU_UID ? TRUE : FALSE);    /* su? */

    /* We need the equivalent of
    * open(CORE_NAME, O_WRONLY|O_CREAT|O_TRUNC|O_NONBLOCK, CORE_MODE)
    */

```



```

/* Create a new inode by calling new_node(). */
omode = I_REGULAR | (CORE_MODE & ALL_MODES & rfp->fp_umask);
rip = new_node(&ldirp, CORE_NAME, omode, NO_ZONE, 0, NULL);
r = err_code;
put_inode(ldirp);
exists= (r == EEXIST);
if (r != OK && r != EEXIST) return(r); /* error */

/* Only do the normal open code if we didn't just create the file. */
if (exists) {
    /* Check protections. */
    r = forbidden(rip, W_BIT);
    if (r != OK)
    {
        put_inode(rip);
        return r;
    }

    /* Make sure it is a regular file */
    switch (rip->i_mode & I_TYPE) {
        case I_REGULAR:
            break;

        case I_DIRECTORY:
            /* Directories may be read but not written. */
            r = EISDIR;
            break;

        case I_CHAR_SPECIAL:
        case I_BLOCK_SPECIAL:
        case I_NAMED_PIPE:
            r = EPERM;
            break;
    }

    if (r != OK)
    {
        put_inode(rip);
        return r;
    }

    /* Truncate the file */
    truncate_inode(rip, 0);
    wipe_inode(rip);
    /* Send the inode from the inode cache to the
     * block cache, so it gets written on the next
     * cache flush.
     */
    rw_inode(rip, WRITING);
}

/* Copy segments from PM */
r= sys_datacopy(PM_PROC_NR, (vir_bytes)seg_ptr,
    SELF, (vir_bytes)segs, sizeof(segs));
if (r != OK) panic(__FILE__, "dumpcore: cannot copy segment info", r);

off= 0;
r= write_bytes(rip, off, (char *)segs, sizeof(segs));
if (r != OK)
{
    put_inode(rip);
    return r;
}
off += sizeof(segs);

/* Write out the whole kernel process table entry to get the regs. */
for (trace_off= 0;; trace_off += sizeof(long))
{
    r= sys_trace(T_GETUSER, proc_e, trace_off, &trace_data);
    if (r != OK)
    {
        printf("dumpcore: sys_trace failed at offset %d: %d\n",
            trace_off, r);
        break;
    }
}

```

```

    }
    r= write_bytes(rip, off, (char *)&trace_data,
                   sizeof(trace_data));
    if (r != OK)
    {
        put_inode(rip);
        return r;
    }
    off += sizeof(trace_data);
}

/* Loop through segments and write the segments themselves out. */
for (seg = 0; seg < NR_LOCAL_SEGS; seg++) {
    len= segs[seg].mem_len << CLICK_SHIFT;
    seg_off= segs[seg].mem_vir << CLICK_SHIFT;
    r= write_seg(rip, off, proc_e, seg, seg_off, len);
    if (r != OK)
    {
        put_inode(rip);
        return r;
    }
    off += len;
}

rip->i_size= off;
rip->i_dirt = DIRTY;

put_inode(rip);
return OK;
}

/*=====
*                               write_bytes                               *
*=====*/
PRIVATE int write_bytes(rip, off, buf, bytes)
struct inode *rip;          /* inode descriptor to read from */
off_t off;                  /* offset in file */
char *buf;
size_t bytes;               /* how much is to be transferred? */
{
    int r, block_size;
    off_t n, o, b_off;
    block_t b;
    struct buf *bp;

    block_size= rip->i_sp->s_block_size;
    for (o= off - (off % block_size); o < off+bytes; o += block_size)
    {
        if (o < off)
            b_off= off-o;
        else
            b_off= 0;
        n= block_size-b_off;
        if (o+b_off+n > off+bytes)
            n= off+bytes-(o+b_off);

        b = read_map(rip, o);

        if (b == NO_BLOCK) {
            /* Writing to a nonexistent block. Create and enter
             * in inode.
             */
            if ((bp= new_block(rip, o)) == NIL_BUF)
                return(err_code);
        }
        else
        {
            /* Just read the block, no need to optimize for
             * writing entire blocks.
             */
            bp = get_block(rip->i_dev, b, NORMAL);
        }
    }
}

```

```

        if (n != block_size && o >= rip->i_size && b_off == 0) {
            zero_block(bp);
        }

        /* Copy a chunk from user space to the block buffer. */
        memcpy((bp->b_data+b_off), buf, n);
        bp->b_dirt = DIRTY;
        if (b_off + n == block_size)
            put_block(bp, FULL_DATA_BLOCK);
        else
            put_block(bp, PARTIAL_DATA_BLOCK);

        buf += n;
    }

    return OK;
}

/*=====
 *                               write_seg                               *
 *=====*/
PRIVATE int write_seg(rip, off, proc_e, seg, seg_off, seg_bytes)
struct inode *rip;          /* inode descriptor to read from */
off_t off;                  /* offset in file */
int proc_e;                 /* process number (endpoint) */
int seg;                    /* T, D, or S */
off_t seg_off;              /* Offset in segment */
phys_bytes seg_bytes;       /* how much is to be transferred? */
{
    int r, block_size, fl;
    off_t n, o, b_off;
    block_t b;
    struct buf *bp;

    block_size= rip->i_sp->s_block_size;
    for (o= off - (off % block_size); o < off+seg_bytes; o += block_size)
    {
        if (o < off)
            b_off= off-o;
        else
            b_off= 0;
        n= block_size-b_off;
        if (o+b_off+n > off+seg_bytes)
            n= off+seg_bytes-(o+b_off);

        b = read_map(rip, o);
        if (b == NO_BLOCK) {
            /* Writing to a nonexistent block. Create and enter in inode.*/
            if ((bp= new_block(rip, o)) == NIL_BUF)
                return(err_code);
        } else {
            /* Normally an existing block to be partially overwritten is
             * first read in. However, a full block need not be read in.
             * If it is already in the cache, acquire it, otherwise just
             * acquire a free buffer.
             */
            fl = (n == block_size ? NO_READ : NORMAL);
            bp = get_block(rip->i_dev, b, fl);
        }

        if (n != block_size && o >= rip->i_size && b_off == 0) {
            zero_block(bp);
        }

        /* Copy a chunk from user space to the block buffer. */
        r = sys_vircopy(proc_e, seg, (phys_bytes) seg_off,
                        FS_PROC_NR, D, (phys_bytes) (bp->b_data+b_off),
                        (phys_bytes) n);
        bp->b_dirt = DIRTY;
        fl = (b_off + n == block_size ? FULL_DATA_BLOCK : PARTIAL_DATA_BLOCK);
        put_block(bp, fl);

        seg_off += n;
    }
}

```

```
return OK;  
}
```

```

/* This file performs the MOUNT and UMount system calls.
 *
 * The entry points into this file are
 *   do_mount: perform the MOUNT system call
 *   do_umount: perform the UMount system call
 */

#include "fs.h"
#include <fcntl.h>
#include <string.h>
#include <minix/com.h>
#include <sys/stat.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

/* Allow the root to be replaced before the first 'real' mount. */
PRIVATE int allow_newroot= 1;

FORWARD _PROTOTYPE( dev_t name_to_dev, (char *path) ) ;

/*=====
 *                               do_mount                               *
 *=====*/
PUBLIC int do_mount()
{
/* Perform the mount(name, mfile, rd_only) system call. */

    register struct inode *rip, *root_ip;
    struct super_block *xp, *sp;
    dev_t dev;
    mode_t bits;
    int rdir, mdir;                /* TRUE iff {root/mount} file is dir */
    int i, r, found;
    struct fproc *tfp;

    /* Only the super-user may do MOUNT. */
    if (!super_user) return(EPERM);

    /* If 'name' is not for a block special file, return error. */
    if (fetch_name(m_in.namel, m_in.namel_length, M1) != OK) return(err_code);
    if ( (dev = name_to_dev(user_path)) == NO_DEV) return(err_code);

    /* Scan super block table to see if dev already mounted & find a free slot.*/
    sp = NIL_SUPER;
    found = FALSE;
    for (xp = &super_block[0]; xp < &super_block[NR_SUPERS]; xp++) {
        if (xp->s_dev == dev)
        {
            /* is it mounted already? */
            found = TRUE;
            sp= xp;
            break;
        }
        if (xp->s_dev == NO_DEV) sp = xp;        /* record free slot */
    }
    if (found)
    {
        printf(
"do_mount: s_imount= 0x%x (%x, %d), s_isup= 0x%x (%x, %d), fp_rootdir= 0x%x\n",
            xp->s_imount, xp->s_imount->i_dev, xp->s_imount->i_num,
            xp->s_isup, xp->s_isup->i_dev, xp->s_isup->i_num,
            fproc[FS_PROC_NR].fp_rootdir);
        /* It is possible that we have an old root lying around that
         * needs to be remounted.
         */
        if (xp->s_imount != xp->s_isup ||
            xp->s_isup == fproc[FS_PROC_NR].fp_rootdir)
        {
            /* Normally, s_imount refers to the mount point. For a root
             * filesystem, s_imount is equal to the root inode. We assume

```

```

        * that the root of FS is always the real root. If the two
        * inodes are different or if the root of FS is equal two the
        * root of the filesystem we found, we found a filesystem that
        * is in use.
        */
        return(EBUSY); /* already mounted */
    }

    if (root_dev == xp->s_dev)
    {
        panic("fs", "inconsistency remounting old root",
              NO_NUM);
    }

    /* Now get the inode of the file to be mounted on. */
    if (fetch_name(m_in.name2, m_in.name2_length, M1) != OK) {
        return(err_code);
    }

    if ( (rip = eat_path(user_path)) == NIL_INODE) {
        return(err_code);
    }

    r = OK;

    /* It may not be special. */
    bits = rip->i_mode & I_TYPE;
    if (bits == I_BLOCK_SPECIAL || bits == I_CHAR_SPECIAL)
        r = ENOTDIR;

    /* Get the root inode of the mounted file system. */
    root_ip= sp->s_isup;

    /* File types of 'rip' and 'root_ip' may not conflict. */
    if (r == OK) {
        mdir = ((rip->i_mode & I_TYPE) == I_DIRECTORY);
        /* TRUE iff dir */
        rdir = ((root_ip->i_mode & I_TYPE) == I_DIRECTORY);
        if (!mdir && rdir) r = EISDIR;
    }

    /* If error, return the mount point. */
    if (r != OK) {
        put_inode(rip);
        return(r);
    }

    /* Nothing else can go wrong. Perform the mount. */
    rip->i_mount = I_MOUNT; /* this bit says the inode is
                           * mounted on
                           */
    put_inode(sp->s_imount);
    sp->s_imount = rip;
    sp->s_rd_only = m_in.rd_only;
    allow_newroot= 0; /* The root is now fixed */
    return(OK);
}

if (sp == NIL_SUPER) return(ENFILE); /* no super block available */

/* Open the device the file system lives on. */
if (dev_open(dev, who_e, m_in.rd_only ? R_BIT : (R_BIT|W_BIT)) != OK)
    return(EINVAL);

/* Make the cache forget about blocks it has open on the filesystem */
(void) do_sync();
invalidate(dev);

/* Fill in the super block. */
sp->s_dev = dev; /* read_super() needs to know which dev */
r = read_super(sp);

/* Is it recognized as a Minix filesystem? */
if (r != OK) {
    dev_close(dev);

```

```

    sp->s_dev = NO_DEV;
    return(r);
}

/* Now get the inode of the file to be mounted on. */
if (fetch_name(m_in.name2, m_in.name2_length, M1) != OK) {
    dev_close(dev);
    sp->s_dev = NO_DEV;
    return(err_code);
}

if (strcmp(user_path, "/") == 0 && allow_newroot)
{
    printf("Replacing root\n");

    /* Get the root inode of the mounted file system. */
    if ( (root_ip = get_inode(dev, ROOT_INODE)) == NIL_INODE) r = err_code;
    if (root_ip != NIL_INODE && root_ip->i_mode == 0) {
        r = EINVAL;
    }

    /* If error, return the super block and both inodes; release the
     * maps.
     */
    if (r != OK) {
        put_inode(root_ip);
        (void) do_sync();
        invalidate(dev);
        dev_close(dev);
        sp->s_dev = NO_DEV;
        return(r);
    }

    /* Nothing else can go wrong. Perform the mount. */
    sp->s_imount = root_ip;
    dup_inode(root_ip);
    sp->s_isup = root_ip;
    sp->s_rd_only = m_in.rd_only;
    root_dev= dev;

    /* Replace all root and working directories */
    for (i= 0, tfp= fproc; i<NR_PROCS; i++, tfp++)
    {
        if (tfp->fp_pid == PID_FREE)
            continue;
        if (tfp->fp_rootdir == NULL)
            panic("fs", "do_mount: null rootdir", i);
        put_inode(tfp->fp_rootdir);
        dup_inode(root_ip);
        tfp->fp_rootdir= root_ip;

        if (tfp->fp_workdir== NULL)
            panic("fs", "do_mount: null workdir", i);
        put_inode(tfp->fp_workdir);
        dup_inode(root_ip);
        tfp->fp_workdir= root_ip;
    }

    /* Leave the old filesystem lying around. */
    return(OK);
}

if ( (rip = eat_path(user_path)) == NIL_INODE) {
    dev_close(dev);
    sp->s_dev = NO_DEV;
    return(err_code);
}

/* It may not be busy. */
r = OK;
if (rip->i_count > 1) r = EBUSY;

/* It may not be special. */
bits = rip->i_mode & I_TYPE;

```

```

if (bits == I_BLOCK_SPECIAL || bits == I_CHAR_SPECIAL) r = ENOTDIR;

/* Get the root inode of the mounted file system. */
root_ip = NIL_INODE;          /* if 'r' not OK, make sure this is defined */
if (r == OK) {
    if ( (root_ip = get_inode(dev, ROOT_INODE)) == NIL_INODE) r = err_code;
}
if (root_ip != NIL_INODE && root_ip->i_mode == 0) {
    r = EINVAL;
}

/* File types of 'rip' and 'root_ip' may not conflict. */
if (r == OK) {
    mdir = ((rip->i_mode & I_TYPE) == I_DIRECTORY); /* TRUE iff dir */
    rdir = ((root_ip->i_mode & I_TYPE) == I_DIRECTORY);
    if (!mdir && rdir) r = EISDIR;
}

/* If error, return the super block and both inodes; release the maps. */
if (r != OK) {
    put_inode(rip);
    put_inode(root_ip);
    (void) do_sync();
    invalidate(dev);
    dev_close(dev);
    sp->s_dev = NO_DEV;
    return(r);
}

/* Nothing else can go wrong. Perform the mount. */
rip->i_mount = I_MOUNT;      /* this bit says the inode is mounted on */
sp->s_imount = rip;
sp->s_isup = root_ip;
sp->s_rd_only = m_in.rd_only;
allow_newroot = 0;          /* The root is now fixed */
return(OK);
}

/*=====
*                               do_umount                               *
*=====*/
PUBLIC int do_umount()
{
    /* Perform the umount(name) system call. */
    dev_t dev;

    /* Only the super-user may do UMOUNT. */
    if (!super_user) return(EPERM);

    /* If 'name' is not for a block special file, return error. */
    if (fetch_name(m_in.name, m_in.name_length, M3) != OK) return(err_code);
    if ( (dev = name_to_dev(user_path)) == NO_DEV) return(err_code);

    return(unmount(dev));
}

/*=====
*                               unmount                               *
*=====*/
PUBLIC int unmount(dev)
Dev_t dev;
{
    /* Unmount a file system by device number. */
    register struct inode *rip;
    struct super_block *sp, *sp1;
    int count;

    /* See if the mounted device is busy. Only 1 inode using it should be
     * open -- the root inode -- and that inode only 1 time.
     */
    count = 0;
    for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++)
        if (rip->i_count > 0 && rip->i_dev == dev) count += rip->i_count;
    if (count > 1) return(EBUSY); /* can't umount a busy file system */

```



```

/* Find the super block. */
sp = NIL_SUPER;
for (spl = &super_block[0]; spl < &super_block[NR_SUPERS]; spl++) {
    if (spl->s_dev == dev) {
        sp = spl;
        break;
    }
}

/* Sync the disk, and invalidate cache. */
(void) do_sync(); /* force any cached blocks out of memory */
invalidate(dev); /* invalidate cache entries for this dev */
if (sp == NIL_SUPER) {
    return(EINVAL);
}

/* Close the device the file system lives on. */
dev_close(dev);

/* Finish off the unmount. */
sp->s_imount->i_mount = NO_MOUNT; /* inode returns to normal */
put_inode(sp->s_imount); /* release the inode mounted on */
put_inode(sp->s_isup); /* release the root inode of the mounted fs */
sp->s_imount = NIL_INODE;
sp->s_dev = NO_DEV;
return(OK);
}

/*=====
*                               name_to_dev                               *
*=====*/
PRIVATE dev_t name_to_dev(path)
char *path; /* pointer to path name */
{
/* Convert the block special file 'path' to a device number. If 'path'
* is not a block special file, return error code in 'err_code'.
*/

register struct inode *rip;
register dev_t dev;

/* If 'path' can't be opened, give up immediately. */
if ( (rip = eat_path(path)) == NIL_INODE) return(NO_DEV);

/* If 'path' is not a block special file, return error. */
if ( (rip->i_mode & I_TYPE) != I_BLOCK_SPECIAL) {
    err_code = ENOTBLK;
    put_inode(rip);
    return(NO_DEV);
}

/* Extract the device number. */
dev = (dev_t) rip->i_zone[0];
put_inode(rip);
return(dev);
}

```

```

/* This file contains the procedures for creating, opening, closing, and
 * seeking on files.
 *
 * The entry points into this file are
 * do_creat: perform the CREAT system call
 * do_open: perform the OPEN system call
 * do_mknod: perform the MKNOD system call
 * do_mkdir: perform the MKDIR system call
 * do_close: perform the CLOSE system call
 * do_lseek: perform the LSEEK system call
 * new_node: create a new file, directory, etc.
 */

#include "fs.h"
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "lock.h"
#include "param.h"
#include "super.h"

#define offset m2_l1

PRIVATE char mode_map[] = {R_BIT, W_BIT, R_BIT|W_BIT, 0};

FORWARD _PROTOTYPE( int common_open, (int oflags, mode_t omode) );
FORWARD _PROTOTYPE( int pipe_open, (struct inode *rip, mode_t bits, int oflags));

/*=====
 * do_creat
 *=====*/
PUBLIC int do_creat()
{
/* Perform the creat(name, mode) system call. */
int r;

if (fetch_name(m_in.name, m_in.name_length, M3) != OK) return(err_code);
r = common_open(O_WRONLY | O_CREAT | O_TRUNC, (mode_t) m_in.mode);
return(r);
}

/*=====
 * do_open
 *=====*/
PUBLIC int do_open()
{
/* Perform the open(name, flags,...) system call. */

int create_mode = 0; /* is really mode_t but this gives problems */
int r;

/* If O_CREAT is set, open has three parameters, otherwise two. */
if (m_in.mode & O_CREAT) {
create_mode = m_in.c_mode;
r = fetch_name(m_in.c_name, m_in.name1_length, M1);
} else {
r = fetch_name(m_in.name, m_in.name_length, M3);
}

if (r != OK) return(err_code); /* name was bad */
r = common_open(m_in.mode, create_mode);
return(r);
}

/*=====
 * common_open
 *=====*/

```

```

PRIVATE int common_open(register int oflags, mode_t omode)
{
    /* Common code from do_creat and do_open. */

    struct inode *rip, *ldirp;
    int r, b, exist = TRUE;
    dev_t dev;
    mode_t bits;
    off_t pos;
    struct filp *filp_ptr, *filp2;

    /* Remap the bottom two bits of oflags. */
    bits = (mode_t) mode_map[oflags & O_ACCMODE];

    /* See if file descriptor and filp slots are available. */
    if ( (r = get_fd(0, bits, &m_in.fd, &filp_ptr)) != OK) return(r);

    /* If O_CREATE is set, try to make the file. */
    if (oflags & O_CREAT) {
        /* Create a new inode by calling new_node(). */
        omode = I_REGULAR | (omode & ALL_MODES & fp->fp_umask);
        rip = new_node(&ldirp, user_path, omode, NO_ZONE, oflags&O_EXCL, NULL);
        r = err_code;
        put_inode(ldirp);
        if (r == OK) exist = FALSE;          /* we just created the file */
        else if (r != EEXIST) return(r);    /* other error */
        else exist = !(oflags & O_EXCL);    /* file exists, if the O_EXCL
                                           flag is set this is an error */
    } else {
        /* Scan path name. */
        if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
    }

    /* Claim the file descriptor and filp slot and fill them in. */
    fp->fp_filp[m_in.fd] = filp_ptr;
    FD_SET(m_in.fd, &fp->fp_filp_inuse);
    filp_ptr->filp_count = 1;
    filp_ptr->filp_ino = rip;
    filp_ptr->filp_flags = oflags;

    /* Only do the normal open code if we didn't just create the file. */
    if (exist) {
        /* Check protections. */
        if ((r = forbidden(rip, bits)) == OK) {
            /* Opening reg. files directories and special files differ. */
            switch (rip->i_mode & I_TYPE) {
                case I_REGULAR:
                    /* Truncate regular file if O_TRUNC. */
                    if (oflags & O_TRUNC) {
                        if ((r = forbidden(rip, W_BIT)) != OK) break;
                        truncate_inode(rip, 0);
                        wipe_inode(rip);
                        /* Send the inode from the inode cache to the
                         * block cache, so it gets written on the next
                         * cache flush.
                         */
                        rw_inode(rip, WRITING);
                    }
                    break;

                case I_DIRECTORY:
                    /* Directories may be read but not written. */
                    r = (bits & W_BIT ? EISDIR : OK);
                    break;

                case I_CHAR_SPECIAL:
                case I_BLOCK_SPECIAL:
                    /* Invoke the driver for special processing. */
                    dev = (dev_t) rip->i_zone[0];
                    r = dev_open(dev, who_e, bits | (oflags & ~O_ACCMODE));
                    break;

                case I_NAMED_PIPE:
                    oflags |= O_APPEND;          /* force append mode */
            }
        }
    }
}

```

```

        fil_ptr->filp_flags = oflags;
        r = pipe_open(rip, bits, oflags);
        if (r != ENXIO) {
            /* See if someone else is doing a rd or wt on
             * the FIFO. If so, use its filp entry so the
             * file position will be automatically shared.
             */
            b = (bits & R_BIT ? R_BIT : W_BIT);
            fil_ptr->filp_count = 0; /* don't find self */
            if ((filp2 = find_filp(rip, b)) != NIL_FILP) {
                /* Co-reader or writer found. Use it.*/
                fp->fp_filp[m_in.fd] = filp2;
                filp2->filp_count++;
                filp2->filp_ino = rip;
                filp2->filp_flags = oflags;

                /* i_count was incremented incorrectly
                 * by eatpath above, not knowing that
                 * we were going to use an existing
                 * filp entry. Correct this error.
                 */
                rip->i_count--;
            } else {
                /* Nobody else found. Restore filp. */
                fil_ptr->filp_count = 1;
                if (b == R_BIT)
                    pos = rip->i_zone[V2_NR_DZONES+0];
                else
                    pos = rip->i_zone[V2_NR_DZONES+1];
                fil_ptr->filp_pos = pos;
            }
        }
        break;
    }
}

/* If error, release inode. */
if (r != OK) {
    if (r == SUSPEND) return(r); /* Oops, just suspended */
    fp->fp_filp[m_in.fd] = NIL_FILP;
    FD_CLR(m_in.fd, &fp->fp_filp_inuse);
    fil_ptr->filp_count = 0;
    put_inode(rip);
    return(r);
}

return(m_in.fd);
}

/*=====
 *                               new_node                               *
 *=====*/
PUBLIC struct inode *new_node(struct inode **ldirp,
    char *path, mode_t bits, zone_t z0, int opaque, char *parsed)
{
    /* New_node() is called by common_open(), do_mknod(), and do_mkdir().
     * In all cases it allocates a new inode, makes a directory entry for it on
     * the path 'path', and initializes it. It returns a pointer to the inode if
     * it can do this; otherwise it returns NIL_INODE. It always sets 'err_code'
     * to an appropriate value (OK or an error code).
     *
     * The parsed path rest is returned in 'parsed' if parsed is nonzero. It
     * has to hold at least NAME_MAX bytes.
     */

    register struct inode *rip;
    register int r;
    char string[NAME_MAX];

    *ldirp = parse_path(path, string, opaque ? LAST_DIR : LAST_DIR_EATSYM);
    if (*ldirp == NIL_INODE) return(NIL_INODE);

    /* The final directory is accessible. Get final component of the path. */

```

```

rip = advance(ldirp, string);

if (S_ISDIR(bits) &&
    (*ldirp)->i_nlinks >= ((*ldirp)->i_sp->s_version == V1 ?
    CHAR_MAX : SHRT_MAX)) {
    /* New entry is a directory, alas we can't give it a "." */
    put_inode(rip);
    err_code = EMLINK;
    return(NIL_INODE);
}

if (rip == NIL_INODE && err_code == ENOENT) {
    /* Last path component does not exist. Make new directory entry. */
    if ( (rip = alloc_inode((*ldirp)->i_dev, bits)) == NIL_INODE) {
        /* Can't creat new inode: out of inodes. */
        return(NIL_INODE);
    }

    /* Force inode to the disk before making directory entry to make
     * the system more robust in the face of a crash: an inode with
     * no directory entry is much better than the opposite.
     */
    rip->i_nlinks++;
    rip->i_zone[0] = z0;          /* major/minor device numbers */
    rw_inode(rip, WRITING);      /* force inode to disk now */

    /* New inode acquired. Try to make directory entry. */
    if ((r = search_dir(*ldirp, string, &rip->i_num, ENTER)) != OK) {
        rip->i_nlinks--;          /* pity, have to free disk inode */
        rip->i_dirt = DIRTY;      /* dirty inodes are written out */
        put_inode(rip); /* this call frees the inode */
        err_code = r;
        return(NIL_INODE);
    }
} else {
    /* Either last component exists, or there is some problem. */
    if (rip != NIL_INODE)
        r = EEXIST;
    else
        r = err_code;
}

if(parsed) { /* Give the caller the parsed string if requested. */
    strncpy(parsed, string, NAME_MAX-1);
    parsed[NAME_MAX-1] = '\0';
}

/* The caller has to return the directory inode (*ldirp). */
err_code = r;
return(rip);
}

/*=====
 *                               pipe_open                               *
 *=====*/
PRIVATE int pipe_open(register struct inode *rip, register mode_t bits,
    register int oflags)
{
    /* This function is called from common_open. It checks if
     * there is at least one reader/writer pair for the pipe, if not
     * it suspends the caller, otherwise it revives all other blocked
     * processes hanging on the pipe.
     */

    rip->i_pipe = I_PIPE;

    if((bits & (R_BIT|W_BIT)) == (R_BIT|W_BIT)) {
        printf("pipe opened RW.\n");
        return ENXIO;
    }

    if (find_filp(rip, bits & W_BIT ? R_BIT : W_BIT) == NIL_FILP) {
        if (oflags & O_NONBLOCK) {

```

```

        if (bits & W_BIT) return(ENXIO);
    } else {
        suspend(XPOPEN);          /* suspend caller */
        return(SUSPEND);
    }
} else if (susp_count > 0) { /* revive blocked processes */
    release(rip, OPEN, susp_count);
    release(rip, CREAT, susp_count);
}
return(OK);
}

/*=====
 *                               do_mknod                               *
 *=====*/
PUBLIC int do_mknod()
{
    /* Perform the mknod(name, mode, addr) system call. */

    register mode_t bits, mode_bits;
    struct inode *ip, *ldirp;

    /* Only the super_user may make nodes other than fifos. */
    mode_bits = (mode_t) m_in.mk_mode;          /* mode of the inode */
    if (!super_user && ((mode_bits & I_TYPE) != I_NAMED_PIPE)) return(EPERM);
    if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
    bits = (mode_bits & I_TYPE) | (mode_bits & ALL_MODES & fp->fp_umask);
    ip = new_node(&ldirp, user_path, bits, (zone_t) m_in.mk_z0, TRUE, NULL);
    put_inode(ip);
    put_inode(ldirp);
    return(err_code);
}

/*=====
 *                               do_mkdir                               *
 *=====*/
PUBLIC int do_mkdir()
{
    /* Perform the mkdir(name, mode) system call. */

    int r1, r2;                          /* status codes */
    ino_t dot, dotdot;                   /* inode numbers for . and .. */
    mode_t bits;                          /* mode bits for the new inode */
    char string[NAME_MAX];                /* last component of the new dir's path name */
    struct inode *rip, *ldirp;

    if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);

    /* Next make the inode. If that fails, return error code. */
    bits = I_DIRECTORY | (m_in.mode & RWX_MODES & fp->fp_umask);
    rip = new_node(&ldirp, user_path, bits, (zone_t) 0, TRUE, string);
    if (rip == NIL_INODE || err_code == EEXIST) {
        put_inode(rip);                  /* can't make dir: it already exists */
        put_inode(ldirp);
        return(err_code);
    }

    /* Get the inode numbers for . and .. to enter in the directory. */
    dotdot = ldirp->i_num;                /* parent's inode number */
    dot = rip->i_num;                      /* inode number of the new dir itself */

    /* Now make dir entries for . and .. unless the disk is completely full. */
    /* Use dot1 and dot2, so the mode of the directory isn't important. */
    rip->i_mode = bits;                    /* set mode */
    r1 = search_dir(rip, dot1, &dot, ENTER); /* enter . in the new dir */
    r2 = search_dir(rip, dot2, &dotdot, ENTER); /* enter .. in the new dir */

    /* If both . and .. were successfully entered, increment the link counts. */
    if (r1 == OK && r2 == OK) {
        /* Normal case. It was possible to enter . and .. in the new dir. */
        rip->i_nlinks++;                  /* this accounts for . */
        ldirp->i_nlinks++;                /* this accounts for .. */
        ldirp->i_dirt = DIRTY;            /* mark parent's inode as dirty */
    } else {

```

```

    /* It was not possible to enter . or .. probably disk was full -
    * links counts haven't been touched.
    */
    if(search_dir(ldirp, string, (ino_t *) 0, DELETE) != OK)
        panic(__FILE__, "Dir disappeared ", rip->i_num);
    rip->i_nlinks--; /* undo the increment done in new_node() */
}
rip->i_dirt = DIRTY; /* either way, i_nlinks has changed */

put_inode(ldirp); /* return the inode of the parent dir */
put_inode(rip); /* return the inode of the newly made dir */
return(err_code); /* new_node() always sets 'err_code' */
}

/*=====
* do_close
*=====*/
PUBLIC int do_close()
{
    /* Perform the close(fd) system call. */
    return close_fd(fp, m_in.fd);
}

/*=====
* close_fd
*=====*/
PUBLIC int close_fd(rfp, fd_nr)
struct fproc *rfp;
int fd_nr;
{
    /* Close a filedescriptor for a process. */

    register struct filp *rfilp;
    register struct inode *rip;
    struct file_lock *flp;
    int rw, mode_word, lock_count;
    dev_t dev;

    /* First locate the inode that belongs to the file descriptor. */
    if ( (rfilp = get_filp2(rfp, fd_nr)) == NIL_FILP) return(err_code);
    rip = rfilp->filp_ino; /* 'rip' points to the inode */

    if (rfilp->filp_count - 1 == 0 && rfilp->filp_mode != FILP_CLOSED) {
        /* Check to see if the file is special. */
        mode_word = rip->i_mode & I_TYPE;
        if (mode_word == I_CHAR_SPECIAL || mode_word == I_BLOCK_SPECIAL) {
            dev = (dev_t) rip->i_zone[0];
            if (mode_word == I_BLOCK_SPECIAL) {
                /* Invalidate cache entries unless special is mounted
                * or ROOT
                */
                if (!mounted(rip)) {
                    (void) do_sync(); /* purge cache */
                    invalidate(dev);
                }
            }
            /* Do any special processing on device close. */
            dev_close(dev);
        }
    }

    /* If the inode being closed is a pipe, release everyone hanging on it. */
    if (rip->i_pipe == I_PIPE) {
        rw = (rfilp->filp_mode & R_BIT ? WRITE : READ);
        release(rip, rw, NR_PROCS);
    }

    /* If a write has been done, the inode is already marked as DIRTY. */
    if (--rfilp->filp_count == 0) {
        if (rip->i_pipe == I_PIPE && rip->i_count > 1) {
            /* Save the file position in the i-node in case needed later.
            * The read and write positions are saved separately. The
            * last 3 zones in the i-node are not used for (named) pipes.
            */

```

```

        if (rfilp->filp_mode == R_BIT)
            rip->i_zone[V2_NR_DZONES+0] = (zone_t) rfilp->filp_pos;
        else
            rip->i_zone[V2_NR_DZONES+1] = (zone_t) rfilp->filp_pos;
    }
    put_inode(rip);
}

rfp->fp_cloexec &= ~(1L << fd_nr);    /* turn off close-on-exec bit */
rfp->fp_filp[fd_nr] = NIL_FILP;
FD_CLR(fd_nr, &rfp->fp_filp_inuse);

/* Check to see if the file is locked. If so, release all locks. */
if (nr_locks == 0) return(OK);
lock_count = nr_locks;    /* save count of locks */
for (flp = &file_lock[0]; flp < &file_lock[NR_LOCKS]; flp++) {
    if (flp->lock_type == 0) continue;    /* slot not in use */
    if (flp->lock_inode == rip && flp->lock_pid == rfp->fp_pid) {
        flp->lock_type = 0;
        nr_locks--;
    }
}
if (nr_locks < lock_count) lock_revive();    /* lock released */
return(OK);
}

/*=====
*                               do_lseek                               *
*=====*/
PUBLIC int do_lseek()
{
    /* Perform the lseek(ls_fd, offset, whence) system call. */

    register struct filp *rfilp;
    register off_t pos;

    /* Check to see if the file descriptor is valid. */
    if ( ( rfilp = get_filp(m_in.ls_fd)) == NIL_FILP) return(err_code);

    /* No lseek on pipes. */
    if (rfilp->filp_ino->i_pipe == I_PIPE) return(ESPIPE);

    /* The value of 'whence' determines the start position to use. */
    switch(m_in.whence) {
        case SEEK_SET: pos = 0; break;
        case SEEK_CUR: pos = rfilp->filp_pos; break;
        case SEEK_END: pos = rfilp->filp_ino->i_size; break;
        default: return(EINVAL);
    }

    /* Check for overflow. */
    if (((long)m_in.offset > 0) && ((long)(pos + m_in.offset) < (long)pos))
        return(EINVAL);
    if (((long)m_in.offset < 0) && ((long)(pos + m_in.offset) > (long)pos))
        return(EINVAL);
    pos = pos + m_in.offset;

    if (pos != rfilp->filp_pos)
        rfilp->filp_ino->i_seek = ISEEK;    /* inhibit read ahead */
    rfilp->filp_pos = pos;
    m_out.reply_ll = pos;    /* insert the long into the output message */
    return(OK);
}

/*=====
*                               do_slink                               *
*=====*/
PUBLIC int do_slink()
{
    /* Perform the symlink(name1, name2) system call. */

    register int r;    /* error code */
    char string[NAME_MAX];    /* last component of the new dir's path name */
    struct inode *sip;    /* inode containing symbolic link */

```



```

struct buf *bp;                /* disk buffer for link */
struct inode *ldirp;           /* directory containing link */

if (fetch_name(m_in.name2, m_in.name2_length, M1) != OK)
    return(err_code);

if (m_in.name1_length <= 1 || m_in.name1_length >= _MIN_BLOCK_SIZE)
    return(ENAMETOOLONG);

/* Create the inode for the symlink. */
sip = new_node(&ldirp, user_path, (mode_t) (I_SYMBOLIC_LINK | RWX_MODES),
              (zone_t) 0, TRUE, string);

/* Allocate a disk block for the contents of the symlink.
 * Copy contents of symlink (the name pointed to) into first disk block.
 */
if ((r = err_code) == OK) {
    r = (bp = new_block(sip, (off_t) 0)) == NIL_BUF
        ? err_code
        : sys_vircopy(who_e, D, (vir_bytes) m_in.name1,
                      SELF, D, (vir_bytes) bp->b_data,
                      (vir_bytes) m_in.name1_length-1);

    if(r == OK) {
        bp->b_data[_MIN_BLOCK_SIZE-1] = '\0';
        sip->i_size = strlen(bp->b_data);
        if(sip->i_size != m_in.name1_length-1) {
            /* This can happen if the user provides a buffer
             * with a \0 in it. This can cause a lot of trouble
             * when the symlink is used later. We could just use
             * the strlen() value, but we want to let the user
             * know he did something wrong. ENAMETOOLONG doesn't
             * exactly describe the error, but there is no
             * ENAMETOOWRONG.
             */
            r = ENAMETOOLONG;
        }
    }

    put_block(bp, DIRECTORY_BLOCK); /* put_block() accepts NIL_BUF. */

    if (r != OK) {
        sip->i_nlinks = 0;
        if (search_dir(ldirp, string, (ino_t *) 0, DELETE) != OK)
            panic(__FILE__, "Symbolic link vanished", NO_NUM);
    }
}

/* put_inode() accepts NIL_INODE as a noop, so the below are safe. */
put_inode(sip);
put_inode(ldirp);

return(r);
}

```

```
/* The following names are synonyms for the variables in the input message. */
#define acc_time      m2_l1
#define addr          m1_i3
#define buffer        m1_p1
#define child_endpt   m1_i2
#define co_mode       m1_i1
#define eff_grp_id    m1_i3
#define eff_user_id   m1_i3
#define erki          m1_p1
#define fd            m1_i1
#define fd2           m1_i2
#define ioflags       m1_i3
#define group         m1_i3
#define real_grp_id   m1_i2
#define ls_fd         m2_i1
#define mk_mode       m1_i2
#define mk_z0         m1_i3
#define mode          m3_i2
#define c_mode        m1_i3
#define c_name        m1_p1
#define name          m3_p1
#define name1         m1_p1
#define name2         m1_p2
#define name_length   m3_i1
#define name1_length  m1_i1
#define name2_length  m1_i2
#define nbytes        m1_i2
#define owner         m1_i2
#define parent_endpt  m1_i1
#define pathname      m3_cal
#define pid           m1_i3
#define ENDPT         m1_i1
#define ctl_req       m4_l1
#define driver_nr     m4_l2
#define dev_nr        m4_l3
#define dev_style     m4_l4
#define m_force       m4_l5
#define rd_only       m1_i3
#define real_user_id  m1_i2
#define request       m1_i2
#define sig           m1_i2
#define endpt1        m1_i1
#define tp            m2_l1
#define utime_actime   m2_l1
#define utime_modtime m2_l2
#define utime_file     m2_p1
#define utime_length   m2_i1
#define utime_strlen  m2_i2
#define whence        m2_i2
#define svrctl_req     m2_i1
#define svrctl_argp    m2_p1
#define pm_stime       m1_i1
#define info_what      m1_i1
#define info_where     m1_p1

/* The following names are synonyms for the variables in the output message. */
#define reply_type     m_type
#define reply_l1       m2_l1
#define reply_i1       m1_i1
#define reply_i2       m1_i2
#define reply_t1       m4_l1
#define reply_t2       m4_l2
#define reply_t3       m4_l3
#define reply_t4       m4_l4
#define reply_t5       m4_l5
```

```

/* This file contains the procedures that look up path names in the directory
 * system and determine the inode number that goes with a given path name.
 *
 * The entry points into this file are
 * eat_path: the 'main' routine of the path-to-inode conversion mechanism
 * last_dir: find the final directory on a given path
 * advance: parse one component of a path name
 * search_dir: search a directory for a string and return its inode number
 */

#include "fs.h"
#include <string.h>
#include <minix/callnr.h>
#include <sys/stat.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "super.h"

PUBLIC char dot1[2] = "."; /* used for search_dir to bypass the access */
PUBLIC char dot2[3] = ".."; /* permissions for . and .. */

FORWARD _PROTOTYPE( char *get_name, (char *old_name, char string [NAME_MAX]) );

FORWARD _PROTOTYPE( struct inode *ltraverse, (struct inode *rip,
                                             char *path, char *suffix, struct inode *ldip) );

/*=====
 * parse_path
 *=====*/
PUBLIC struct inode *parse_path(path, string, action)
char *path; /* the path name to be parsed */
char string[NAME_MAX]; /* the final component is returned here */
int action; /* action on last part of path */
{
/* This is the actual code for last_dir and eat_path. Return the inode of
 * the last directory and the name of object within that directory, or the
 * inode of the last object (an empty name will be returned). Names are
 * returned in string. If string is null the name is discarded. The action
 * code determines how "last" is defined. If an error occurs, NIL_INODE
 * will be returned with an error code in err_code.
 */

    struct inode *rip, *dir_ip;
    char *new_name;
    int symloop;
    char lstring[NAME_MAX];

    /* Is the path absolute or relative? Initialize 'rip' accordingly. */
    rip = (*path == '/' ? fp->fp_rootdir : fp->fp_workdir);

    /* If dir has been removed or path is empty, return ENOENT. */
    if (rip->i_nlinks == 0 || *path == '\0') {
        err_code = ENOENT;
        return(NIL_INODE);
    }

    dup_inode(rip); /* inode will be returned with put_inode */

    symloop = 0; /* symbolic link traversal count */
    if (string == (char *) 0) string = lstring;

    /* Scan the path component by component. */
    while (TRUE) {
        /* Extract one component. */
        if ( (new_name = get_name(path, string)) == (char*) 0) {
            put_inode(rip); /* bad path in user space */
            return(NIL_INODE);
        }
        if (*new_name == '\0' && (action & PATH_PENULTIMATE)) {
            if ( (rip->i_mode & I_TYPE) == I_DIRECTORY) {
                return(rip); /* normal exit */
            }
        }
    }
}

```

```

        } else {
            /* last file of path prefix is not a directory */
            put_inode(rip);
            err_code = ENOTDIR;
            return(NIL_INODE);
        }
    }

    /* There is more path. Keep parsing. */
    dir_ip = rip;
    rip = advance(&dir_ip, string);

    if (rip == NIL_INODE) {
        if (*new_name == '\0' && (action & PATH_NONSYMBOLIC) != 0)
            return(dir_ip);
        else {
            put_inode(dir_ip);
            return(NIL_INODE);
        }
    }

    /* The call to advance() succeeded. Fetch next component. */
    if (S_ISLNK(rip->i_mode)) {
        if (*new_name != '\0' || (action & PATH_OPAQUE) == 0) {
            if (*new_name != '\0') new_name--;
            rip = ltraverse(rip, path, new_name, dir_ip);
            put_inode(dir_ip);
            if (++symloop > SYMLOOP) {
                err_code = ELOOP;
                put_inode(rip);
                rip = NIL_INODE;
            }
            if (rip == NIL_INODE) return(NIL_INODE);
            continue;
        }
    } else if (*new_name != '\0') {
        put_inode(dir_ip);
        path = new_name;
        continue;
    }

    /* Either last name reached or symbolic link is opaque */
    if ((action & PATH_NONSYMBOLIC) != 0) {
        put_inode(rip);
        return(dir_ip);
    } else {
        put_inode(dir_ip);
        return(rip);
    }
}

}

/*=====
 *
 * eat_path
 *=====*/
PUBLIC struct inode *eat_path(path)
char *path; /* the path name to be parsed */
{
    /* Parse the path 'path' and put its inode in the inode table. If not possible,
     * return NIL_INODE as function value and an error code in 'err_code'.
     */

    return parse_path(path, (char *) 0, EAT_PATH);
}

/*=====
 *
 * last_dir
 *=====*/
PUBLIC struct inode *last_dir(path, string)
char *path; /* the path name to be parsed */
char string[NAME_MAX]; /* the final component is returned here */
{
    /* Given a path, 'path', located in the fs address space, parse it as
     * far as the last directory, fetch the inode for the last directory into

```

```

* the inode table, and return a pointer to the inode. In
* addition, return the final component of the path in 'string'.
* If the last directory can't be opened, return NIL_INODE and
* the reason for failure in 'err_code'.
*/

return parse_path(path, string, LAST_DIR);
}

/*=====
*                               ltraverse                               *
*=====*/
PRIVATE struct inode *ltraverse(rip, path, suffix, ldip)
register struct inode *rip;    /* symbolic link */
char *path;                  /* path containing link */
char *suffix;                /* suffix following link within path */
register struct inode *ldip;   /* directory containing link */
{
/* Traverse a symbolic link. Copy the link text from the inode and insert
* the text into the path. Return the inode of base directory and the
* ammended path. The symbolic link inode is always freed. The inode
* returned is already duplicated. NIL_INODE is returned on error.
*/

    block_t b;                /* block containing link text */
    struct inode *bip;         /* inode of base directory */
    struct buf *bp;           /* buffer containing link text */
    size_t sl;                /* length of link */
    size_t tl;                /* length of suffix */
    char *sp;                 /* start of link text */

    bip = NIL_INODE;
    bp = NIL_BUF;

    if ((b = read_map(rip, (off_t) 0)) != NO_BLOCK) {
        bp = get_block(rip->i_dev, b, NORMAL);
        sl = rip->i_size;
        sp = bp->b_data;

        /* Insert symbolic text into path name. */
        tl = strlen(suffix);
        if (sl > 0 && sl + tl <= PATH_MAX-1) {
            memmove(path+sl, suffix, tl);
            memmove(path, sp, sl);
            path[sl+tl] = 0;
            dup_inode(bip = path[0] == '/' ? fp->fp_rootdir : ldip);
        }
    }

    put_block(bp, DIRECTORY_BLOCK);
    put_inode(rip);
    if (bip == NIL_INODE)
    {
        err_code = ENOENT;
    }
    return (bip);
}

/*=====
*                               get_name                               *
*=====*/
PRIVATE char *get_name(old_name, string)
char *old_name;              /* path name to parse */
char string[NAME_MAX];       /* component extracted from 'old_name' */
{
/* Given a pointer to a path name in fs space, 'old_name', copy the next
* component to 'string' and pad with zeros. A pointer to that part of
* the name as yet unparsed is returned. Roughly speaking,
* 'get_name' = 'old_name' - 'string'.
*
* This routine follows the standard convention that /usr/ast, /usr//ast,
* //usr///ast and /usr/ast/ are all equivalent.
*/

```

```

register int c;
register char *np, *rnp;

np = string;                                /* 'np' points to current position */
rnp = old_name;                             /* 'rnp' points to unparsed string */
while ( (c = *rnp) == '/') rnp++;           /* skip leading slashes */

/* Copy the unparsed path, 'old_name', to the array, 'string'. */
while ( rnp < &old_name[PATH_MAX] && c != '/' && c != '\0') {
    if (np < &string[NAME_MAX]) *np++ = c;
    c = *++rnp;                             /* advance to next character */
}

/* To make /usr/ast/ equivalent to /usr/ast, skip trailing slashes. */
while (c == '/' && rnp < &old_name[PATH_MAX]) c = *++rnp;

if (np < &string[NAME_MAX]) *np = '\0';     /* Terminate string */

if (rnp >= &old_name[PATH_MAX]) {
    err_code = ENAMETOOLONG;
    return((char *) 0);
}
return(rnp);
}

/*=====
*
*                               advance
*=====*/
PUBLIC struct inode *advance(pdirp, string)
struct inode **pdirp;           /* inode for directory to be searched */
char string[NAME_MAX];         /* component name to look for */
{
    /* Given a directory and a component of a path, look up the component in
     * the directory, find the inode, open it, and return a pointer to its inode
     * slot. If it can't be done, return NIL_INODE.
     */

    register struct inode *rip, *dirp;
    register struct super_block *sp;
    int r, inumb;
    dev_t mnt_dev;
    ino_t numb;

    dirp = *pdirp;

    /* If 'string' is empty, yield same inode straight away. */
    if (string[0] == '\0') { return(get_inode(dirp->i_dev, (int) dirp->i_num)); }

    /* Check for NIL_INODE. */
    if (dirp == NIL_INODE) { return(NIL_INODE); }

    /* If 'string' is not present in the directory, signal error. */
    if ( (r = search_dir(dirp, string, &numb, LOOK_UP)) != OK) {
        err_code = r;
        return(NIL_INODE);
    }

    /* Don't go beyond the current root directory, unless the string is dot2. */
    if (dirp == fp->fp_rootdir && strcmp(string, "..") == 0 && string != dot2)
        return(get_inode(dirp->i_dev, (int) dirp->i_num));

    /* The component has been found in the directory. Get inode. */
    if ( (rip = get_inode(dirp->i_dev, (int) numb)) == NIL_INODE) {
        return(NIL_INODE);
    }

    /* The following test is for "mountpoint/.." where mountpoint is a
     * mountpoint. "." will refer to the root of the mounted filesystem,
     * but has to become a reference to the parent of the 'mountpoint'
     * directory.
     *
     * This case is recognized by the looked up name pointing to a
     * root inode, and the directory in which it is held being a
     * root inode, _and_ the name[1] being '..'. (This is a test for '..')

```

```

    * and excludes '.'.)
    */
    if (rip->i_num == ROOT_INODE)
        if (dirp->i_num == ROOT_INODE) {
            if (string[1] == '.') {
                sp = rip->i_sp;
                if (sp->s_imount != sp->s_isup)
                {
                    /* Release the root inode. Replace by the
                     * inode mounted on. Update parent.
                     */
                    put_inode(rip);
                    put_inode(dirp);
                    mnt_dev = sp->s_imount->i_dev;
                    inumb = (int) sp->s_imount->i_num;
                    dirp = *pdip = get_inode(mnt_dev, inumb);
                    rip = advance(pdip, string);
                }
            }
        }
    if (rip == NIL_INODE) return(NIL_INODE);

    /* See if the inode is mounted on. If so, switch to root directory of the
     * mounted file system. The super_block provides the linkage between the
     * inode mounted on and the root directory of the mounted file system.
     */
    while (rip != NIL_INODE && rip->i_mount == I_MOUNT) {
        /* The inode is indeed mounted on. */
        for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++) {
            if (sp->s_imount == rip) {
                /* Release the inode mounted on. Replace by the
                 * inode of the root inode of the mounted device.
                 */
                put_inode(rip);
                rip = get_inode(sp->s_dev, ROOT_INODE);
                break;
            }
        }
    }
    return(rip);          /* return pointer to inode's component */
}

/*=====
 *                               search_dir                               *
 *=====*/
PUBLIC int search_dir(ldir_ptr, string, numb, flag)
register struct inode *ldir_ptr; /* ptr to inode for dir to search */
char string[NAME_MAX];          /* component to search for */
ino_t *numb;                    /* pointer to inode number */
int flag;                       /* LOOK_UP, ENTER, DELETE or IS_EMPTY */
{
    /* This function searches the directory whose inode is pointed to by 'ldir':
     * if (flag == ENTER) enter 'string' in the directory with inode # '*numb';
     * if (flag == DELETE) delete 'string' from the directory;
     * if (flag == LOOK_UP) search for 'string' and return inode # in 'numb';
     * if (flag == IS_EMPTY) return OK if only . and .. in dir else ENOTEMPTY;
     *
     * if 'string' is dot1 or dot2, no access permissions are checked.
     */

    register struct direct *dp = NULL;
    register struct buf *bp = NULL;
    int i, r, e_hit, t, match;
    mode_t bits;
    off_t pos;
    unsigned new_slots, old_slots;
    block_t b;
    struct super_block *sp;
    int extended = 0;

    /* If 'ldir_ptr' is not a pointer to a dir inode, error. */
    if ( (ldir_ptr->i_mode & I_TYPE) != I_DIRECTORY) {
        return(ENOTDIR);
    }
}

```

```

r = OK;

if (flag != IS_EMPTY) {
    bits = (flag == LOOK_UP ? X_BIT : W_BIT | X_BIT);

    if (string == dot1 || string == dot2) {
        if (flag != LOOK_UP) r = read_only(ldir_ptr);
        /* only a writable device is required. */
    }
    else r = forbidden(ldir_ptr, bits); /* check access permissions */
}
if (r != OK) return(r);

/* Step through the directory one block at a time. */
old_slots = (unsigned) (ldir_ptr->i_size/DIR_ENTRY_SIZE);
new_slots = 0;
e_hit = FALSE;
match = 0; /* set when a string match occurs */

for (pos = 0; pos < ldir_ptr->i_size; pos += ldir_ptr->i_sp->s_block_size) {
    b = read_map(ldir_ptr, pos); /* get block number */

    /* Since directories don't have holes, 'b' cannot be NO_BLOCK. */
    bp = get_block(ldir_ptr->i_dev, b, NORMAL); /* get a dir block */

    if (bp == NO_BLOCK)
        panic(__FILE__, "get_block returned NO_BLOCK", NO_NUM);

    /* Search a directory block. */
    for (dp = &bp->b_dir[0];
         dp < &bp->b_dir[NR_DIR_ENTRIES(ldir_ptr->i_sp->s_block_size)];
         dp++) {
        if (++new_slots > old_slots) { /* not found, but room left */
            if (flag == ENTER) e_hit = TRUE;
            break;
        }

        /* Match occurs if string found. */
        if (flag != ENTER && dp->d_ino != 0) {
            if (flag == IS_EMPTY) {
                /* If this test succeeds, dir is not empty. */
                if (strcmp(dp->d_name, ".") != 0 &&
                    strcmp(dp->d_name, "..") != 0) match = 1;
            } else {
                if (strncmp(dp->d_name, string, NAME_MAX) == 0) {
                    match = 1;
                }
            }
        }

        if (match) {
            /* LOOK_UP or DELETE found what it wanted. */
            r = OK;
            if (flag == IS_EMPTY) r = ENOTEMPTY;
            else if (flag == DELETE) {
                /* Save d_ino for recovery. */
                t = NAME_MAX - sizeof(ino_t);
                *((ino_t *) &dp->d_name[t]) = dp->d_ino;
                dp->d_ino = 0; /* erase entry */
                bp->b_dirt = DIRTY;
                ldir_ptr->i_update |= CTIME | MTIME;
                ldir_ptr->i_dirt = DIRTY;
            } else {
                sp = ldir_ptr->i_sp; /* 'flag' is LOOK_UP */
                *numb = conv4(sp->s_native, (int) dp->d_ino);
            }
            put_block(bp, DIRECTORY_BLOCK);
            return(r);
        }
    }

    /* Check for free slot for the benefit of ENTER. */
    if (flag == ENTER && dp->d_ino == 0) {
        e_hit = TRUE; /* we found a free slot */
    }
}

```



```
                break;
            }
        }

        /* The whole block has been searched or ENTER has a free slot. */
        if (e_hit) break; /* e_hit set if ENTER can be performed now */
        put_block(bp, DIRECTORY_BLOCK); /* otherwise, continue searching dir */
    }

    /* The whole directory has now been searched. */
    if (flag != ENTER) {
        return(flag == IS_EMPTY ? OK : ENOENT);
    }

    /* This call is for ENTER. If no free slot has been found so far, try to
       * extend directory.
       */
    if (e_hit == FALSE) { /* directory is full and no room left in last block */
        new_slots++; /* increase directory size by 1 entry */
        if (new_slots == 0) return(EFBIG); /* dir size limited by slot count */
        if ((bp = new_block(ldir_ptr, ldir_ptr->i_size)) == NIL_BUF)
            return(err_code);
        dp = &bp->b_dir[0];
        extended = 1;
    }

    /* 'bp' now points to a directory block with space. 'dp' points to slot. */
    (void) memset(dp->d_name, 0, (size_t) NAME_MAX); /* clear entry */
    for (i = 0; string[i] && i < NAME_MAX; i++) dp->d_name[i] = string[i];
    sp = ldir_ptr->i_sp;
    dp->d_ino = conv4(sp->s_native, (int) *numb);
    bp->b_dirt = DIRTY;
    put_block(bp, DIRECTORY_BLOCK);
    ldir_ptr->i_update |= CTIME | MTIME; /* mark mtime for update later */
    ldir_ptr->i_dirt = DIRTY;
    if (new_slots > old_slots) {
        ldir_ptr->i_size = (off_t) new_slots * DIR_ENTRY_SIZE;
        /* Send the change to disk if the directory is extended. */
        if (extended) rw_inode(ldir_ptr, WRITING);
    }
    return(OK);
}
```

```

/* This file deals with the suspension and revival of processes. A process can
 * be suspended because it wants to read or write from a pipe and can't, or
 * because it wants to read or write from a special file and can't. When a
 * process can't continue it is suspended, and revived later when it is able
 * to continue.
 *
 * The entry points into this file are
 * do_pipe:      perform the PIPE system call
 * pipe_check:   check to see that a read or write on a pipe is feasible now
 * suspend:      suspend a process that cannot do a requested read or write
 * release:      check to see if a suspended process can be released and do
 *               it
 * revive:       mark a suspended process as able to run again
 * unsuspend_by_endpt: revive all processes blocking on a given process
 * do_unpause:   a signal has been sent to a process; see if it suspended
 */

#include "fs.h"
#include <fcntl.h>
#include <signal.h>
#include <minix/callnr.h>
#include <minix/endpoint.h>
#include <minix/com.h>
#include <sys/select.h>
#include <sys/time.h>
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"
#include "select.h"

/*=====
 *                               do_pipe                               *
 *=====*/
PUBLIC int do_pipe()
{
    /* Perform the pipe(fil_des) system call. */

    register struct fproc *rfp;
    register struct inode *rip;
    int r;
    struct filp *fil_ptr0, *fil_ptr1;
    int fil_des[2];          /* reply goes here */

    /* Acquire two file descriptors. */
    rfp = ffp;
    if ( (r = get_fd(0, R_BIT, &fil_des[0], &fil_ptr0)) != OK) return(r);
    rfp->fp_filp[fil_des[0]] = fil_ptr0;
    FD_SET(fil_des[0], &rfp->fp_filp_inuse);
    fil_ptr0->filp_count = 1;
    if ( (r = get_fd(0, W_BIT, &fil_des[1], &fil_ptr1)) != OK) {
        rfp->fp_filp[fil_des[0]] = NIL_FILP;
        FD_CLR(fil_des[0], &rfp->fp_filp_inuse);
        fil_ptr0->filp_count = 0;
        return(r);
    }
    rfp->fp_filp[fil_des[1]] = fil_ptr1;
    FD_SET(fil_des[1], &rfp->fp_filp_inuse);
    fil_ptr1->filp_count = 1;

    /* Make the inode on the pipe device. */
    if ( (rip = alloc_inode(root_dev, I_REGULAR) ) == NIL_INODE) {
        rfp->fp_filp[fil_des[0]] = NIL_FILP;
        FD_CLR(fil_des[0], &rfp->fp_filp_inuse);
        fil_ptr0->filp_count = 0;
        rfp->fp_filp[fil_des[1]] = NIL_FILP;
        FD_CLR(fil_des[1], &rfp->fp_filp_inuse);
        fil_ptr1->filp_count = 0;
        return(err_code);
    }

    if (read_only(rip) != OK)
        panic(__FILE__, "pipe device is read only", NO_NUM);
}

```

```

rip->i_pipe = I_PIPE;
rip->i_mode &= ~I_REGULAR;
rip->i_mode |= I_NAMED_PIPE; /* pipes and FIFOs have this bit set */
fil_ptr0->filp_ino = rip;
fil_ptr0->filp_flags = O_RDONLY;
dup_inode(rip); /* for double usage */
fil_ptr1->filp_ino = rip;
fil_ptr1->filp_flags = O_WRONLY;
rw_inode(rip, WRITING); /* mark inode as allocated */
m_out.reply_i1 = fil_des[0];
m_out.reply_i2 = fil_des[1];
rip->i_update = ATIME | CTIME | MTIME;
return(OK);
}

/*=====
*                                     pipe_check                                     *
*=====*/
PUBLIC int pipe_check(rip, rw_flag, oflags, bytes, position, canwrite, notouch)
register struct inode *rip; /* the inode of the pipe */
int rw_flag; /* READING or WRITING */
int oflags; /* flags set by open or fcntl */
register int bytes; /* bytes to be read or written (all chunks) */
register off_t position; /* current file position */
int *canwrite; /* return: number of bytes we can write */
int notouch; /* check only */
{
/* Pipes are a little different. If a process reads from an empty pipe for
* which a writer still exists, suspend the reader. If the pipe is empty
* and there is no writer, return 0 bytes. If a process is writing to a
* pipe and no one is reading from it, give a broken pipe error.
*/

/* If reading, check for empty pipe. */
if (rw_flag == READING) {
    if (position >= rip->i_size) {
        /* Process is reading from an empty pipe. */
        int r = 0;
        if (find_filp(rip, W_BIT) != NIL_FILP) {
            /* Writer exists */
            if (oflags & O_NONBLOCK) {
                r = EAGAIN;
            } else {
                if (!notouch)
                    suspend(XPIPE); /* block reader */
                r = SUSPEND;
            }
            /* If need be, activate sleeping writers. */
            if (susp_count > 0 && !notouch)
                release(rip, WRITE, susp_count);
        }
        return(r);
    }
} else {
    /* Process is writing to a pipe. */
    if (find_filp(rip, R_BIT) == NIL_FILP) {
        /* Tell kernel to generate a SIGPIPE signal. */
        if (!notouch) {
            sys_kill(fp->fp_endpoint, SIGPIPE);
        }
        return(EPIPE);
    }

    if (position + bytes > PIPE_SIZE(rip->i_sp->s_block_size)) {
        if ((oflags & O_NONBLOCK)
            && bytes < PIPE_SIZE(rip->i_sp->s_block_size))
            return(EAGAIN);
        else if ((oflags & O_NONBLOCK)
            && bytes > PIPE_SIZE(rip->i_sp->s_block_size)) {
            if ((*canwrite = (PIPE_SIZE(rip->i_sp->s_block_size)
                - position)) > 0) {
                /* Do a partial write. Need to wakeup reader */
                if (!notouch)

```

```

        release(rip, READ, susp_count);
        return(1);
    } else {
        return(EAGAIN);
    }
}

if (bytes > PIPE_SIZE(rip->i_sp->s_block_size)) {
    if ((*canwrite = PIPE_SIZE(rip->i_sp->s_block_size)
        - position) > 0) {
        /* Do a partial write. Need to wakeup reader
         * since we'll suspend ourself in read_write()
         */
        if (!notouch)
            release(rip, READ, susp_count);
        return(1);
    }
}

if (!notouch)
    suspend(XPIPE); /* stop writer -- pipe full */
return(SUSPEND);
}

/* Writing to an empty pipe. Search for suspended reader. */
if (position == 0 && !notouch)
    release(rip, READ, susp_count);
}

*canwrite = 0;
return(1);
}

/*=====
 *                               suspend                               *
 *=====*/
PUBLIC void suspend(task)
int task;                /* who is proc waiting for? (PIPE = pipe) */
{
    /* Take measures to suspend the processing of the present system call.
     * Store the parameters to be used upon resuming in the process table.
     * (Actually they are not used when a process is waiting for an I/O device,
     * but they are needed for pipes, and it is not worth making the distinction.)
     * The SUSPEND pseudo error should be returned after calling suspend().
     */

    if (task == XPIPE || task == XPOPEN) susp_count++; /* #procs susp'ed on pipe*/
    fp->fp_suspended = SUSPENDED;
    fp->fp_fd = m_in.fd << 8 | call_nr;
    if(task == NONE)
        panic(__FILE__, "suspend on NONE", NO_NUM);
    fp->fp_task = -task;
    if (task == XLOCK) {
        fp->fp_buffer = (char *) m_in.name1; /* third arg to fcntl() */
        fp->fp_nbytes = m_in.request; /* second arg to fcntl() */
    } else {
        fp->fp_buffer = m_in.buffer; /* for reads and writes */
        fp->fp_nbytes = m_in.nbytes;
    }
}

/*=====
 *                               unsuspend_by_endpt                               *
 *=====*/
PUBLIC void unsuspend_by_endpt(int proc_e)
{
    struct fproc *rp;
    int client = 0;

    /* Revive processes waiting for drivers (SUSPENDED) that have
     * disappeared with return code EAGAIN.
     */
    for (rp = &fproc[0]; rp < &fproc[NR_PROCS]; rp++, client++)
        if(rp->fp_pid != PID_FREE &&
            rp->fp_suspended == SUSPENDED && rp->fp_task == -proc_e) {
            revive(rp->fp_endpoint, EAGAIN);
        }
}

```

```

    }

    /* Revive processes waiting in drivers on select()s
     * with EAGAIN too.
     */
    select_unsuspend_by_endpt(proc_e);

    return;
}

/*=====
 *                               release                               *
 *=====*/
PUBLIC void release(ip, call_nr, count)
register struct inode *ip;      /* inode of pipe */
int call_nr;                   /* READ, WRITE, OPEN or CREAT */
int count;                     /* max number of processes to release */
{
    /* Check to see if any process is hanging on the pipe whose inode is in 'ip'.
     * If one is, and it was trying to perform the call indicated by 'call_nr',
     * release it.
     */

    register struct fproc *rp;
    struct filp *f;

    /* Trying to perform the call also includes SELECTing on it with that
     * operation.
     */
    if (call_nr == READ || call_nr == WRITE) {
        int op;
        if (call_nr == READ)
            op = SEL_RD;
        else
            op = SEL_WR;
        for(f = &filp[0]; f < &filp[NR_FILPS]; f++) {
            if (f->filp_count < 1 || !(f->filp_pipe_select_ops & op) ||
                f->filp_ino != ip)
                continue;
            select_callback(f, op);
            f->filp_pipe_select_ops &= ~op;
        }
    }

    /* Search the proc table. */
    for (rp = &fproc[0]; rp < &fproc[NR_PROCS]; rp++) {
        if (rp->fp_pid != PID_FREE && rp->fp_suspended == SUSPENDED &&
            rp->fp_revived == NOT_REVIVING &&
            (rp->fp_fd & BYTE) == call_nr &&
            rp->fp_filp[rp->fp_fd>>8]->filp_ino == ip) {
            revive(rp->fp_endpoint, 0);
            susp_count--; /* keep track of who is suspended */
            if (--count == 0) return;
        }
    }
}

/*=====
 *                               revive                               *
 *=====*/
PUBLIC void revive(proc_nr_e, returned)
int proc_nr_e;                 /* process to revive */
int returned;                   /* if hanging on task, how many bytes read */
{
    /* Revive a previously blocked process. When a process hangs on tty, this
     * is the way it is eventually released.
     */

    register struct fproc *rfp;
    register int task;
    int proc_nr;

    if(isokendpt(proc_nr_e, &proc_nr) != OK)

```

```

    return;

    rfp = &fproc[proc_nr];
    if (rfp->fp_suspended == NOT_SUSPENDED || rfp->fp_revived == REVIVING) return;

    /* The 'reviving' flag only applies to pipes. Processes waiting for TTY get
     * a message right away. The revival process is different for TTY and pipes.
     * For select and TTY revival, the work is already done, for pipes it is not:
     * the proc must be restarted so it can try again.
     */
    task = -rfp->fp_task;
    if (task == XPIPE || task == XLOCK) {
        /* Revive a process suspended on a pipe or lock. */
        rfp->fp_revived = REVIVING;
        reviving++;
        /* process was waiting on pipe or lock */
    } else {
        rfp->fp_suspended = NOT_SUSPENDED;
        if (task == XOPEN) /* process blocked in open or create */
            reply(proc_nr_e, rfp->fp_fd>>8);
        else if (task == XSELECT) {
            reply(proc_nr_e, returned);
        } else {
            /* Revive a process suspended on TTY or other device. */
            rfp->fp_nbytes = returned; /*pretend it wants only what there is*/
            reply(proc_nr_e, returned); /* unblock the process */
        }
    }
}

/*=====
 *                               do_unpause                               *
 *=====*/
PUBLIC int do_unpause()
{
    /* A signal has been sent to a user who is paused on the file system.
     * Abort the system call with the EINTR error message.
     */
    int proc_nr_e;

    if (who_e != PM_PROC_NR) return(EPERM);
    proc_nr_e = m_in.ENDPT;
    return unpause(proc_nr_e);
}

/*=====
 *                               unpause                               *
 *=====*/
PUBLIC int unpause(proc_nr_e)
int proc_nr_e;
{
    /* A signal has been sent to a user who is paused on the file system.
     * Abort the system call with the EINTR error message.
     */

    register struct fproc *rfp;
    int proc_nr_p, task, fild;
    struct filp *f;
    dev_t dev;
    message mess;

    okendpt(proc_nr_e, &proc_nr_p);
    rfp = &fproc[proc_nr_p];
    if (rfp->fp_suspended == NOT_SUSPENDED) return(OK);
    task = -rfp->fp_task;

    if (rfp->fp_revived == REVIVING)
        reviving--;

    switch (task) {
        case XPIPE:
            /* process trying to read or write a pipe */
            break;

        case XLOCK:
            /* process trying to set a lock with FCNTL */
            break;
    }

```

```

    case XSELECT:          /* process blocking on select() */
        select_forget(proc_nr_e);
        break;

    case XPOPEN:           /* process trying to open a fifo */
        break;

    default:               /* process trying to do device I/O (e.g. tty)*/
        fild = (rfp->fp_fd >> 8) & BYTE; /* extract file descriptor */
        if (fil_d < 0 || fil_d >= OPEN_MAX)
            panic(__FILE__, "unpause err 2", NO_NUM);
        f = rfp->fp_filp[fil_d];
        dev = (dev_t) f->filp_ino->i_zone[0]; /* device hung on */
        mess.TTY_LINE = (dev >> MINOR) & BYTE;
        mess.IO_ENDPT = proc_nr_e;

        /* Tell kernel R or W. Mode is from current call, not open. */
        mess.COUNT = (rfp->fp_fd & BYTE) == READ ? R_BIT : W_BIT;
        mess.m_type = CANCEL;
        fp = rfp; /* hack - cttty_io uses fp */
        (*dmap[(dev >> MAJOR) & BYTE].dmap_io)(task, &mess);
}

rfp->fp_suspended = NOT_SUSPENDED;
reply(proc_nr_e, EINTR); /* signal interrupted call */
return(OK);
}

/*=====
*                               select_request_pipe                               *
*=====*/
PUBLIC int select_request_pipe(struct filp *f, int *ops, int block)
{
    int orig_ops, r = 0, err, canwrite;
    orig_ops = *ops;
    if ((*ops & (SEL_RD|SEL_ERR))) {
        if ((err = pipe_check(f->filp_ino, READING, 0,
            1, f->filp_pos, &canwrite, 1)) != SUSPEND && err > 0)
            r |= SEL_RD;
        if (err < 0 && err != SUSPEND)
            r |= SEL_ERR;
    }
    if ((*ops & (SEL_WR|SEL_ERR))) {
        if ((err = pipe_check(f->filp_ino, WRITING, 0,
            1, f->filp_pos, &canwrite, 1)) != SUSPEND &&
            err > 0 && canwrite > 0)
            r |= SEL_WR;
        if (err < 0 && err != SUSPEND)
            r |= SEL_ERR;
    }

    /* Some options we collected might not be requested. */
    *ops = r & orig_ops;

    if (!*ops && block) {
        f->filp_pipe_select_ops |= orig_ops;
    }

    return SEL_OK;
}

/*=====
*                               select_match_pipe                               *
*=====*/
PUBLIC int select_match_pipe(struct filp *f)
{
    /* recognize either pipe or named pipe (FIFO) */
    if (f && f->filp_ino && (f->filp_ino->i_mode & I_NAMED_PIPE))
        return 1;
    return 0;
}

```

```

/* This file deals with protection in the file system.  It contains the code
 * for four system calls that relate to protection.
 *
 * The entry points into this file are
 * do_chmod: perform the CHMOD and FCHMOD system calls
 * do_chown: perform the CHOWN and FCHOWN system calls
 * do_umask: perform the UMASK system call
 * do_access: perform the ACCESS system call
 * forbidden: check to see if a given access is allowed on a given inode
 */

#include "fs.h"
#include <unistd.h>
#include <minix/callnr.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

/*=====
 *
 * do_chmod
 *=====*/
PUBLIC int do_chmod()
{
    /* Perform the chmod(name, mode) system call. */

    register struct inode *rip;
    register int r;

    if(call_nr == CHMOD) {
        /* Temporarily open the file. */
        if (fetch_name(m_in.name, m_in.name_length, M3) != OK) return(err_code);
        if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
    } else if(call_nr == FCHMOD) {
        struct filp *filp;
        if(!(filp = get_filp(m_in.m3_i1))) return(err_code);
        rip = filp->filp_ino;
    } else panic(__FILE__, "do_chmod called with strange call_nr", call_nr);

    /* Only the owner or the super_user may change the mode of a file.
     * No one may change the mode of a file on a read-only file system.
     */
    if (rip->i_uid != fp->fp_effuid && !super_user)
        r = EPERM;
    else
        r = read_only(rip);

    /* If error, return inode. */
    if (r != OK) {
        if(call_nr == CHMOD) put_inode(rip);
        return(r);
    }

    /* Now make the change. Clear setgid bit if file is not in caller's grp */
    rip->i_mode = (rip->i_mode & ~ALL_MODES) | (m_in.mode & ALL_MODES);
    if (!super_user && rip->i_gid != fp->fp_effgid) rip->i_mode &= ~I_SET_GID_BIT;
    rip->i_update |= CTIME;
    rip->i_dirt = DIRTY;

    if(call_nr == CHMOD) put_inode(rip);
    return(OK);
}

/*=====
 *
 * do_chown
 *=====*/
PUBLIC int do_chown()
{
    /* Perform the chown(name, owner, group) system call. */

    register struct inode *rip;
    register int r;

```



```

if(call_nr == CHOWN) {
    /* Temporarily open the file. */
    if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
    if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
} else if(call_nr == FCHOWN) {
    struct filp *filp;
    if(!(filp = get_filp(m_in.m1_i1))) return(err_code);
    rip = filp->filp_ino;
} else panic(__FILE__, "do_chown called with strange call_nr", call_nr);

/* Not permitted to change the owner of a file on a read-only file sys. */
r = read_only(rip);
if (r == OK) {
    /* FS is R/W. Whether call is allowed depends on ownership, etc. */
    if (super_user) {
        /* The super user can do anything. */
        rip->i_uid = m_in.owner; /* others later */
    } else {
        /* Regular users can only change groups of their own files. */
        if (rip->i_uid != fp->fp_effuid) r = EPERM;
        if (rip->i_uid != m_in.owner) r = EPERM; /* no giving away */
        if (fp->fp_effgid != m_in.group) r = EPERM;
    }
}
if (r == OK) {
    rip->i_gid = m_in.group;
    rip->i_mode &= ~(I_SET_UID_BIT | I_SET_GID_BIT);
    rip->i_update |= CTIME;
    rip->i_dirt = DIRTY;
}

if(call_nr == CHOWN) put_inode(rip);
return(r);
}

/*=====
*
* do_umask
*=====*/
PUBLIC int do_umask()
{
    /* Perform the umask(co_mode) system call. */
    register mode_t r;

    r = ~fp->fp_umask; /* set 'r' to complement of old mask */
    fp->fp_umask = ~(m_in.co_mode & RWX_MODES);
    return(r); /* return complement of old mask */
}

/*=====
*
* do_access
*=====*/
PUBLIC int do_access()
{
    /* Perform the access(name, mode) system call. */

    struct inode *rip;
    register int r;

    /* First check to see if the mode is correct. */
    if ( (m_in.mode & ~(R_OK | W_OK | X_OK)) != 0 && m_in.mode != F_OK)
        return(EINVAL);

    /* Temporarily open the file whose access is to be checked. */
    if (fetch_name(m_in.name, m_in.name_length, M3) != OK) return(err_code);
    if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);

    /* Now check the permissions. */
    r = forbidden(rip, (mode_t) m_in.mode);
    put_inode(rip);
    return(r);
}

/*=====

```

```

*                               forbidden                               *
*=====*/
PUBLIC int forbidden(register struct inode *rip, mode_t access_desired)
{
/* Given a pointer to an inode, 'rip', and the access desired, determine
* if the access is allowed, and if not why not. The routine looks up the
* caller's uid in the 'fproc' table. If access is allowed, OK is returned
* if it is forbidden, EACCES is returned.
*/

register struct inode *old_rip = rip;
register struct super_block *sp;
register mode_t bits, perm_bits;
int r, shift, test_uid, test_gid, type;

if (rip->i_mount == I_MOUNT) /* The inode is mounted on. */
    for (sp = &super_block[1]; sp < &super_block[NR_SUPERS]; sp++)
        if (sp->s_imount == rip) {
            rip = get_inode(sp->s_dev, ROOT_INODE);
            break;
        } /* if */

/* Isolate the relevant rwx bits from the mode. */
bits = rip->i_mode;
test_uid = (call_nr == ACCESS ? fp->fp_realuid : fp->fp_effuid);
test_gid = (call_nr == ACCESS ? fp->fp_realgid : fp->fp_effgid);
if (test_uid == SU_UID) {
    /* Grant read and write permission. Grant search permission for
    * directories. Grant execute permission (for non-directories) if
    * and only if one of the 'X' bits is set.
    */
    if ( (bits & I_TYPE) == I_DIRECTORY ||
        bits & ((X_BIT << 6) | (X_BIT << 3) | X_BIT))
        perm_bits = R_BIT | W_BIT | X_BIT;
    else
        perm_bits = R_BIT | W_BIT;
} else {
    if (test_uid == rip->i_uid) shift = 6; /* owner */
    else if (test_gid == rip->i_gid) shift = 3; /* group */
    else shift = 0; /* other */
    perm_bits = (bits >> shift) & (R_BIT | W_BIT | X_BIT);
}

/* If access desired is not a subset of what is allowed, it is refused. */
r = OK;
if ((perm_bits | access_desired) != perm_bits) r = EACCES;

/* Check to see if someone is trying to write on a file system that is
* mounted read-only.
*/
type = rip->i_mode & I_TYPE;
if (r == OK)
    if (access_desired & W_BIT)
        r = read_only(rip);

if (rip != old_rip) put_inode(rip);

return(r);
}

/*=====*
*                               read_only                               *
*=====*/
PUBLIC int read_only(ip)
struct inode *ip; /* ptr to inode whose file sys is to be ckd */
{
/* Check to see if the file system on which the inode 'ip' resides is mounted
* read only. If so, return EROFS, else return OK.
*/

register struct super_block *sp;

sp = ip->i_sp;
return(sp->s_rd_only ? EROFS : OK);
}

```

```
}
```

```

/* Function prototypes. */

#include "timers.h"

/* Structs used in prototypes must be declared as such first. */
struct buf;
struct filp;
struct fproc;
struct inode;
struct super_block;

/* cache.c */
_PROTOTYPE( zone_t alloc_zone, (Dev_t dev, zone_t z) );
_PROTOTYPE( void flushall, (Dev_t dev) );
_PROTOTYPE( void free_zone, (Dev_t dev, zone_t numb) );
_PROTOTYPE( struct buf *get_block, (Dev_t dev, block_t block, int only_search));
_PROTOTYPE( void invalidate, (Dev_t device) );
_PROTOTYPE( void put_block, (struct buf *bp, int block_type) );
_PROTOTYPE( void rw_scattered, (Dev_t dev,
                               struct buf **bufq, int bufqsize, int rw_flag) );

#if ENABLE_CACHE2
/* cache2.c */
_PROTOTYPE( void init_cache2, (unsigned long size) );
_PROTOTYPE( int get_block2, (struct buf *bp, int only_search) );
_PROTOTYPE( void put_block2, (struct buf *bp) );
_PROTOTYPE( void invalidate2, (Dev_t device) );
#endif

/* device.c */
_PROTOTYPE( int dev_open, (Dev_t dev, int proc, int flags) );
_PROTOTYPE( void dev_close, (Dev_t dev) );
_PROTOTYPE( int dev_bio, (int op, Dev_t dev, int proc, void *buf,
                          off_t pos, int bytes, int flags) );
_PROTOTYPE( int dev_io, (int op, Dev_t dev, int proc, void *buf,
                         off_t pos, int bytes, int flags) );
_PROTOTYPE( int gen_opcl, (int op, Dev_t dev, int proc, int flags) );
_PROTOTYPE( int gen_io, (int task_nr, message *mess_ptr) );
_PROTOTYPE( int no_dev, (int op, Dev_t dev, int proc, int flags) );
_PROTOTYPE( int no_dev_io, (int, message *) );
_PROTOTYPE( int tty_opcl, (int op, Dev_t dev, int proc, int flags) );
_PROTOTYPE( int ctty_opcl, (int op, Dev_t dev, int proc, int flags) );
_PROTOTYPE( int clone_opcl, (int op, Dev_t dev, int proc, int flags) );
_PROTOTYPE( int ctty_io, (int task_nr, message *mess_ptr) );
_PROTOTYPE( int do_ioctl, (void) );
_PROTOTYPE( void pm_setsid, (int proc_e) );
_PROTOTYPE( void dev_status, (message *) );
_PROTOTYPE( void dev_up, (int major) );

/* dmp.c */
_PROTOTYPE( int do_fkey_pressed, (void) );

/* dmap.c */
_PROTOTYPE( int do_devctl, (void) );
_PROTOTYPE( int fs_devctl, (int req, int dev, int proc_nr_e, int style,
                           int force) );
_PROTOTYPE( void build_dmap, (void) );
_PROTOTYPE( int map_driver, (int major, int proc_nr, int dev_style,
                           int force) );
_PROTOTYPE( int dmap_driver_match, (int proc, int major) );
_PROTOTYPE( void dmap_unmap_by_endpt, (int proc_nr) );
_PROTOTYPE( void dmap_endpt_up, (int proc_nr) );

/* exec.c */
_PROTOTYPE( int pm_exec, (int proc_e, char *path, vir_bytes path_len,
                          char *frame, vir_bytes frame_len) );

/* filedес.c */
_PROTOTYPE( struct filp *find_filp, (struct inode *rip, mode_t bits) );
_PROTOTYPE( int get_fd, (int start, mode_t bits, int *k, struct filp **fpt) );
_PROTOTYPE( struct filp *get_filp, (int fild) );
_PROTOTYPE( struct filp *get_filp2, (struct fproc *rfp, int fild) );
_PROTOTYPE( int inval_filp, (struct filp *) );

```

```

/* inode.c */
_PROTOTYPE( struct inode *alloc_inode, (dev_t dev, mode_t bits) );
_PROTOTYPE( void dup_inode, (struct inode *ip) );
_PROTOTYPE( void free_inode, (Dev_t dev, Ino_t numb) );
_PROTOTYPE( struct inode *get_inode, (Dev_t dev, int numb) );
_PROTOTYPE( void put_inode, (struct inode *rip) );
_PROTOTYPE( void update_times, (struct inode *rip) );
_PROTOTYPE( void rw_inode, (struct inode *rip, int rw_flag) );
_PROTOTYPE( void wipe_inode, (struct inode *rip) );

/* link.c */
_PROTOTYPE( int do_link, (void) );
_PROTOTYPE( int do_unlink, (void) );
_PROTOTYPE( int do_rename, (void) );
_PROTOTYPE( int do_truncate, (void) );
_PROTOTYPE( int do_ftruncate, (void) );
_PROTOTYPE( int truncate_inode, (struct inode *rip, off_t len) );
_PROTOTYPE( int freesp_inode, (struct inode *rip, off_t st, off_t end) );

/* lock.c */
_PROTOTYPE( int lock_op, (struct filp *f, int req) );
_PROTOTYPE( void lock_revive, (void) );

/* main.c */
_PROTOTYPE( int main, (void) );
_PROTOTYPE( void reply, (int whom, int result) );

/* misc.c */
_PROTOTYPE( int do_dup, (void) );
_PROTOTYPE( void pm_exit, (int proc) );
_PROTOTYPE( int do_fcntl, (void) );
_PROTOTYPE( void pm_fork, (int pproc, int cproc, int cpid) );
_PROTOTYPE( void pm_setgid, (int proc_e, int egid, int rgid) );
_PROTOTYPE( void pm_setuid, (int proc_e, int euid, int ruid) );
_PROTOTYPE( int do_sync, (void) );
_PROTOTYPE( int do_fsync, (void) );
_PROTOTYPE( void pm_reboot, (void) );
_PROTOTYPE( int do_svrctl, (void) );
_PROTOTYPE( int do_getsysinfo, (void) );
_PROTOTYPE( int pm_dumpcore, (int proc_e, struct mem_map *seg_ptr) );

/* mount.c */
_PROTOTYPE( int do_mount, (void) );
_PROTOTYPE( int do_umount, (void) );
_PROTOTYPE( int unmount, (Dev_t dev) );

/* open.c */
_PROTOTYPE( int do_close, (void) );
_PROTOTYPE( int close_fd, (struct fproc *rfp, int fd_nr) );
_PROTOTYPE( int do_creat, (void) );
_PROTOTYPE( int do_lseek, (void) );
_PROTOTYPE( int do_mknod, (void) );
_PROTOTYPE( int do_mkdir, (void) );
_PROTOTYPE( int do_open, (void) );
_PROTOTYPE( int do_slink, (void) );
_PROTOTYPE( struct inode *new_node, (struct inode **ldirp,
    char *path, mode_t bits, zone_t z0, int opaque, char *string) );

/* path.c */
_PROTOTYPE( struct inode *advance, (struct inode **dirp, char string[NAME_MAX]));
_PROTOTYPE( int search_dir, (struct inode *ldir_ptr,
    char string [NAME_MAX], ino_t *numb, int flag) );
_PROTOTYPE( struct inode *eat_path, (char *path) );
_PROTOTYPE( struct inode *last_dir, (char *path, char string [NAME_MAX]));
_PROTOTYPE( struct inode *parse_path, (char *path, char string[NAME_MAX],
    int action) );

/* pipe.c */
_PROTOTYPE( int do_pipe, (void) );
_PROTOTYPE( int do_unpause, (void) );
_PROTOTYPE( int unpause, (int proc_nr_e) );
_PROTOTYPE( int pipe_check, (struct inode *rip, int rw_flag,
    int oflags, int bytes, off_t position, int *canwrite, int notouch
));

```

```

_PROTOTYPE( void release, (struct inode *ip, int call_nr, int count) ) ;
_PROTOTYPE( void revive, (int proc_nr, int bytes) ) ;
_PROTOTYPE( void suspend, (int task) ) ;
_PROTOTYPE( int select_request_pipe, (struct filp *f, int *ops, int bl) ) ;
_PROTOTYPE( int select_cancel_pipe, (struct filp *f) ) ;
_PROTOTYPE( int select_match_pipe, (struct filp *f) ) ;
_PROTOTYPE( void unsuspend_by_endpt, (int) ) ;

/* protect.c */
_PROTOTYPE( int do_access, (void) ) ;
_PROTOTYPE( int do_chmod, (void) ) ;
_PROTOTYPE( int do_chown, (void) ) ;
_PROTOTYPE( int do_umask, (void) ) ;
_PROTOTYPE( int forbidden, (struct inode *rip, mode_t access_desired) ) ;
_PROTOTYPE( int read_only, (struct inode *ip) ) ;

/* read.c */
_PROTOTYPE( int do_read, (void) ) ;
_PROTOTYPE( struct buf *rahead, (struct inode *rip, block_t baseblock,
                                off_t position, unsigned bytes_ahead) ) ;
_PROTOTYPE( void read_ahead, (void) ) ;
_PROTOTYPE( block_t read_map, (struct inode *rip, off_t pos) ) ;
_PROTOTYPE( int read_write, (int rw_flag) ) ;
_PROTOTYPE( zone_t rd_indir, (struct buf *bp, int index) ) ;

/* stadir.c */
_PROTOTYPE( int do_chdir, (void) ) ;
_PROTOTYPE( int do_fchdir, (void) ) ;
_PROTOTYPE( int do_chroot, (void) ) ;
_PROTOTYPE( int do_fstat, (void) ) ;
_PROTOTYPE( int do_stat, (void) ) ;
_PROTOTYPE( int do_fstatfs, (void) ) ;
_PROTOTYPE( int do_rdlink, (void) ) ;
_PROTOTYPE( int do_lstat, (void) ) ;

/* super.c */
_PROTOTYPE( bit_t alloc_bit, (struct super_block *sp, int map, bit_t origin));
_PROTOTYPE( void free_bit, (struct super_block *sp, int map,
                           bit_t bit_returned) ) ;
_PROTOTYPE( struct super_block *get_super, (Dev_t dev) ) ;
_PROTOTYPE( int mounted, (struct inode *rip) ) ;
_PROTOTYPE( int read_super, (struct super_block *sp) ) ;
_PROTOTYPE( int get_block_size, (dev_t dev) ) ;

/* time.c */
_PROTOTYPE( int do_stime, (void) ) ;
_PROTOTYPE( int do_utime, (void) ) ;

/* utility.c */
_PROTOTYPE( time_t clock_time, (void) ) ;
_PROTOTYPE( unsigned conv2, (int norm, int w) ) ;
_PROTOTYPE( long conv4, (int norm, long x) ) ;
_PROTOTYPE( int fetch_name, (char *path, int len, int flag) ) ;
_PROTOTYPE( int no_sys, (void) ) ;
_PROTOTYPE( int isokendpt_f, (char *f, int l, int e, int *p, int ft));
_PROTOTYPE( void panic, (char *who, char *mess, int num) ) ;

#define okendpt(e, p) isokendpt_f(__FILE__, __LINE__, (e), (p), 1)
#define isokendpt(e, p) isokendpt_f(__FILE__, __LINE__, (e), (p), 0)

/* write.c */
_PROTOTYPE( void clear_zone, (struct inode *rip, off_t pos, int flag) ) ;
_PROTOTYPE( int do_write, (void) ) ;
_PROTOTYPE( struct buf *new_block, (struct inode *rip, off_t position) ) ;
_PROTOTYPE( void zero_block, (struct buf *bp) ) ;
_PROTOTYPE( int write_map, (struct inode *, off_t, zone_t, int) ) ;

/* select.c */
_PROTOTYPE( int do_select, (void) ) ;
_PROTOTYPE( int select_callback, (struct filp *, int ops) ) ;
_PROTOTYPE( void select_forget, (int fproc) ) ;
_PROTOTYPE( void select_timeout_check, (timer_t *) ) ;
_PROTOTYPE( void init_select, (void) ) ;
_PROTOTYPE( void select_unsuspend_by_endpt, (int proc) ) ;

```

```
_PROTOTYPE( int select_notified, (int major, int minor, int ops)      );  
  
/* timers.c */  
_PROTOTYPE( void fs_set_timer, (timer_t *tp, int delta, tmr_func_t watchdog, int arg));  
_PROTOTYPE( void fs_expire_timers, (clock_t now)                        );  
_PROTOTYPE( void fs_cancel_timer, (timer_t *tp)                        );  
_PROTOTYPE( void fs_init_timer, (timer_t *tp)                          );
```

```

/* This file contains the heart of the mechanism used to read (and write)
 * files. Read and write requests are split up into chunks that do not cross
 * block boundaries. Each chunk is then processed in turn. Reads on special
 * files are also detected and handled.
 *
 * The entry points into this file are
 * do_read: perform the READ system call by calling read_write
 * read_write: actually do the work of READ and WRITE
 * read_map: given an inode and file position, look up its zone number
 * rd_indir: read an entry in an indirect block
 * read_ahead: manage the block read ahead business
 */

#include "fs.h"
#include <fcntl.h>
#include <unistd.h>
#include <minix/com.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

FORWARD _PROTOTYPE( int rw_chunk, (struct inode *rip, off_t position,
    unsigned off, int chunk, unsigned left, int rw_flag,
    char *buff, int seg, int usr, int block_size, int *completed));

/*=====
 * do_read
 *=====*/
PUBLIC int do_read()
{
    return(read_write(READING));
}

/*=====
 * read_write
 *=====*/
PUBLIC int read_write(rw_flag)
int rw_flag; /* READING or WRITING */
{
    /* Perform read(fd, buffer, nbytes) or write(fd, buffer, nbytes) call. */

    register struct inode *rip;
    register struct filp *f;
    off_t bytes_left, f_size, position;
    unsigned int off, cum_io;
    int op, oflags, r, chunk, usr, seg, block_spec, char_spec;
    int regular, partial_pipe = 0, partial_cnt = 0;
    mode_t mode_word;
    struct filp *wf;
    int block_size;
    int completed, r2 = OK;
    phys_bytes p;

    /* PM loads segments by putting funny things in other bits of the
     * message, indicated by a high bit in fd.
     */
    if (who_e == PM_PROC_NR && (m_in.fd & _PM_SEG_FLAG)) {
        seg = (int) m_in.ml_p2;
        usr = (int) m_in.ml_p3;
        m_in.fd &= ~(_PM_SEG_FLAG); /* get rid of flag bit */
    } else {
        usr = who_e; /* normal case */
        seg = D;
    }

    /* If the file descriptor is valid, get the inode, size and mode. */
    if (m_in.nbytes < 0) return(EINVAL);
    if ((f = get_filp(m_in.fd)) == NIL_FILP) return(err_code);
    if (((f->filp_mode) & (rw_flag == READING ? R_BIT : W_BIT)) == 0) {
        return(f->filp_mode == FILP_CLOSED ? EIO : EBADF);
    }

```



```

if (m_in.nbytes == 0)
    return(0);    /* so char special files need not check for 0*/

/* check if user process has the memory it needs.
 * if not, copying will fail later.
 * do this after 0-check above because umap doesn't want to map 0 bytes.
 */
if ((r = sys_umap(usr, seg, (vir_bytes) m_in.buffer, m_in.nbytes, &p)) != OK) {
    printf("FS: read_write: umap failed for process %d\n", usr);
    return r;
}
position = f->filp_pos;
oflags = f->filp_flags;
rip = f->filp_ino;
f_size = rip->i_size;
r = OK;
if (rip->i_pipe == I_PIPE) {
    /* fp->fp_cum_io_partial is only nonzero when doing partial writes */
    cum_io = fp->fp_cum_io_partial;
} else {
    cum_io = 0;
}
op = (rw_flag == READING ? DEV_READ : DEV_WRITE);
mode_word = rip->i_mode & I_TYPE;
regular = mode_word == I_REGULAR || mode_word == I_NAMED_PIPE;

if ((char_spec = (mode_word == I_CHAR_SPECIAL ? 1 : 0))) {
    if (rip->i_zone[0] == NO_DEV)
        panic(__FILE__, "read_write tries to read from "
              "character device NO_DEV", NO_NUM);
    block_size = get_block_size(rip->i_zone[0]);
}
if ((block_spec = (mode_word == I_BLOCK_SPECIAL ? 1 : 0))) {
    f_size = ULONG_MAX;
    if (rip->i_zone[0] == NO_DEV)
        panic(__FILE__, "read_write tries to read from "
              "block device NO_DEV", NO_NUM);
    block_size = get_block_size(rip->i_zone[0]);
}

if (!char_spec && !block_spec)
    block_size = rip->i_sp->s_block_size;

rdwt_err = OK;    /* set to EIO if disk error occurs */

/* Check for character special files. */
if (char_spec) {
    dev_t dev;
    dev = (dev_t) rip->i_zone[0];
    r = dev_io(op, dev, usr, m_in.buffer, position, m_in.nbytes, oflags);
    if (r >= 0) {
        cum_io = r;
        position += r;
        r = OK;
    }
} else {
    if (rw_flag == WRITING && block_spec == 0) {
        /* Check in advance to see if file will grow too big. */
        if (position > rip->i_sp->s_max_size - m_in.nbytes)
            return(EFBIG);

        /* Check for O_APPEND flag. */
        if (oflags & O_APPEND) position = f_size;

        /* Clear the zone containing present EOF if hole about
         * to be created. This is necessary because all unwritten
         * blocks prior to the EOF must read as zeros.
         */
        if (position > f_size) clear_zone(rip, f_size, 0);
    }

    /* Pipes are a little different. Check. */
    if (rip->i_pipe == I_PIPE) {
        r = pipe_check(rip, rw_flag, oflags,

```

```

        m_in.nbytes, position, &partial_cnt, 0);
    if (r <= 0) return(r);
}

if (partial_cnt > 0) partial_pipe = 1;

/* Split the transfer into chunks that don't span two blocks. */
while (m_in.nbytes != 0) {

    off = (unsigned int) (position % block_size); /* offset in blk*/
    if (partial_pipe) { /* pipes only */
        chunk = MIN(partial_cnt, block_size - off);
    } else
        chunk = MIN(m_in.nbytes, block_size - off);
    if (chunk < 0) chunk = block_size - off;

    if (rw_flag == READING) {
        bytes_left = f_size - position;
        if (position >= f_size) break; /* we are beyond EOF */
        if (chunk > bytes_left) chunk = (int) bytes_left;
    }

    /* Read or write 'chunk' bytes. */
    r = rw_chunk(rip, position, off, chunk, (unsigned) m_in.nbytes,
        rw_flag, m_in.buffer, seg, usr, block_size, &completed);

    if (r != OK) break; /* EOF reached */
    if (rdwt_err < 0) break;

    /* Update counters and pointers. */
    m_in.buffer += chunk; /* user buffer address */
    m_in.nbytes -= chunk; /* bytes yet to be read */
    cum_io += chunk; /* bytes read so far */
    position += chunk; /* position within the file */

    if (partial_pipe) {
        partial_cnt -= chunk;
        if (partial_cnt <= 0) break;
    }
}

/* On write, update file size and access time. */
if (rw_flag == WRITING) {
    if (regular || mode_word == I_DIRECTORY) {
        if (position > f_size) rip->i_size = position;
    }
} else {
    if (rip->i_pipe == I_PIPE) {
        if (position >= rip->i_size) {
            /* Reset pipe pointers. */
            rip->i_size = 0; /* no data left */
            position = 0; /* reset reader(s) */
            wf = find_filp(rip, W_BIT);
            if (wf != NIL_FILP) wf->filp_pos = 0;
        }
    }
}
f->filp_pos = position;

/* Check to see if read-ahead is called for, and if so, set it up. */
if (rw_flag == READING && rip->i_seek == NO_SEEK && position % block_size == 0
    && (regular || mode_word == I_DIRECTORY)) {
    rdahed_inode = rip;
    rdahedpos = position;
}
rip->i_seek = NO_SEEK;

if (rdwt_err != OK) r = rdwt_err; /* check for disk error */
if (rdwt_err == END_OF_FILE) r = OK;

/* if user-space copying failed, read/write failed. */
if (r == OK && r2 != OK) {
    r = r2;
}

```

```

}
if (r == OK) {
    if (rw_flag == READING) rip->i_update |= ATIME;
    if (rw_flag == WRITING) rip->i_update |= CTIME | MTIME;
    rip->i_dirt = DIRTY; /* inode is thus now dirty */
    if (partial_pipe) {
        partial_pipe = 0;
        /* partial write on pipe with */
        /* O_NONBLOCK, return write count */
        if (!(oflags & O_NONBLOCK)) {
            fp->fp_cum_io_partial = cum_io;
            suspend(XPIPE); /* partial write on pipe with */
            return(SUSPEND); /* nbytes > PIPE_SIZE - non-atomic */
        }
    }
    fp->fp_cum_io_partial = 0;
    return(cum_io);
}
return(r);
}

/*=====
*                               rw_chunk                               *
*=====*/
PRIVATE int rw_chunk(rip, position, off, chunk, left, rw_flag, buff,
    seg, usr, block_size, completed)
register struct inode *rip; /* pointer to inode for file to be rd/wr */
off_t position; /* position within file to read or write */
unsigned off; /* off within the current block */
int chunk; /* number of bytes to read or write */
unsigned left; /* max number of bytes wanted after position */
int rw_flag; /* READING or WRITING */
char *buff; /* virtual address of the user buffer */
int seg; /* T or D segment in user space */
int usr; /* which user process */
int block_size; /* block size of FS operating on */
int *completed; /* number of bytes copied */
{
    /* Read or write (part of) a block. */

    register struct buf *bp;
    register int r = OK;
    int n, block_spec;
    block_t b;
    dev_t dev;

    *completed = 0;

    block_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
    if (block_spec) {
        b = position/block_size;
        dev = (dev_t) rip->i_zone[0];
    } else {
        b = read_map(rip, position);
        dev = rip->i_dev;
    }

    if (!block_spec && b == NO_BLOCK) {
        if (rw_flag == READING) {
            /* Reading from a nonexistent block. Must read as all zeros.*/
            bp = get_block(NO_DEV, NO_BLOCK, NORMAL); /* get a buffer */
            zero_block(bp);
        } else {
            /* Writing to a nonexistent block. Create and enter in inode.*/
            if ((bp = new_block(rip, position)) == NIL_BUF) return(err_code);
        }
    } else if (rw_flag == READING) {
        /* Read and read ahead if convenient. */
        bp = rahead(rip, b, position, left);
    } else {
        /* Normally an existing block to be partially overwritten is first read
        * in. However, a full block need not be read in. If it is already in
        * the cache, acquire it, otherwise just acquire a free buffer.
        */
    }
}

```

```

    n = (chunk == block_size ? NO_READ : NORMAL);
    if (!block_spec && off == 0 && position >= rip->i_size) n = NO_READ;
    bp = get_block(dev, b, n);
}

/* In all cases, bp now points to a valid buffer. */
if (bp == NIL_BUF) {
    panic(__FILE__, "bp not valid in rw_chunk, this can't happen", NO_NUM);
}
if (rw_flag == WRITING && chunk != block_size && !block_spec &&
    position >= rip->i_size && off == 0) {
    zero_block(bp);
}

if (rw_flag == READING) {
    /* Copy a chunk from the block buffer to user space. */
    r = sys_vircopy(FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
        usr, seg, (phys_bytes) buff,
        (phys_bytes) chunk);
} else {
    /* Copy a chunk from user space to the block buffer. */
    r = sys_vircopy(usr, seg, (phys_bytes) buff,
        FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
        (phys_bytes) chunk);
    bp->b_dirt = DIRTY;
}
n = (off + chunk == block_size ? FULL_DATA_BLOCK : PARTIAL_DATA_BLOCK);
put_block(bp, n);

return(r);
}

/*=====
*                               read_map                               *
*=====*/
PUBLIC block_t read_map(rip, position)
register struct inode *rip;      /* ptr to inode to map from */
off_t position;                 /* position in file whose blk wanted */
{
    /* Given an inode and a position within the corresponding file, locate the
    * block (not zone) number in which that position is to be found and return it.
    */

    register struct buf *bp;
    register zone_t z;
    int scale, boff, dzones, nr_indirects, index, zind, ex;
    block_t b;
    long excess, zone, block_pos;

    scale = rip->i_sp->s_log_zone_size; /* for block-zone conversion */
    block_pos = position/rip->i_sp->s_block_size; /* relative blk # in file */
    zone = block_pos >> scale; /* position's zone */
    boff = (int) (block_pos - (zone << scale)); /* relative blk # within zone */
    dzones = rip->i_ndzones;
    nr_indirects = rip->i_nindirs;

    /* Is 'position' to be found in the inode itself? */
    if (zone < dzones) {
        zind = (int) zone; /* index should be an int */
        z = rip->i_zone[zind];
        if (z == NO_ZONE) return(NO_BLOCK);
        b = ((block_t) z << scale) + boff;
        return(b);
    }

    /* It is not in the inode, so it must be single or double indirect. */
    excess = zone - dzones; /* first Vx_NR_DZONES don't count */

    if (excess < nr_indirects) {
        /* 'position' can be located via the single indirect block. */
        z = rip->i_zone[dzones];
    } else {
        /* 'position' can be located via the double indirect block. */

```

```

    if ( (z = rip->i_zone[dzones+1]) == NO_ZONE) return(NO_BLOCK);
    excess -= nr_indirects; /* single indir doesn't count */
    b = (block_t) z << scale;
    bp = get_block(rip->i_dev, b, NORMAL); /* get double indirect block */
    index = (int) (excess/nr_indirects);
    z = rd_indir(bp, index); /* z= zone for single */
    put_block(bp, INDIRECT_BLOCK); /* release double ind block */
    excess = excess % nr_indirects; /* index into single ind blk */
}

/* 'z' is zone num for single indirect block; 'excess' is index into it. */
if (z == NO_ZONE) return(NO_BLOCK);
b = (block_t) z << scale; /* b is blk # for single ind */
bp = get_block(rip->i_dev, b, NORMAL); /* get single indirect block */
ex = (int) excess; /* need an integer */
z = rd_indir(bp, ex); /* get block pointed to */
put_block(bp, INDIRECT_BLOCK); /* release single indir blk */
if (z == NO_ZONE) return(NO_BLOCK);
b = ((block_t) z << scale) + boff;
return(b);
}

/*=====
*                               rd_indir                               *
*=====*/
PUBLIC zone_t rd_indir(bp, index)
struct buf *bp; /* pointer to indirect block */
int index; /* index into *bp */
{
/* Given a pointer to an indirect block, read one entry. The reason for
* making a separate routine out of this is that there are four cases:
* V1 (IBM and 68000), and V2 (IBM and 68000).
*/

    struct super_block *sp;
    zone_t zone; /* V2 zones are longs (shorts in V1) */

    if(bp == NIL_BUF)
        panic(__FILE__, "rd_indir() on NIL_BUF", NO_NUM);

    sp = get_super(bp->b_dev); /* need super block to find file sys type */

    /* read a zone from an indirect block */
    if (sp->s_version == V1)
        zone = (zone_t) conv2(sp->s_native, (int) bp->b_v1_ind[index]);
    else
        zone = (zone_t) conv4(sp->s_native, (long) bp->b_v2_ind[index]);

    if (zone != NO_ZONE &&
        (zone < (zone_t) sp->s_firstdatazone || zone >= sp->s_zones)) {
        printf("Illegal zone number %ld in indirect block, index %d\n",
            (long) zone, index);
        panic(__FILE__, "check file system", NO_NUM);
    }
    return(zone);
}

/*=====
*                               read_ahead                               *
*=====*/
PUBLIC void read_ahead()
{
/* Read a block into the cache before it is needed. */
    int block_size;
    register struct inode *rip;
    struct buf *bp;
    block_t b;

    rip = rdahed_inode; /* pointer to inode to read ahead from */
    block_size = get_block_size(rip->i_dev);
    rdahed_inode = NIL_INODE; /* turn off read ahead */
    if ( (b = read_map(rip, rdahedpos)) == NO_BLOCK) return; /* at EOF */
    bp = rahead(rip, b, rdahedpos, block_size);
    put_block(bp, PARTIAL_DATA_BLOCK);
}

```

```

}

/*=====
 *
 *                      rahead
 *=====*/
PUBLIC struct buf *rahead(rip, baseblock, position, bytes_ahead)
register struct inode *rip;      /* pointer to inode for file to be read */
block_t baseblock;              /* block at current position */
off_t position;                 /* position within file */
unsigned bytes_ahead;           /* bytes beyond position for immediate use */
{
/* Fetch a block from the cache or the device.  If a physical read is
 * required, prefetch as many more blocks as convenient into the cache.
 * This usually covers bytes_ahead and is at least BLOCKS_MINIMUM.
 * The device driver may decide it knows better and stop reading at a
 * cylinder boundary (or after an error).  Rw_scattered() puts an optional
 * flag on all reads to allow this.
 */
    int block_size;
/* Minimum number of blocks to prefetch. */
# define BLOCKS_MINIMUM      (NR_BUFS < 50 ? 18 : 32)
    int block_spec, scale, read_q_size;
    unsigned int blocks_ahead, fragment;
    block_t block, blocks_left;
    off_t indl_pos;
    dev_t dev;
    struct buf *bp;
    static struct buf *read_q[NR_BUFS];

    block_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
    if (block_spec) {
        dev = (dev_t) rip->i_zone[0];
    } else {
        dev = rip->i_dev;
    }
    block_size = get_block_size(dev);

    block = baseblock;
    bp = get_block(dev, block, PREFETCH);
    if (bp->b_dev != NO_DEV) return(bp);

/* The best guess for the number of blocks to prefetch:  A lot.
 * It is impossible to tell what the device looks like, so we don't even
 * try to guess the geometry, but leave it to the driver.
 *
 * The floppy driver can read a full track with no rotational delay, and it
 * avoids reading partial tracks if it can, so handing it enough buffers to
 * read two tracks is perfect.  (Two, because some diskette types have
 * an odd number of sectors per track, so a block may span tracks.)
 *
 * The disk drivers don't try to be smart.  With todays disks it is
 * impossible to tell what the real geometry looks like, so it is best to
 * read as much as you can.  With luck the caching on the drive allows
 * for a little time to start the next read.
 *
 * The current solution below is a bit of a hack, it just reads blocks from
 * the current file position hoping that more of the file can be found.  A
 * better solution must look at the already available zone pointers and
 * indirect blocks (but don't call read_map!).
 */

    fragment = position % block_size;
    position -= fragment;
    bytes_ahead += fragment;

    blocks_ahead = (bytes_ahead + block_size - 1) / block_size;

    if (block_spec && rip->i_size == 0) {
        blocks_left = NR_IOREQS;
    } else {
        blocks_left = (rip->i_size - position + block_size - 1) / block_size;

        /* Go for the first indirect block if we are in its neighborhood. */
        if (!block_spec) {

```

```
        scale = rip->i_sp->s_log_zone_size;
        indl_pos = (off_t) rip->i_ndzones * (block_size << scale);
        if (position <= indl_pos && rip->i_size > indl_pos) {
            blocks_ahead++;
            blocks_left++;
        }
    }
}

/* No more than the maximum request. */
if (blocks_ahead > NR_IOREQS) blocks_ahead = NR_IOREQS;

/* Read at least the minimum number of blocks, but not after a seek. */
if (blocks_ahead < BLOCKS_MINIMUM && rip->i_seek == NO_SEEK)
    blocks_ahead = BLOCKS_MINIMUM;

/* Can't go past end of file. */
if (blocks_ahead > blocks_left) blocks_ahead = blocks_left;

read_q_size = 0;

/* Acquire block buffers. */
for (;;) {
    read_q[read_q_size++] = bp;

    if (--blocks_ahead == 0) break;

    /* Don't trash the cache, leave 4 free. */
    if (bufs_in_use >= NR_BUFS - 4) break;

    block++;

    bp = get_block(dev, block, PREFETCH);
    if (bp->b_dev != NO_DEV) {
        /* Oops, block already in the cache, get out. */
        put_block(bp, FULL_DATA_BLOCK);
        break;
    }
}
rw_scattered(dev, read_q, read_q_size, READING);
return(get_block(dev, baseblock, NORMAL));
}
```

```

/* Implement entry point to select system call.
 *
 * The entry points into this file are
 *   do_select:      perform the SELECT system call
 *   select_callback: notify select system of possible fd operation
 *   select_notified: low-level entry for device notifying select
 *   select_unsuspend_by_endpt: cancel a blocking select on exiting driver
 *
 * Changes:
 *   6 june 2005   Created (Ben Gras)
 */

#define DEBUG_SELECT 0

#include "fs.h"
#include "select.h"
#include "file.h"
#include "inode.h"

#include <sys/time.h>
#include <sys/select.h>
#include <minix/com.h>
#include <string.h>

/* max. number of simultaneously pending select() calls */
#define MAXSELECTS 25

PRIVATE struct selectentry {
    struct fproc *requestor;          /* slot is free iff this is NULL */
    int req_endpt;
    fd_set readfds, writefds, errorfds;
    fd_set ready_readfds, ready_writefds, ready_errorfds;
    fd_set *vir_readfds, *vir_writefds, *vir_errorfds;
    struct filp *filps[FD_SETSIZE];
    int type[FD_SETSIZE];
    int nfds, nreadyfds;
    clock_t expiry;
    timer_t timer; /* if expiry > 0 */
} selecttab[MAXSELECTS];

#define SELFD_FILE      0
#define SELFD_PIPE      1
#define SELFD_TTY       2
#define SELFD_INET      3
#define SELFD_LOG       4
#define SEL_FDS         5

FORWARD _PROTOTYPE(int select_reevaluate, (struct filp *fp));

FORWARD _PROTOTYPE(int select_request_file,
    (struct filp *f, int *ops, int block));
FORWARD _PROTOTYPE(int select_match_file, (struct filp *f));

FORWARD _PROTOTYPE(int select_request_general,
    (struct filp *f, int *ops, int block));
FORWARD _PROTOTYPE(int select_major_match,
    (int match_major, struct filp *file));

FORWARD _PROTOTYPE(void select_cancel_all, (struct selectentry *e));
FORWARD _PROTOTYPE(void select_wakeup, (struct selectentry *e, int r));
FORWARD _PROTOTYPE(void select_return, (struct selectentry *, int));

/* The Open Group:
 * "The pselect() and select() functions shall support
 * regular files, terminal and pseudo-terminal devices,
 * STREAMS-based files, FIFOs, pipes, and sockets."
 */

PRIVATE struct fdtype {
    int (*select_request)(struct filp *, int *ops, int block);
    int (*select_match)(struct filp *);
    int select_major;
} fdtypes[SEL_FDS] = {
    /* SELFD_FILE */

```



```

    { select_request_file, select_match_file, 0 },
        /* SELFD_TTY (also PTY) */
    { select_request_general, NULL, TTY_MAJOR },
        /* SELFD_INET */
    { select_request_general, NULL, INET_MAJOR },
        /* SELFD_PIPE (pipe(2) pipes and FS FIFOs) */
    { select_request_pipe, select_match_pipe, 0 },
        /* SELFD_LOG (/dev/klog) */
    { select_request_general, NULL, LOG_MAJOR },
};

/* Open Group:
 * "File descriptors associated with regular files shall always select true
 * for ready to read, ready to write, and error conditions."
 */

/*=====
 *
 *                      select_request_file
 *=====*/
PRIVATE int select_request_file(struct filp *f, int *ops, int block)
{
    /* output *ops is input *ops */
    return SEL_OK;
}

/*=====
 *
 *                      select_match_file
 *=====*/
PRIVATE int select_match_file(struct filp *file)
{
    if (file && file->filp_ino && (file->filp_ino->i_mode & I_REGULAR))
        return 1;
    return 0;
}

/*=====
 *
 *                      select_request_general
 *=====*/
PRIVATE int select_request_general(struct filp *f, int *ops, int block)
{
    int rops = *ops;
    if (block) rops |= SEL_NOTIFY;
    *ops = dev_io(DEV_SELECT, f->filp_ino->i_zone[0], rops, NULL, 0, 0, 0);
    if (*ops < 0)
        return SEL_ERR;
    return SEL_OK;
}

/*=====
 *
 *                      select_major_match
 *=====*/
PRIVATE int select_major_match(int match_major, struct filp *file)
{
    int major;
    if (!(file && file->filp_ino &&
        (file->filp_ino->i_mode & I_TYPE) == I_CHAR_SPECIAL))
        return 0;
    major = (file->filp_ino->i_zone[0] >> MAJOR) & BYTE;
    if (major == match_major)
        return 1;
    return 0;
}

/*=====
 *
 *                      tab2ops
 *=====*/
PRIVATE int tab2ops(int fd, struct selectentry *e)
{
    return (FD_ISSET(fd, &e->readfds) ? SEL_RD : 0) |
        (FD_ISSET(fd, &e->writefds) ? SEL_WR : 0) |
        (FD_ISSET(fd, &e->errorfds) ? SEL_ERR : 0);
}

/*=====

```

```

*                                     ops2tab                                     *
*=====*/
PRIVATE void ops2tab(int ops, int fd, struct selectentry *e)
{
    if ((ops & SEL_RD) && e->vir_readfds && FD_ISSET(fd, &e->readfds)
        && !FD_ISSET(fd, &e->ready_readfds)) {
        FD_SET(fd, &e->ready_readfds);
        e->nreadyfds++;
    }
    if ((ops & SEL_WR) && e->vir_writefds && FD_ISSET(fd, &e->writefds)
        && !FD_ISSET(fd, &e->ready_writefds)) {
        FD_SET(fd, &e->ready_writefds);
        e->nreadyfds++;
    }
    if ((ops & SEL_ERR) && e->vir_errorfds && FD_ISSET(fd, &e->errorfds)
        && !FD_ISSET(fd, &e->ready_errorfds)) {
        FD_SET(fd, &e->ready_errorfds);
        e->nreadyfds++;
    }
}

return;
}

/*=====*
*                                     copy_fdsets                               *
*=====*/
PRIVATE void copy_fdsets(struct selectentry *e)
{
    if (e->vir_readfds)
        sys_vircopy(SELF, D, (vir_bytes) &e->ready_readfds,
            e->req_endpt, D, (vir_bytes) e->vir_readfds, sizeof(fd_set));
    if (e->vir_writefds)
        sys_vircopy(SELF, D, (vir_bytes) &e->ready_writefds,
            e->req_endpt, D, (vir_bytes) e->vir_writefds, sizeof(fd_set));
    if (e->vir_errorfds)
        sys_vircopy(SELF, D, (vir_bytes) &e->ready_errorfds,
            e->req_endpt, D, (vir_bytes) e->vir_errorfds, sizeof(fd_set));

    return;
}

/*=====*
*                                     do_select                                 *
*=====*/
PUBLIC int do_select(void)
{
    int r, nfds, is_timeout = 1, nonzero_timeout = 0,
        fd, s, block = 0;
    struct timeval timeout;
    nfds = m_in.SEL_NFDS;

    if (nfds < 0 || nfds > FD_SETSIZE)
        return EINVAL;

    for(s = 0; s < MAXSELECTS; s++)
        if (!selecttab[s].requestor)
            break;

    if (s >= MAXSELECTS)
        return ENOSPC;

    selecttab[s].req_endpt = who_e;
    selecttab[s].nfds = 0;
    selecttab[s].nreadyfds = 0;
    memset(selecttab[s].filps, 0, sizeof(selecttab[s].filps));

    /* defaults */
    FD_ZERO(&selecttab[s].readfds);
    FD_ZERO(&selecttab[s].writefds);
    FD_ZERO(&selecttab[s].errorfds);
    FD_ZERO(&selecttab[s].ready_readfds);
    FD_ZERO(&selecttab[s].ready_writefds);
    FD_ZERO(&selecttab[s].ready_errorfds);

```

```

selecttab[s].vir_readfds = (fd_set *) m_in.SEL_READFDS;
selecttab[s].vir_writefds = (fd_set *) m_in.SEL_WRITEFDS;
selecttab[s].vir_errorfds = (fd_set *) m_in.SEL_ERRORFDS;

/* copy args */
if (selecttab[s].vir_readfds
    && (r=sys_vircopy(who_e, D, (vir_bytes) m_in.SEL_READFDS,
        SELF, D, (vir_bytes) &selecttab[s].readfds, sizeof(fd_set))) != OK)
    return r;

if (selecttab[s].vir_writefds
    && (r=sys_vircopy(who_e, D, (vir_bytes) m_in.SEL_WRITEFDS,
        SELF, D, (vir_bytes) &selecttab[s].writefds, sizeof(fd_set))) != OK)
    return r;

if (selecttab[s].vir_errorfds
    && (r=sys_vircopy(who_e, D, (vir_bytes) m_in.SEL_ERRORFDS,
        SELF, D, (vir_bytes) &selecttab[s].errorfds, sizeof(fd_set))) != OK)
    return r;

if (!m_in.SEL_TIMEOUT)
    is_timeout = nonzero_timeout = 0;
else
    if ((r=sys_vircopy(who_e, D, (vir_bytes) m_in.SEL_TIMEOUT,
        SELF, D, (vir_bytes) &timeout, sizeof(timeout))) != OK)
        return r;

/* No nonsense in the timeval please. */
if (is_timeout && (timeout.tv_sec < 0 || timeout.tv_usec < 0))
    return EINVAL;

/* if is_timeout if 0, we block forever. otherwise, if nonzero_timeout
 * is 0, we do a poll (don't block). otherwise, we block up to the
 * specified time interval.
 */
if (is_timeout && (timeout.tv_sec > 0 || timeout.tv_usec > 0))
    nonzero_timeout = 1;

if (nonzero_timeout || !is_timeout)
    block = 1;
else
    block = 0; /* timeout set as (0,0) - this effects a poll */

/* no timeout set (yet) */
selecttab[s].expiry = 0;

for(fd = 0; fd < nfds; fd++) {
    int orig_ops, ops, t, type = -1, r;
    struct filp *filp;

    if (!(orig_ops = ops = tab2ops(fd, &selecttab[s])))
        continue;
    if (!(filp = selecttab[s].filps[fd] = get_filp(fd))) {
        select_cancel_all(&selecttab[s]);
        return EBADF;
    }

    for(t = 0; t < SEL_FDS; t++) {
        if (fdtypes[t].select_match) {
            if (fdtypes[t].select_match(filp)) {
                printf("select: fd %d is type %d ", fd, t);

                if (type != -1)
                    printf("select: double match\n");
                type = t;
            }
            else if (select_major_match(fdtypes[t].select_major, filp)) {
                type = t;
            }
        }
    }

    /* Open Group:
     * "The pselect() and select() functions shall support

```

```

        * regular files, terminal and pseudo-terminal devices,
        * STREAMS-based files, FIFOs, pipes, and sockets. The
        * behavior of pselect() and select() on file descriptors
        * that refer to other types of file is unspecified."
        *
        * If all types are implemented, then this is another
        * type of file and we get to do whatever we want.
        */
        if (type == -1)
        {
#if DEBUG_SELECT
                printf("do_select: bad type\n");
#endif
                return EBADF;
        }

        selecttab[s].type[fd] = type;

        if ((selecttab[s].filps[fd]->filp_select_ops & ops) != ops) {
                int wantops;
                /* Request the select on this fd.  */
#if DEBUG_SELECT
                printf("%p requesting ops %d -> ",
                        selecttab[s].filps[fd],
                        selecttab[s].filps[fd]->filp_select_ops);
#endif
                wantops = (selecttab[s].filps[fd]->filp_select_ops | ops);
#if DEBUG_SELECT
                printf("%d\n", selecttab[s].filps[fd]->filp_select_ops);
#endif
                if ((r = fdtypes[type].select_request(filp,
                        &wantops, block)) != SEL_OK) {
                        /* error or bogus return code.. backpaddle */
                        select_cancel_all(&selecttab[s]);
                        printf("select: select_request returned error\n");
                        return EINVAL;
                }
                if (wantops) {
                        if (wantops & ops) {
                                /* operations that were just requested
                                 * are ready to go right away
                                 */
                                ops2tab(wantops, fd, &selecttab[s]);
                        }
                        /* if there are any other select()s blocking
                         * on these operations of this fp, they can
                         * be awoken too
                         */
                        select_callback(filp, ops);
                }
#if DEBUG_SELECT
                printf("select request ok; ops returned %d\n", wantops);
#endif
#endif
        } else {
#if DEBUG_SELECT
                printf("select already happening on that filp\n");
#endif
#endif
        }

        selecttab[s].nfdns = fd+1;
        selecttab[s].filps[fd]->filp_selectors++;

#if DEBUG_SELECT
        printf("[fd %d ops: %d] ", fd, ops);
#endif
}

if (selecttab[s].nreadyfds > 0 || !block) {
        /* fd's were found that were ready to go right away, and/or
         * we were instructed not to block at all. Must return
         * immediately.
         */
        copy_fdsets(&selecttab[s]);
        select_cancel_all(&selecttab[s]);
}

```

```

        selecttab[s].requestor = NULL;

        /* Open Group:
         * "Upon successful completion, the pselect() and select()
         * functions shall return the total number of bits
         * set in the bit masks."
         */
    #if DEBUG_SELECT
        printf("returning\n");
    #endif

        return selecttab[s].nreadyfds;
    }
    #if DEBUG_SELECT
        printf("not returning (%d, %d)\n", selecttab[s].nreadyfds, block);
    #endif

    /* Convert timeval to ticks and set the timer. If it fails, undo
     * all, return error.
     */
    if (is_timeout) {
        int ticks;
        /* Open Group:
         * "If the requested timeout interval requires a finer
         * granularity than the implementation supports, the
         * actual timeout interval shall be rounded up to the next
         * supported value."
         */
    #define USECPERSEC 1000000
        while(timeout.tv_usec >= USECPERSEC) {
            /* this is to avoid overflow with *HZ below */
            timeout.tv_usec -= USECPERSEC;
            timeout.tv_sec++;
        }
        ticks = timeout.tv_sec * HZ +
            (timeout.tv_usec * HZ + USECPERSEC-1) / USECPERSEC;
        selecttab[s].expiry = ticks;
        fs_set_timer(&selecttab[s].timer, ticks, select_timeout_check, s);
    #if DEBUG_SELECT
        printf("%d: blocking %d ticks\n", s, ticks);
    #endif
    }

    /* if we're blocking, the table entry is now valid. */
    selecttab[s].requestor = fp;

    /* process now blocked */
    suspend(XSELECT);
    return SUSPEND;
}

/*=====
 *                               select_cancel_all                               *
 *=====*/
PRIVATE void select_cancel_all(struct selectentry *e)
{
    int fd;

    for(fd = 0; fd < e->nfds; fd++) {
        struct filp *fp;
        fp = e->filps[fd];
        if (!fp) {
    #if DEBUG_SELECT
            printf("[ fd %d/%d NULL ] ", fd, e->nfds);
    #endif
            continue;
        }
        if (fp->filp_selectors < 1) {
    #if DEBUG_SELECT
            printf("select: %d selectors?!\n", fp->filp_selectors);
    #endif
            continue;
        }
        fp->filp_selectors--;
    }
}

```

```

        e->filps[fd] = NULL;
        select_reevaluate(fp);
    }

    if (e->expiry > 0) {
#ifdef DEBUG_SELECT
        printf("cancelling timer %d\n", e - selecttab);
#endif
        fs_cancel_timer(&e->timer);
        e->expiry = 0;
    }

    return;
}

/*=====
 *                               select_wakeup                               *
 *=====*/
PRIVATE void select_wakeup(struct selectentry *e, int r)
{
    revive(e->req_endpt, r);
}

/*=====
 *                               select_reevaluate                           *
 *=====*/
PRIVATE int select_reevaluate(struct filp *fp)
{
    int s, remain_ops = 0, fd, type = -1;

    if (!fp) {
        printf("fs: select: reevalute NULL fp\n");
        return 0;
    }

    for(s = 0; s < MAXSELECTS; s++) {
        if (!selecttab[s].requestor)
            continue;
        for(fd = 0; fd < selecttab[s].nfd; fd++)
            if (fp == selecttab[s].filps[fd]) {
                remain_ops |= tab2ops(fd, &selecttab[s]);
                type = selecttab[s].type[fd];
            }
    }

    /* If there are any select()s open that want any operations on
     * this fd that haven't been satisfied by this callback, then we're
     * still in the market for it.
     */
    fp->filp_select_ops = remain_ops;
#ifdef DEBUG_SELECT
    printf("remaining operations on fp are %d\n", fp->filp_select_ops);
#endif

    return remain_ops;
}

/*=====
 *                               select_return                               *
 *=====*/
PRIVATE void select_return(struct selectentry *s, int r)
{
    select_cancel_all(s);
    copy_fdsets(s);
    select_wakeup(s, r ? r : s->nreadyfds);
    s->requestor = NULL;
}

/*=====
 *                               select_callback                             *
 *=====*/
PUBLIC int select_callback(struct filp *fp, int ops)
{
    int s, fd, want_ops, type;

```

```

/* We are being notified that file pointer fp is available for
 * operations 'ops'. We must re-register the select for
 * operations that we are still interested in, if any.
 */

want_ops = 0;
type = -1;
for(s = 0; s < MAXSELECTS; s++) {
    int wakehim = 0;
    if (!selecttab[s].requestor)
        continue;
    for(fd = 0; fd < selecttab[s].nfd; fd++) {
        if (!selecttab[s].filps[fd])
            continue;
        if (selecttab[s].filps[fd] == fp) {
            int this_want_ops;
            this_want_ops = tab2ops(fd, &selecttab[s]);
            want_ops |= this_want_ops;
            if (this_want_ops & ops) {
                /* this select() has been satisfied. */
                ops2tab(ops, fd, &selecttab[s]);
                wakehim = 1;
            }
            type = selecttab[s].type[fd];
        }
    }
    if (wakehim)
        select_return(&selecttab[s], 0);
}

return 0;
}

/*=====
 *                               select_notified                               *
 *=====*/
PUBLIC int select_notified(int major, int minor, int selected_ops)
{
    int s, f, t;

#ifdef DEBUG_SELECT
    printf("select callback: %d, %d: %d\n", major, minor, selected_ops);
#endif

    for(t = 0; t < SEL_FDS; t++)
        if (!fdtypes[t].select_match && fdtypes[t].select_major == major)
            break;

    if (t >= SEL_FDS) {
#ifdef DEBUG_SELECT
        printf("select callback: no fdtype found for device %d\n", major);
#endif
        return OK;
    }

    /* We have a select callback from major device no.
     * d, which corresponds to our select type t.
     */

    for(s = 0; s < MAXSELECTS; s++) {
        int s_minor, ops;
        if (!selecttab[s].requestor)
            continue;
        for(f = 0; f < selecttab[s].nfd; f++) {
            if (!selecttab[s].filps[f] ||
                !select_major_match(major, selecttab[s].filps[f]))
                continue;
            ops = tab2ops(f, &selecttab[s]);
            s_minor =
                (selecttab[s].filps[f]->filp_ino->i_zone[0] >> MINOR)
                & BYTE;
            if ((s_minor == minor) &&
                (selected_ops & ops)) {

```

```

select_callback(selecttab[s].filps[f], (selected_ops & op
s));
    }
}

return OK;
}

/*=====
 *                               init_select                               *
 *=====*/
PUBLIC void init_select(void)
{
    int s;

    for(s = 0; s < MAXSELECTS; s++)
        fs_init_timer(&selecttab[s].timer);
}

/*=====
 *                               select_forget                               *
 *=====*/
PUBLIC void select_forget(int proc_e)
{
    /* something has happened (e.g. signal delivered that interrupts
     * select()). totally forget about the select().
     */
    int s;

    for(s = 0; s < MAXSELECTS; s++) {
        if (selecttab[s].requestor &&
            selecttab[s].req_endpt == proc_e) {
            break;
        }
    }

    if (s >= MAXSELECTS) {
#ifdef DEBUG_SELECT
        printf("select: cancelled select() not found");
#endif
        return;
    }

    select_cancel_all(&selecttab[s]);
    selecttab[s].requestor = NULL;

    return;
}

/*=====
 *                               select_timeout_check                               *
 *=====*/
PUBLIC void select_timeout_check(timer_t *timer)
{
    int s;

    s = tmr_arg(timer)->ta_int;

    if (s < 0 || s >= MAXSELECTS) {
#ifdef DEBUG_SELECT
        printf("select: bogus slot arg to watchdog %d\n", s);
#endif
        return;
    }

    if (!selecttab[s].requestor) {
#ifdef DEBUG_SELECT
        printf("select: no requestor in watchdog\n");
#endif
        return;
    }
}

```



```
        if (selecttab[s].expiry <= 0) {
#if DEBUG_SELECT
            printf("select: strange expiry value in watchdog\n", s);
#endif
            return;
        }

        selecttab[s].expiry = 0;
        select_return(&selecttab[s], 0);

        return;
    }

/*=====
 *                               select_unsuspend_by_endpt                               *
 *=====*/
PUBLIC void select_unsuspend_by_endpt(int proc_e)
{
    int fd, s;

    for(s = 0; s < MAXSELECTS; s++) {
        if (!selecttab[s].requestor)
            continue;
        for(fd = 0; fd < selecttab[s].nfd; fd++) {
            int maj;
            if (!selecttab[s].filps[fd] || !selecttab[s].filps[fd]->filp_ino)
                continue;
            maj = (selecttab[s].filps[fd]->filp_ino->i_zone[0] >> MAJOR)&BYTE;
            if(dmap_driver_match(proc_e, maj)) {
                select_return(&selecttab[s], EAGAIN);
            }
        }
    }

    return;
}
```

```
#ifndef _FS_SELECT_H
#define _FS_SELECT_H 1

/* return codes for select_request_* and select_cancel_* */
#define SEL_OK      0      /* ready */
#define SEL_ERROR   1      /* failed */

#endif
```

```

/* This file contains the code for performing four system calls relating to
 * status and directories.
 *
 * The entry points into this file are
 * do_chdir: perform the CHDIR system call
 * do_chroot: perform the CHROOT system call
 * do_stat: perform the STAT system call
 * do_fstat: perform the FSTAT system call
 * do_fstatfs: perform the FSTATFS system call
 * do_lstat: perform the LSTAT system call
 * do_rdlink: perform the RDLNK system call
 */

#include "fs.h"
#include <sys/stat.h>
#include <sys/statfs.h>
#include <minix/com.h>
#include <string.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

FORWARD _PROTOTYPE( int change, (struct inode **iip, char *name_ptr, int len));
FORWARD _PROTOTYPE( int change_into, (struct inode **iip, struct inode *ip));
FORWARD _PROTOTYPE( int stat_inode, (struct inode *rip, struct filp *fil_ptr,
                                     char *user_addr) );

/*=====
 *
 * do_fchdir
 *=====*/
PUBLIC int do_fchdir()
{
    /* Change directory on already-opened fd. */
    struct filp *rfilp;

    /* Is the file descriptor valid? */
    if ( (rfilp = get_filp(m_in.fd)) == NIL_FILP) return(err_code);
    dup_inode(rfilp->filp_ino);
    return change_into(&fp->fp_workdir, rfilp->filp_ino);
}

/*=====
 *
 * do_chdir
 *=====*/
PUBLIC int do_chdir()
{
    /* Change directory. This function is also called by MM to simulate a chdir
     * in order to do EXEC, etc. It also changes the root directory, the uids and
     * gids, and the umask.
     */

    int r;
    register struct fproc *rfp;

    if (who_e == PM_PROC_NR) {
        int slot;
        if(isokendpt(m_in.endpt1, &slot) != OK)
            return EINVAL;
        rfp = &fproc[slot];
        put_inode(fp->fp_rootdir);
        dup_inode(fp->fp_rootdir = rfp->fp_rootdir);
        put_inode(fp->fp_workdir);
        dup_inode(fp->fp_workdir = rfp->fp_workdir);

        /* MM uses access() to check permissions. To make this work, pretend
         * that the user's real ids are the same as the user's effective ids.
         * FS calls other than access() do not use the real ids, so are not
         * affected.
         */
        fp->fp_realuid =
        fp->fp_effuid = rfp->fp_effuid;
    }
}

```

```

        fp->fp_realgid =
        fp->fp_effgid = rfp->fp_effgid;
        fp->fp_umask = rfp->fp_umask;
        return(OK);
    }

    /* Perform the chdir(name) system call. */
    r = change(&fp->fp_workdir, m_in.name, m_in.name_length);
    return(r);
}

/*=====
 *                               do_chroot                               *
 *=====*/
PUBLIC int do_chroot()
{
    /* Perform the chroot(name) system call. */

    register int r;

    if (!super_user) return(EPERM);          /* only su may chroot() */
    r = change(&fp->fp_rootdir, m_in.name, m_in.name_length);
    return(r);
}

/*=====
 *                               change                               *
 *=====*/
PRIVATE int change(iip, name_ptr, len)
struct inode **iip;          /* pointer to the inode pointer for the dir */
char *name_ptr;              /* pointer to the directory name to change to */
int len;                     /* length of the directory name string */
{
    /* Do the actual work for chdir() and chroot(). */
    struct inode *rip;

    /* Try to open the new directory. */
    if (fetch_name(name_ptr, len, M3) != OK) return(err_code);
    if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
    return change_into(iip, rip);
}

/*=====
 *                               change_into                               *
 *=====*/
PRIVATE int change_into(iip, rip)
struct inode **iip;          /* pointer to the inode pointer for the dir */
struct inode *rip;           /* this is what the inode has to become */
{
    register int r;

    /* It must be a directory and also be searchable. */
    if ( (rip->i_mode & I_TYPE) != I_DIRECTORY)
        r = ENOTDIR;
    else
        r = forbidden(rip, X_BIT);          /* check if dir is searchable */

    /* If error, return inode. */
    if (r != OK) {
        put_inode(rip);
        return(r);
    }

    /* Everything is OK. Make the change. */
    put_inode(*iip);          /* release the old directory */
    *iip = rip;               /* acquire the new one */
    return(OK);
}

/*=====
 *                               do_stat                               *
 *=====*/
PUBLIC int do_stat()
{

```

```

/* Perform the stat(name, buf) system call. */

register struct inode *rip;
register int r;

/* Both stat() and fstat() use the same routine to do the real work. That
 * routine expects an inode, so acquire it temporarily.
 */
if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
r = stat_inode(rip, NIL_FILP, m_in.name2); /* actually do the work.*/
put_inode(rip); /* release the inode */
return(r);
}

/*=====
 * do_fstat
 *=====*/
PUBLIC int do_fstat()
{
/* Perform the fstat(fd, buf) system call. */

register struct filp *rfilp;

/* Is the file descriptor valid? */
if ( (rfilp = get_filp(m_in.fd)) == NIL_FILP) return(err_code);

return(stat_inode(rfilp->filp_ino, rfilp, m_in.buffer));
}

/*=====
 * stat_inode
 *=====*/
PRIVATE int stat_inode(rip, fil_ptr, user_addr)
register struct inode *rip; /* pointer to inode to stat */
struct filp *fil_ptr; /* filp pointer, supplied by 'fstat' */
char *user_addr; /* user space address where stat buf goes */
{
/* Common code for stat and fstat system calls. */

struct stat statbuf;
mode_t mo;
int r, s;

/* Update the atime, ctime, and mtime fields in the inode, if need be. */
if (rip->i_update) update_times(rip);

/* Fill in the statbuf struct. */
mo = rip->i_mode & I_TYPE;

/* true iff special */
s = (mo == I_CHAR_SPECIAL || mo == I_BLOCK_SPECIAL);

statbuf.st_dev = rip->i_dev;
statbuf.st_ino = rip->i_num;
statbuf.st_mode = rip->i_mode;
statbuf.st_nlink = rip->i_nlinks;
statbuf.st_uid = rip->i_uid;
statbuf.st_gid = rip->i_gid;
statbuf.st_rdev = (dev_t) (s ? rip->i_zone[0] : NO_DEV);
statbuf.st_size = rip->i_size;

if (rip->i_pipe == I_PIPE) {
statbuf.st_mode &= ~I_REGULAR; /* wipe out I_REGULAR bit for pipes */
if (fil_ptr != NIL_FILP && fil_ptr->filp_mode & R_BIT)
statbuf.st_size -= fil_ptr->filp_pos;
}

statbuf.st_atime = rip->i_atime;
statbuf.st_mtime = rip->i_mtime;
statbuf.st_ctime = rip->i_ctime;

/* Copy the struct to user space. */
r = sys_datacopy(FS_PROC_NR, (vir_bytes) &statbuf,

```

```

        who_e, (vir_bytes) user_addr, (phys_bytes) sizeof(statbuf));
    }
    return(r);
}

/*=====
 *                               do_fstatfs                               *
 *=====*/
PUBLIC int do_fstatfs()
{
    /* Perform the fstatfs(fd, buf) system call. */
    struct statfs st;
    register struct filp *rfilp;
    int r;

    /* Is the file descriptor valid? */
    if ( (rfilp = get_filp(m_in.fd)) == NIL_FILP) return(err_code);

    st.f_bsize = rfilp->filp_ino->i_sp->s_block_size;

    r = sys_datacopy(FS_PROC_NR, (vir_bytes) &st,
        who_e, (vir_bytes) m_in.buffer, (phys_bytes) sizeof(st));

    return(r);
}

/*=====
 *                               do_lstat                               *
 *=====*/
PUBLIC int do_lstat()
{
    /* Perform the lstat(name, buf) system call. */

    register int r;          /* return value */
    register struct inode *rip; /* target inode */

    if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
    if ((rip = parse_path(user_path, (char *) 0, EAT_PATH_OPAQUE)) == NIL_INODE)
        return(err_code);
    r = stat_inode(rip, NIL_FILP, m_in.name2);
    put_inode(rip);
    return(r);
}

/*=====
 *                               do_rdlink                               *
 *=====*/
PUBLIC int do_rdlink()
{
    /* Perform the readlink(name, buf) system call. */

    register int r;          /* return value */
    block_t b;              /* block containing link text */
    struct buf *bp;          /* buffer containing link text */
    register struct inode *rip; /* target inode */
    int copylen;
    copylen = m_in.m1_i2;
    if(copylen < 0) return EINVAL;

    if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
    if ((rip = parse_path(user_path, (char *) 0, EAT_PATH_OPAQUE)) == NIL_INODE)
        return(err_code);

    r = EACCES;
    if (S_ISLNK(rip->i_mode) && (b = read_map(rip, (off_t) 0)) != NO_BLOCK) {
        if (m_in.name2_length <= 0) r = EINVAL;
        else if (m_in.name2_length < rip->i_size) r = ERANGE;
        else {
            if(rip->i_size < copylen) copylen = rip->i_size;
            bp = get_block(rip->i_dev, b, NORMAL);
            r = sys_vircopy(SELFS, D, (vir_bytes) bp->b_data,
                who_e, D, (vir_bytes) m_in.name2, (vir_bytes) copylen);

            if (r == OK) r = copylen;
            put_block(bp, DIRECTORY_BLOCK);
        }
    }
}

```

```
    }  
}  
  
put_inode(rip);  
return(r);  
}
```

```

/* This file manages the super block table and the related data structures,
 * namely, the bit maps that keep track of which zones and which inodes are
 * allocated and which are free. When a new inode or zone is needed, the
 * appropriate bit map is searched for a free entry.
 *
 * The entry points into this file are
 *   alloc_bit:      somebody wants to allocate a zone or inode; find one
 *   free_bit:       indicate that a zone or inode is available for allocation
 *   get_super:      search the 'superblock' table for a device
 *   mounted:        tells if file inode is on mounted (or ROOT) file system
 *   read_super:     read a superblock
 */

#include "fs.h"
#include <string.h>
#include <minix/com.h>
#include "buf.h"
#include "inode.h"
#include "super.h"
#include "const.h"

/*=====
 *                               alloc_bit                               *
 *=====*/
PUBLIC bit_t alloc_bit(sp, map, origin)
struct super_block *sp;      /* the filesystem to allocate from */
int map;                     /* IMAP (inode map) or ZMAP (zone map) */
bit_t origin;                /* number of bit to start searching at */
{
/* Allocate a bit from a bit map and return its bit number. */

    block_t start_block;      /* first bit block */
    bit_t map_bits;           /* how many bits are there in the bit map? */
    unsigned bit_blocks;      /* how many blocks are there in the bit map? */
    unsigned block, word, bcount;
    struct buf *bp;
    bitchunk_t *wptr, *wlim, k;
    bit_t i, b;

    if (sp->s_rd_only)
        panic(__FILE__, "can't allocate bit on read-only fileys.", NO_NUM);

    if (map == IMAP) {
        start_block = START_BLOCK;
        map_bits = sp->s_ninodes + 1;
        bit_blocks = sp->s_imap_blocks;
    } else {
        start_block = START_BLOCK + sp->s_imap_blocks;
        map_bits = sp->s_zones - (sp->s_firstdatazone - 1);
        bit_blocks = sp->s_zmap_blocks;
    }

    /* Figure out where to start the bit search (depends on 'origin'). */
    if (origin >= map_bits) origin = 0; /* for robustness */

    /* Locate the starting place. */
    block = origin / FS_BITS_PER_BLOCK(sp->s_block_size);
    word = (origin % FS_BITS_PER_BLOCK(sp->s_block_size)) / FS_BITCHUNK_BITS;

    /* Iterate over all blocks plus one, because we start in the middle. */
    bcount = bit_blocks + 1;
    do {
        bp = get_block(sp->s_dev, start_block + block, NORMAL);
        wlim = &bp->b_bitmap[FS_BITMAP_CHUNKS(sp->s_block_size)];

        /* Iterate over the words in block. */
        for (wptr = &bp->b_bitmap[word]; wptr < wlim; wptr++) {

            /* Does this word contain a free bit? */
            if (*wptr == (bitchunk_t) ~0) continue;

            /* Find and allocate the free bit. */
            k = conv2(sp->s_native, (int) *wptr);
            for (i = 0; (k & (1 << i)) != 0; ++i) {}

```



```

        /* Bit number from the start of the bit map. */
        b = ((bit_t) block * FS_BITS_PER_BLOCK(sp->s_block_size))
            + (wptr - &bp->b_bitmap[0]) * FS_BITCHUNK_BITS
            + i;

        /* Don't allocate bits beyond the end of the map. */
        if (b >= map_bits) break;

        /* Allocate and return bit number. */
        k |= 1 << i;
        *wptr = conv2(sp->s_native, (int) k);
        bp->b_dirt = DIRTY;
        put_block(bp, MAP_BLOCK);
        return(b);
    }
    put_block(bp, MAP_BLOCK);
    if (++block >= bit_blocks) block = 0; /* last block, wrap around */
    word = 0;
} while (--bcount > 0);
return(NO_BIT); /* no bit could be allocated */
}

/*=====
*                                     free_bit                                     *
*=====*/
PUBLIC void free_bit(sp, map, bit_returned)
struct super_block *sp; /* the filesystem to operate on */
int map; /* IMAP (inode map) or ZMAP (zone map) */
bit_t bit_returned; /* number of bit to insert into the map */
{
    /* Return a zone or inode by turning off its bitmap bit. */

    unsigned block, word, bit;
    struct buf *bp;
    bitchunk_t k, mask;
    block_t start_block;

    if (sp->s_rd_only)
        panic(__FILE__, "can't free bit on read-only filesystem", NO_NUM);

    if (map == IMAP) {
        start_block = START_BLOCK;
    } else {
        start_block = START_BLOCK + sp->s_imap_blocks;
    }
    block = bit_returned / FS_BITS_PER_BLOCK(sp->s_block_size);
    word = (bit_returned % FS_BITS_PER_BLOCK(sp->s_block_size))
        / FS_BITCHUNK_BITS;

    bit = bit_returned % FS_BITCHUNK_BITS;
    mask = 1 << bit;

    bp = get_block(sp->s_dev, start_block + block, NORMAL);

    k = conv2(sp->s_native, (int) bp->b_bitmap[word]);
    if (!(k & mask)) {
        panic(__FILE__, map == IMAP ? "tried to free unused inode" :
            "tried to free unused block", bit_returned);
    }

    k &= ~mask;
    bp->b_bitmap[word] = conv2(sp->s_native, (int) k);
    bp->b_dirt = DIRTY;

    put_block(bp, MAP_BLOCK);
}

/*=====
*                                     get_super                                     *
*=====*/
PUBLIC struct super_block *get_super(dev)
dev_t dev; /* device number whose super_block is sought */
{

```

```

/* Search the superblock table for this device.  It is supposed to be there. */

register struct super_block *sp;

if (dev == NO_DEV)
    panic(__FILE__, "request for super_block of NO_DEV", NO_NUM);

for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
    if (sp->s_dev == dev) return(sp);

/* Search failed.  Something wrong. */
panic(__FILE__, "can't find superblock for device (in decimal)", (int) dev);

return(NIL_SUPER);          /* to keep the compiler and lint quiet */
}

/*=====
*                               get_block_size                               *
*=====*/
PUBLIC int get_block_size(dev_t dev)
{
/* Search the superblock table for this device. */

register struct super_block *sp;

if (dev == NO_DEV)
    panic(__FILE__, "request for block size of NO_DEV", NO_NUM);

for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++) {
    if (sp->s_dev == dev) {
        return(sp->s_block_size);
    }
}

/* no mounted filesystem? use this block size then. */
return _MIN_BLOCK_SIZE;
}

/*=====
*                               mounted                               *
*=====*/
PUBLIC int mounted(rip)
register struct inode *rip;    /* pointer to inode */
{
/* Report on whether the given inode is on a mounted (or ROOT) file system. */

register struct super_block *sp;
register dev_t dev;

dev = (dev_t) rip->i_zone[0];
if (dev == root_dev) return(TRUE);    /* inode is on root file system */

for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
    if (sp->s_dev == dev) return(TRUE);

return(FALSE);
}

/*=====
*                               read_super                               *
*=====*/
PUBLIC int read_super(sp)
register struct super_block *sp; /* pointer to a superblock */
{
/* Read a superblock. */
dev_t dev;
int magic;
int version, native, r;
static char sbbuf[_MIN_BLOCK_SIZE];

dev = sp->s_dev;          /* save device (will be overwritten by copy) */
if (dev == NO_DEV)
    panic(__FILE__, "request for super_block of NO_DEV", NO_NUM);
r = dev_io(DEV_READ, dev, FS_PROC_NR,

```

```

    sbbuf, SUPER_BLOCK_BYTES, _MIN_BLOCK_SIZE, 0);
if (r != _MIN_BLOCK_SIZE) {
    return EINVAL;
}
memcpy(sp, sbbuf, sizeof(*sp));
sp->s_dev = NO_DEV;           /* restore later */
magic = sp->s_magic;          /* determines file system type */

/* Get file system version and type. */
if (magic == SUPER_MAGIC || magic == conv2(BYTE_SWAP, SUPER_MAGIC)) {
    version = V1;
    native = (magic == SUPER_MAGIC);
} else if (magic == SUPER_V2 || magic == conv2(BYTE_SWAP, SUPER_V2)) {
    version = V2;
    native = (magic == SUPER_V2);
} else if (magic == SUPER_V3) {
    version = V3;
    native = 1;
} else {
    return(EINVAL);
}

/* If the super block has the wrong byte order, swap the fields; the magic
 * number doesn't need conversion. */
sp->s_ninodes = conv4(native, sp->s_ninodes);
sp->s_nzones = conv2(native, (int) sp->s_nzones);
sp->s_imap_blocks = conv2(native, (int) sp->s_imap_blocks);
sp->s_zmap_blocks = conv2(native, (int) sp->s_zmap_blocks);
sp->s_firstdatazone = conv2(native, (int) sp->s_firstdatazone);
sp->s_log_zone_size = conv2(native, (int) sp->s_log_zone_size);
sp->s_max_size = conv4(native, sp->s_max_size);
sp->s_zones = conv4(native, sp->s_zones);

/* In V1, the device size was kept in a short, s_nzones, which limited
 * devices to 32K zones. For V2, it was decided to keep the size as a
 * long. However, just changing s_nzones to a long would not work, since
 * then the position of s_magic in the super block would not be the same
 * in V1 and V2 file systems, and there would be no way to tell whether
 * a newly mounted file system was V1 or V2. The solution was to introduce
 * a new variable, s_zones, and copy the size there.
 *
 * Calculate some other numbers that depend on the version here too, to
 * hide some of the differences.
 */
if (version == V1) {
    sp->s_block_size = _STATIC_BLOCK_SIZE;
    sp->s_zones = sp->s_nzones;           /* only V1 needs this copy */
    sp->s_inodes_per_block = V1_INODES_PER_BLOCK;
    sp->s_ndzones = V1_NR_DZONES;
    sp->s_nindirs = V1_INDIRECTS;
} else {
    if (version == V2)
        sp->s_block_size = _STATIC_BLOCK_SIZE;
    if (sp->s_block_size < _MIN_BLOCK_SIZE)
        return EINVAL;
    sp->s_inodes_per_block = V2_INODES_PER_BLOCK(sp->s_block_size);
    sp->s_ndzones = V2_NR_DZONES;
    sp->s_nindirs = V2_INDIRECTS(sp->s_block_size);
}

if (sp->s_block_size < _MIN_BLOCK_SIZE) {
    return EINVAL;
}
if (sp->s_block_size > _MAX_BLOCK_SIZE) {
    printf("Filesystem block size is %d kB; maximum filesystem\n"
           "block size is %d kB. This limit can be increased by recompiling.\n",
           sp->s_block_size/1024, _MAX_BLOCK_SIZE/1024);
    return EINVAL;
}
if ((sp->s_block_size % 512) != 0) {
    return EINVAL;
}
if (SUPER_SIZE > sp->s_block_size) {
    return EINVAL;
}

```

```
}
if ((sp->s_block_size % V2_INODE_SIZE) != 0 ||
     (sp->s_block_size % V1_INODE_SIZE) != 0) {
    return EINVAL;
}

sp->s_isearch = 0;           /* inode searches initially start at 0 */
sp->s_zsearch = 0;           /* zone searches initially start at 0 */
sp->s_version = version;
sp->s_native = native;

/* Make a few basic checks to see if super block looks reasonable. */
if (sp->s_imap_blocks < 1 || sp->s_zmap_blocks < 1
    || sp->s_ninodes < 1 || sp->s_zones < 1
    || (unsigned) sp->s_log_zone_size > 4) {
    printf("not enough imap or zone map blocks, \n");
    printf("or not enough inodes, or not enough zones, "
           "or zone size too large\n");
    return(EINVAL);
}
sp->s_dev = dev;             /* restore device number */
return(OK);
}
```

```

/* Super block table. The root file system and every mounted file system
 * has an entry here. The entry holds information about the sizes of the bit
 * maps and inodes. The s_ninodes field gives the number of inodes available
 * for files and directories, including the root directory. Inode 0 is
 * on the disk, but not used. Thus s_ninodes = 4 means that 5 bits will be
 * used in the bit map, bit 0, which is always 1 and not used, and bits 1-4
 * for files and directories. The disk layout is:
 *
 *      Item          # blocks
 *      boot block    1
 *      super block    1      (offset 1kB)
 *      inode map      s_imap_blocks
 *      zone map       s_zmap_blocks
 *      inodes         (s_ninodes + 'inodes per block' - 1) / 'inodes per block'
 *      unused         whatever is needed to fill out the current zone
 *      data zones     (s_zones - s_firstdatazone) << s_log_zone_size
 *
 * A super_block slot is free if s_dev == NO_DEV.
 */

EXTERN struct super_block {
    ino_t s_ninodes;          /* # usable inodes on the minor device */
    zonel_t s_nzones;        /* total device size, including bit maps etc */
    short s_imap_blocks;     /* # of blocks used by inode bit map */
    short s_zmap_blocks;     /* # of blocks used by zone bit map */
    zonel_t s_firstdatazone; /* number of first data zone */
    short s_log_zone_size;   /* log2 of blocks/zone */
    short s_pad;             /* try to avoid compiler-dependent padding */
    off_t s_max_size;        /* maximum file size on this device */
    zone_t s_zones;          /* number of zones (replaces s_nzones in V2) */
    short s_magic;           /* magic number to recognize super-blocks */

    /* The following items are valid on disk only for V3 and above */

    /* The block size in bytes. Minimum MIN_BLOCK_SIZE. SECTOR_SIZE
     * multiple. If V1 or V2 filesystem, this should be
     * initialised to STATIC_BLOCK_SIZE. Maximum MAX_BLOCK_SIZE.
     */
    short s_pad2;            /* try to avoid compiler-dependent padding */
    unsigned short s_block_size; /* block size in bytes. */
    char s_disk_version;     /* filesystem format sub-version */

    /* The following items are only used when the super_block is in memory. */
    struct inode *s_isup;    /* inode for root dir of mounted file sys */
    struct inode *s_imount;  /* inode mounted on */
    unsigned s_inodes_per_block; /* precalculated from magic number */
    dev_t s_dev;            /* whose super block is this? */
    int s_rd_only;          /* set to 1 iff file sys mounted read only */
    int s_native;           /* set to 1 iff not byte swapped file system */
    int s_version;          /* file system version, zero means bad magic */
    int s_ndzones;          /* # direct zones in an inode */
    int s_nindirs;          /* # indirect zones per indirect block */
    bit_t s_isearch;        /* inodes below this bit number are in use */
    bit_t s_zsearch;        /* all zones below this bit number are in use */
} super_block[NR_SUPERS];

#define NIL_SUPER (struct super_block *) 0
#define IMAP      0          /* operating on the inode bit map */
#define ZMAP      1          /* operating on the zone bit map */

```

```
/* This file contains the table used to map system call numbers onto the
 * routines that perform them.
 */
```

```
#define _TABLE
```

```
#include "fs.h"
#include <minix/callnr.h>
#include <minix/com.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "lock.h"
#include "super.h"
```

```
PUBLIC _PROTOTYPE (int (*call_vec[]), (void) ) = {
    no_sys,      /* 0 = unused */
    no_sys,      /* 1 = (exit) */
    no_sys,      /* 2 = (fork) */
    do_read,     /* 3 = read */
    do_write,    /* 4 = write */
    do_open,     /* 5 = open */
    do_close,    /* 6 = close */
    no_sys,      /* 7 = wait */
    do_creat,    /* 8 = creat */
    do_link,     /* 9 = link */
    do_unlink,   /* 10 = unlink */
    no_sys,      /* 11 = waitpid */
    do_chdir,    /* 12 = chdir */
    no_sys,      /* 13 = time */
    do_mknod,    /* 14 = mknod */
    do_chmod,    /* 15 = chmod */
    do_chown,    /* 16 = chown */
    no_sys,      /* 17 = break */
    do_stat,     /* 18 = stat */
    do_lseek,    /* 19 = lseek */
    no_sys,      /* 20 = getpid */
    do_mount,    /* 21 = mount */
    do_umount,   /* 22 = umount */
    no_sys,      /* 23 = (setuid) */
    no_sys,      /* 24 = getuid */
    do_stime,    /* 25 = stime */
    no_sys,      /* 26 = ptrace */
    no_sys,      /* 27 = alarm */
    do_fstat,    /* 28 = fstat */
    no_sys,      /* 29 = pause */
    do_utime,    /* 30 = utime */
    no_sys,      /* 31 = (stty) */
    no_sys,      /* 32 = (gtty) */
    do_access,   /* 33 = access */
    no_sys,      /* 34 = (nice) */
    no_sys,      /* 35 = (ftime) */
    do_sync,     /* 36 = sync */
    no_sys,      /* 37 = kill */
    do_rename,   /* 38 = rename */
    do_mkdir,    /* 39 = mkdir */
    do_unlink,   /* 40 = rmdir */
    do_dup,      /* 41 = dup */
    do_pipe,     /* 42 = pipe */
    no_sys,      /* 43 = times */
    no_sys,      /* 44 = (prof) */
    do_slink,    /* 45 = symlink */
    no_sys,      /* 46 = (setgid) */
    no_sys,      /* 47 = getgid */
    no_sys,      /* 48 = (signal) */
    do_rdlink,   /* 49 = readlink */
    do_lstat,    /* 50 = lstat */
    no_sys,      /* 51 = (acct) */
    no_sys,      /* 52 = (phys) */
    no_sys,      /* 53 = (lock) */
    do_ioctl,    /* 54 = ioctl */
    do_fcntl,    /* 55 = fcntl */
    no_sys,      /* 56 = (mpx) */
}
```

```
no_sys,      /* 57 = unused */
no_sys,      /* 58 = unused */
no_sys,      /* 59 = (execve) */
do_umask,    /* 60 = umask */
do_chroot,   /* 61 = chroot */
no_sys,      /* 62 = (setsid) */
no_sys,      /* 63 = (getpgrp) */

no_sys,      /* 64 = unused */
no_sys,      /* 65 = unused */
no_sys,      /* 66 = unused */
no_sys,      /* 67 = unused */
no_sys,      /* 68 = unused */
no_sys,      /* 69 = unused */
no_sys,      /* 70 = unused */
no_sys,      /* 71 = (sigaction) */
no_sys,      /* 72 = (sigsuspend) */
no_sys,      /* 73 = (sigpending) */
no_sys,      /* 74 = (sigprocmask) */
no_sys,      /* 75 = (sigreturn) */
no_sys,      /* 76 = (reboot) */
do_svrctl,   /* 77 = svrctl */

no_sys,      /* 78 = (sysuname) */
do_getsysinfo, /* 79 = getsysinfo */
no_sys,      /* 80 = unused */
no_sys,      /* 81 = unused */
do_fstatfs,  /* 82 = fstatfs */
no_sys,      /* 83 = unused */
no_sys,      /* 84 = unused */
do_select,   /* 85 = select */
do_fchdir,   /* 86 = fchdir */
do_fsync,    /* 87 = fsync */
no_sys,      /* 88 = (getpriority) */
no_sys,      /* 89 = (setpriority) */
no_sys,      /* 90 = (gettimeofday) */
no_sys,      /* 91 = (seteuid) */
no_sys,      /* 92 = (setegid) */
do_truncate, /* 93 = truncate */
do_ftruncate, /* 94 = truncate */
do_chmod,    /* 95 = fchmod */
do_chown,    /* 96 = fchown */
};
/* This should not fail with "array size is negative": */
extern int dummy[sizeof(call_vec) == NCALLS * sizeof(call_vec[0]) ? 1 : -1];
```

```

/* This file takes care of those system calls that deal with time.
 *
 * The entry points into this file are
 *   do_utime:      perform the UTIME system call
 *   do_stime:      PM informs FS about STIME system call
 */

#include "fs.h"
#include <minix/callnr.h>
#include <minix/com.h>
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"

/*=====
 *                               do_utime                               *
 *=====*/
PUBLIC int do_utime()
{
    /* Perform the utime(name, timep) system call. */

    register struct inode *rip;
    register int len, r;

    /* Adjust for case of 'timep' being NULL;
     * utime_strlen then holds the actual size: strlen(name)+1.
     */
    len = m_in.utime_length;
    if (len == 0) len = m_in.utime_strlen;

    /* Temporarily open the file. */
    if (fetch_name(m_in.utime_file, len, M1) != OK) return(err_code);
    if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);

    /* Only the owner of a file or the super_user can change its time. */
    r = OK;
    if (rip->i_uid != fp->fp_effuid && !super_user) r = EPERM;
    if (m_in.utime_length == 0 && r != OK) r = forbidden(rip, W_BIT);
    if (read_only(rip) != OK) r = EROFS; /* not even su can touch if R/O */
    if (r == OK) {
        if (m_in.utime_length == 0) {
            rip->i_atime = clock_time();
            rip->i_mtime = rip->i_atime;
        } else {
            rip->i_atime = m_in.utime_actime;
            rip->i_mtime = m_in.utime_modtime;
        }
        rip->i_update = CTIME; /* discard any stale ATIME and MTIME flags */
        rip->i_dirt = DIRTY;
    }

    put_inode(rip);
    return(r);
}

/*=====
 *                               do_stime                               *
 *=====*/
PUBLIC int do_stime()
{
    /* Perform the stime(tp) system call. */
    boottime = (long) m_in.pm_stime;
    return(OK);
}

```



```
/* FS timers library
 */

#include "fs.h"

#include <timers.h>
#include <minix/syslib.h>
#include <minix/com.h>

PRIVATE timer_t *fs_timers = NULL;

PUBLIC void fs_set_timer(timer_t *tp, int ticks, tmr_func_t watchdog, int arg)
{
    int r;
    clock_t now, old_head = 0, new_head;

    if ((r = getuptime(&now)) != OK)
        panic(__FILE__, "FS couldn't get uptime from system task.", NO_NUM);

    tmr_arg(tp)->ta_int = arg;

    old_head = tmrs_settimer(&fs_timers, tp, now+ticks, watchdog, &new_head);

    /* reschedule our synchronous alarm if necessary */
    if (!old_head || old_head > new_head) {
        if (sys_setalarm(new_head, 1) != OK)
            panic(__FILE__, "FS set timer "
                "couldn't set synchronous alarm.", NO_NUM);
    }

    return;
}

PUBLIC void fs_expire_timers(clock_t now)
{
    clock_t new_head;
    tmrs_exptimers(&fs_timers, now, &new_head);
    if (new_head > 0) {
        if (sys_setalarm(new_head, 1) != OK)
            panic(__FILE__, "FS expire timer couldn't set "
                "synchronous alarm.", NO_NUM);
    }
}

PUBLIC void fs_init_timer(timer_t *tp)
{
    tmr_inittimer(tp);
}

PUBLIC void fs_cancel_timer(timer_t *tp)
{
    clock_t new_head, old_head;
    old_head = tmrs_clrtimer(&fs_timers, tp, &new_head);

    /* if the earliest timer has been removed, we have to set
     * the synalarm to the next timer, or cancel the synalarm
     * altogether if th last time has been cancelled (new_head
     * will be 0 then).
     */
    if (old_head < new_head || !new_head) {
        if (sys_setalarm(new_head, 1) != OK)
            panic(__FILE__,
                "FS expire timer couldn't set synchronous alarm.",
                NO_NUM);
    }
}
```

```
/* Declaration of the V1 inode as it is on the disk (not in core). */
typedef struct {
    mode_t d1_mode; /* V1.x disk inode */
    uid_t d1_uid; /* file type, protection, etc. */
    off_t d1_size; /* user id of the file's owner */
    time_t d1_mtime; /* current file size in bytes */
    u8_t d1_gid; /* when was file data last changed */
    u8_t d1_nlinks; /* group number */
    ul6_t d1_zone[V1_NR_TZONES]; /* how many links to this file */
} d1_inode; /* block nums for direct, ind, and dbl ind */

/* Declaration of the V2 inode as it is on the disk (not in core). */
typedef struct {
    mode_t d2_mode; /* V2.x disk inode */
    ul6_t d2_nlinks; /* file type, protection, etc. */
    uid_t d2_uid; /* how many links to this file. HACK! */
    ul6_t d2_gid; /* user id of the file's owner. */
    off_t d2_size; /* group number HACK! */
    time_t d2_atime; /* current file size in bytes */
    time_t d2_mtime; /* when was file data last accessed */
    time_t d2_ctime; /* when was file data last changed */
    zone_t d2_zone[V2_NR_TZONES]; /* when was inode data last changed */
} d2_inode; /* block nums for direct, ind, and dbl ind */
```

```

/* This file contains a few general purpose utility routines.
 *
 * The entry points into this file are
 *   clock_time:  ask the clock task for the real time
 *   copy:        copy a block of data
 *   fetch_name:  go get a path name from user space
 *   no_sys:      reject a system call that FS does not handle
 *   panic:       something awful has occurred; MINIX cannot continue
 *   conv2:       do byte swapping on a 16-bit int
 *   conv4:       do byte swapping on a 32-bit long
 */

#include "fs.h"
#include <minix/com.h>
#include <minix/endpoint.h>
#include <unistd.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"

PRIVATE int panicking;          /* inhibits recursive panics during sync */

/*=====
 *                               clock_time                               *
 *=====*/
PUBLIC time_t clock_time()
{
/* This routine returns the time in seconds since 1.1.1970. MINIX is an
 * astrophysically naive system that assumes the earth rotates at a constant
 * rate and that such things as leap seconds do not exist.
 */

    register int k;
    clock_t uptime;

    if ( (k=getuptime(&uptime)) != OK) panic(__FILE__, "clock_time err", k);
    return( (time_t) (boottime + (uptime/HZ)));
}

/*=====
 *                               fetch_name                               *
 *=====*/
PUBLIC int fetch_name(path, len, flag)
char *path;          /* pointer to the path in user space */
int len;             /* path length, including 0 byte */
int flag;            /* M3 means path may be in message */
{
/* Go get path and put it in 'user_path'.
 * If 'flag' = M3 and 'len' <= M3_STRING, the path is present in 'message'.
 * If it is not, go copy it from user space.
 */

    register char *rpu, *rpm;
    int r;

    /* Check name length for validity. */
    if (len <= 0) {
        err_code = EINVAL;
        return(EGENERIC);
    }

    if (len > PATH_MAX) {
        err_code = ENAMETOOLONG;
        return(EGENERIC);
    }

    if (flag == M3 && len <= M3_STRING) {
        /* Just copy the path from the message to 'user_path'. */
        rpu = &user_path[0];
        rpm = m_in.pathname;          /* contained in input message */
        do { *rpu++ = *rpm++; } while (--len);
        r = OK;
    } else {

```

```

    /* String is not contained in the message. Get it from user space. */
    r = sys_datacopy(who_e, (vir_bytes) path,
                     FS_PROC_NR, (vir_bytes) user_path, (phys_bytes) len);
}
return(r);
}

/*=====
 *
 *                               no_sys
 *=====*/
PUBLIC int no_sys()
{
    /* Somebody has used an illegal system call number */
    printf("FS: in no_sys: call %d from %d\n", call_nr, who_e);
    return(EINVAL);
}

/*=====
 *
 *                               panic
 *=====*/
PUBLIC void panic(who, mess, num)
char *who;          /* who caused the panic */
char *mess;         /* panic message string */
int num;            /* number to go with it */
{
    /* Something awful has happened. Panics are caused when an internal
     * inconsistency is detected, e.g., a programming error or illegal value of a
     * defined constant.
     */
    if (panicking) return;          /* do not panic during a sync */
    panicking = TRUE;              /* prevent another panic during the sync */

    printf("FS panic(%)s: %s ", who, mess);
    if (num != NO_NUM) printf("%d", num);
    (void) do_sync();              /* flush everything to the disk */
    sys_exit(SELF);
}

/*=====
 *
 *                               conv2
 *=====*/
PUBLIC unsigned conv2(norm, w)
int norm;           /* TRUE if no swap, FALSE for byte swap */
int w;              /* promotion of 16-bit word to be swapped */
{
    /* Possibly swap a 16-bit word between 8086 and 68000 byte order. */
    if (norm) return( (unsigned) w & 0xFFFF);
    return( ((w&BYTE) << 8) | ((w>>8) & BYTE));
}

/*=====
 *
 *                               conv4
 *=====*/
PUBLIC long conv4(norm, x)
int norm;           /* TRUE if no swap, FALSE for byte swap */
long x;             /* 32-bit long to be byte swapped */
{
    /* Possibly swap a 32-bit long between 8086 and 68000 byte order. */
    unsigned lo, hi;
    long l;

    if (norm) return(x);          /* byte order was already ok */
    lo = conv2(FALSE, (int) x & 0xFFFF); /* low-order half, byte swapped */
    hi = conv2(FALSE, (int) (x>>16) & 0xFFFF); /* high-order half, swapped */
    l = ( (long) lo <<16) | hi;
    return(l);
}

/*=====
 *
 *                               isokendpt_f
 *=====*/
PUBLIC int isokendpt_f(char *file, int line, int endpoint, int *proc, int fatal)
{
    int failed = 0;

```

```
*proc = _ENDPOINT_P(endpoint);
if(*proc < 0 || *proc >= NR_PROCS) {
    printf("FS:%s:%d: proc (%d) from endpoint (%d) out of range\n",
        file, line, *proc, endpoint);
    failed = 1;
} else if(fproc[*proc].fp_endpoint != endpoint) {
    printf("FS:%s:%d: proc (%d) from endpoint (%d) doesn't match "
        "known endpoint (%d)\n",
        file, line, *proc, endpoint, fproc[*proc].fp_endpoint);
    failed = 1;
}

if(failed && fatal)
    panic(__FILE__, "isokendpt_f failed", NO_NUM);

return failed ? EDEADSRCDST : OK;
}
```

```

/* This file is the counterpart of "read.c". It contains the code for writing
 * insofar as this is not contained in read_write().
 *
 * The entry points into this file are
 *   do_write:      call read_write to perform the WRITE system call
 *   clear_zone:    erase a zone in the middle of a file
 *   new_block:     acquire a new block
 */

#include "fs.h"
#include <string.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "super.h"

FORWARD _PROTOTYPE( void wr_indir, (struct buf *bp, int index, zone_t zone) );
FORWARD _PROTOTYPE( int empty_indir, (struct buf *, struct super_block *) );

/*=====
 *
 * do_write
 *=====*/
PUBLIC int do_write()
{
    /* Perform the write(fd, buffer, nbytes) system call. */

    return(read_write(WRITING));
}

/*=====
 *
 * write_map
 *=====*/
PUBLIC int write_map(rip, position, new_zone, op)
struct inode *rip;          /* pointer to inode to be changed */
off_t position;             /* file address to be mapped */
zone_t new_zone;            /* zone # to be inserted */
int op;                     /* special actions */
{
    /* Write a new zone into an inode.
     *
     * If op includes WMAP_FREE, free the data zone corresponding to that position
     * in the inode ('new_zone' is ignored then). Also free the indirect block
     * if that was the last entry in the indirect block.
     * Also free the double indirect block if that was the last entry in the
     * double indirect block.
     */
    int scale, ind_ex, new_ind, new_dbl, zones, nr_indirects, single, zindex, ex;
    zone_t z, z1, z2 = NO_ZONE, old_zone;
    register block_t b;
    long excess, zone;
    struct buf *bp_dindir = NIL_BUF, *bp = NIL_BUF;

    rip->i_dirt = DIRTY;          /* inode will be changed */
    scale = rip->i_sp->s_log_zone_size; /* for zone-block conversion */
    /* relative zone # to insert */
    zone = (position/rip->i_sp->s_block_size) >> scale;
    zones = rip->i_ndzones;        /* # direct zones in the inode */
    nr_indirects = rip->i_nindirs; /* # indirect zones per indirect block */

    /* Is 'position' to be found in the inode itself? */
    if (zone < zones) {
        zindex = (int) zone;      /* we need an integer here */
        if(rip->i_zone[zindex] != NO_ZONE && (op & WMAP_FREE)) {
            free_zone(rip->i_dev, rip->i_zone[zindex]);
            rip->i_zone[zindex] = NO_ZONE;
        } else {
            rip->i_zone[zindex] = new_zone;
        }
        return(OK);
    }

    /* It is not in the inode, so it must be single or double indirect. */
    excess = zone - zones;        /* first Vx_NR_DZONES don't count */

```

```

new_ind = FALSE;
new_dbl = FALSE;

if (excess < nr_indirects) {
    /* 'position' can be located via the single indirect block. */
    z1 = rip->i_zone[zones];          /* single indirect zone */
    single = TRUE;
} else {
    /* 'position' can be located via the double indirect block. */
    if ( (z2 = z = rip->i_zone[zones+1]) == NO_ZONE &&
        !(op & WMAP_FREE)) {
        /* Create the double indirect block. */
        if ( (z = alloc_zone(rip->i_dev, rip->i_zone[0])) == NO_ZONE)
            return(err_code);
        rip->i_zone[zones+1] = z;
        new_dbl = TRUE; /* set flag for later */
    }

    /* 'z' is zone number for double indirect block, either old
     * or newly created.
     * If there wasn't one and WMAP_FREE is set, 'z' is NO_ZONE.
     */
    excess -= nr_indirects; /* single indirect doesn't count */
    ind_ex = (int) (excess / nr_indirects);
    excess = excess % nr_indirects;
    if (ind_ex >= nr_indirects) return(EFBIG);

    if(z == NO_ZONE) {
        /* WMAP_FREE and no double indirect block - then no
         * single indirect block either.
         */
        z1 = NO_ZONE;
    } else {
        b = (block_t) z << scale;
        bp_dindir = get_block(rip->i_dev, b, (new_dbl?NO_READ:NORMAL));
        if (new_dbl) zero_block(bp_dindir);
        z1 = rd_indir(bp_dindir, ind_ex);
    }
    single = FALSE;
}

/* z1 is now single indirect zone, or NO_ZONE; 'excess' is index.
 * We have to create the indirect zone if it's NO_ZONE. Unless
 * we're freeing (WMAP_FREE).
 */
if (z1 == NO_ZONE && !(op & WMAP_FREE)) {
    z1 = alloc_zone(rip->i_dev, rip->i_zone[0]);
    if (single)
        rip->i_zone[zones] = z1; /* update inode w. single indirect */
    else
        wr_indir(bp_dindir, ind_ex, z1);          /* update dbl indir */

    new_ind = TRUE;
    /* If double ind, it is dirty. */
    if (bp_dindir != NIL_BUF) bp_dindir->b_dirty = DIRTY;
    if (z1 == NO_ZONE) {
        /* Release dbl indirect blk. */
        put_block(bp_dindir, INDIRECT_BLOCK);
        return(err_code); /* couldn't create single ind */
    }
}

/* z1 is indirect block's zone number (unless it's NO_ZONE when we're
 * freeing).
 */
if(z1 != NO_ZONE) {
    ex = (int) excess;          /* we need an int here */
    b = (block_t) z1 << scale;
    bp = get_block(rip->i_dev, b, (new_ind ? NO_READ : NORMAL) );
    if (new_ind) zero_block(bp);
    if(op & WMAP_FREE) {
        if((old_zone = rd_indir(bp, ex)) != NO_ZONE) {
            free_zone(rip->i_dev, old_zone);
            wr_indir(bp, ex, NO_ZONE);
        }
    }
}

```

```

    }

    /* Last reference in the indirect block gone? Then
     * Free the indirect block.
     */
    if(empty_indir(bp, rip->i_sp)) {
        free_zone(rip->i_dev, z1);
        z1 = NO_ZONE;
        /* Update the reference to the indirect block to
         * NO_ZONE - in the double indirect block if there
         * is one, otherwise in the inode directly.
         */
        if(single) {
            rip->i_zone[zones] = z1;
        } else {
            wr_indir(bp_dindir, ind_ex, z1);
            bp_dindir->b_dirt = DIRTY;
        }
    }
    } else {
        wr_indir(bp, ex, new_zone);
    }
    bp->b_dirt = DIRTY;
    put_block(bp, INDIRECT_BLOCK);
}

/* If the single indirect block isn't there (or was just freed),
 * see if we have to keep the double indirect block.
 */
if(z1 == NO_ZONE && !single && empty_indir(bp_dindir, rip->i_sp) &&
    z2 != NO_ZONE) {
    free_zone(rip->i_dev, z2);
    rip->i_zone[zones+1] = NO_ZONE;
}

put_block(bp_dindir, INDIRECT_BLOCK); /* release double indirect blk */

return(OK);
}

/*=====
 *                               wr_indir                               *
 *=====*/
PRIVATE void wr_indir(bp, index, zone)
struct buf *bp;          /* pointer to indirect block */
int index;               /* index into *bp */
zone_t zone;             /* zone to write */
{
    /* Given a pointer to an indirect block, write one entry. */

    struct super_block *sp;

    if(bp == NIL_BUF)
        panic(__FILE__, "wr_indir() on NIL_BUF", NO_NUM);

    sp = get_super(bp->b_dev); /* need super block to find file sys type */

    /* write a zone into an indirect block */
    if (sp->s_version == V1)
        bp->b_v1_ind[index] = (zone_t) conv2(sp->s_native, (int) zone);
    else
        bp->b_v2_ind[index] = (zone_t) conv4(sp->s_native, (long) zone);
}

/*=====
 *                               empty_indir                           *
 *=====*/
PRIVATE int empty_indir(bp, sb)
struct buf *bp;          /* pointer to indirect block */
struct super_block *sb;  /* superblock of device block resides on */
{
    /* Return nonzero if the indirect block pointed to by bp contains
     * only NO_ZONE entries.
     */

```



```

    int i;
    if(sb->s_version == V1) {
        for(i = 0; i < V1_INDIRECTS; i++)
            if(bp->b_v1_ind[i] != NO_ZONE)
                return 0;
    } else {
        for(i = 0; i < V2_INDIRECTS(sb->s_block_size); i++)
            if(bp->b_v2_ind[i] != NO_ZONE)
                return 0;
    }

    return 1;
}

/*=====
 *                               clear_zone                               *
 *=====*/
PUBLIC void clear_zone(rip, pos, flag)
register struct inode *rip;    /* inode to clear */
off_t pos;                   /* points to block to clear */
int flag;                     /* 0 if called by read_write, 1 by new_block */
{
    /* Zero a zone, possibly starting in the middle. The parameter 'pos' gives
     * a byte in the first block to be zeroed. Clearzone() is called from
     * read_write and new_block().
     */

    register struct buf *bp;
    register block_t b, blo, bhi;
    register off_t next;
    register int scale;
    register zone_t zone_size;

    /* If the block size and zone size are the same, clear_zone() not needed. */
    scale = rip->i_sp->s_log_zone_size;
    if (scale == 0) return;

    zone_size = (zone_t) rip->i_sp->s_block_size << scale;
    if (flag == 1) pos = (pos/zone_size) * zone_size;
    next = pos + rip->i_sp->s_block_size - 1;

    /* If 'pos' is in the last block of a zone, do not clear the zone. */
    if (next/zone_size != pos/zone_size) return;
    if ( (blo = read_map(rip, next)) == NO_BLOCK) return;
    bhi = ( ((blo)>>scale)+1) << scale - 1;

    /* Clear all the blocks between 'blo' and 'bhi'. */
    for (b = blo; b <= bhi; b++) {
        bp = get_block(rip->i_dev, b, NO_READ);
        zero_block(bp);
        put_block(bp, FULL_DATA_BLOCK);
    }
}

/*=====
 *                               new_block                               *
 *=====*/
PUBLIC struct buf *new_block(rip, position)
register struct inode *rip;    /* pointer to inode */
off_t position;               /* file pointer */
{
    /* Acquire a new block and return a pointer to it. Doing so may require
     * allocating a complete zone, and then returning the initial block.
     * On the other hand, the current zone may still have some unused blocks.
     */

    register struct buf *bp;
    block_t b, base_block;
    zone_t z;
    zone_t zone_size;
    int scale, r;
    struct super_block *sp;

    /* Is another block available in the current zone? */

```

```

if ( (b = read_map(rip, position)) == NO_BLOCK) {
    /* Choose first zone if possible. */
    /* Lose if the file is nonempty but the first zone number is NO_ZONE
     * corresponding to a zone full of zeros. It would be better to
     * search near the last real zone.
     */
    if (rip->i_zone[0] == NO_ZONE) {
        sp = rip->i_sp;
        z = sp->s_firstdatazone;
    } else {
        z = rip->i_zone[0];    /* hunt near first zone */
    }
    if ( (z = alloc_zone(rip->i_dev, z)) == NO_ZONE) return(NIL_BUF);
    if ( (r = write_map(rip, position, z, 0)) != OK) {
        free_zone(rip->i_dev, z);
        err_code = r;
        return(NIL_BUF);
    }

    /* If we are not writing at EOF, clear the zone, just to be safe. */
    if ( position != rip->i_size) clear_zone(rip, position, 1);
    scale = rip->i_sp->s_log_zone_size;
    base_block = (block_t) z << scale;
    zone_size = (zone_t) rip->i_sp->s_block_size << scale;
    b = base_block + (block_t)((position % zone_size)/rip->i_sp->s_block_size);
}

bp = get_block(rip->i_dev, b, NO_READ);
zero_block(bp);
return(bp);
}

/*=====
 *                                zero_block                                *
 *=====*/
PUBLIC void zero_block(bp)
register struct buf *bp;    /* pointer to buffer to zero */
{
    /* Zero a block. */
    memset(bp->b_data, 0, _MAX_BLOCK_SIZE);
    bp->b_dirt = DIRTY;
}

```

```

/* This file handles the EXEC system call. It performs the work as follows:
 * - see if the permissions allow the file to be executed
 * - read the header and extract the sizes
 * - fetch the initial args and environment from the user space
 * - allocate the memory for the new process
 * - copy the initial stack from PM to the process
 * - read in the text and data segments and copy to the process
 * - take care of setuid and setgid bits
 * - fix up 'mproc' table
 * - tell kernel about EXEC
 * - save offset to initial argc (for ps)
 *
 * The entry points into this file are:
 * pm_exec: perform the EXEC system call
 */

#include "fs.h"
#include <sys/stat.h>
#include <minix/callnr.h>
#include <minix/endpoint.h>
#include <minix/com.h>
#include <a.out.h>
#include <signal.h>
#include <string.h>
#include "buf.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

FORWARD _PROTOTYPE( int exec_newmem, (int proc_e, vir_bytes text_bytes,
vir_bytes data_bytes, vir_bytes bss_bytes, vir_bytes tot_bytes,
vir_bytes frame_len, int sep_id,
Dev_t st_dev, ino_t st_ino, time_t st_ctime, char *progname,
int new_uid, int new_gid,
vir_bytes *stack_top, int *load_textp, int *allow_setuidp) );
FORWARD _PROTOTYPE( int read_header, (struct inode *rip, int *sep_id,
vir_bytes *text_bytes, vir_bytes *data_bytes,
vir_bytes *bss_bytes, phys_bytes *tot_bytes, vir_bytes *pc,
int *hdrlenp) );
FORWARD _PROTOTYPE( int patch_stack, (struct inode *rip,
char stack[ARG_MAX], vir_bytes *stk_bytes) );
FORWARD _PROTOTYPE( int insert_arg, (char stack[ARG_MAX],
vir_bytes *stk_bytes, char *arg, int replace) );
FORWARD _PROTOTYPE( void patch_ptr, (char stack[ARG_MAX],
vir_bytes base) );
FORWARD _PROTOTYPE( int read_seg, (struct inode *rip, off_t off,
int proc_e, int seg, phys_bytes seg_bytes) );
FORWARD _PROTOTYPE( void clo_exec, (struct fproc *rfp) );

#define ESCRIPT (-2000) /* Returned by read_header for a #! script. */
#define PTRSIZE sizeof(char *) /* Size of pointers in argv[] and envp[]. */

/*=====
 * pm_exec
 *=====*/
PUBLIC int pm_exec(proc_e, path, path_len, frame, frame_len)
int proc_e;
char *path;
vir_bytes path_len;
char *frame;
vir_bytes frame_len;
{
/* Perform the execve(name, argv, envp) call. The user library builds a
 * complete stack image, including pointers, args, environ, etc. The stack
 * is copied to a buffer inside FS, and then to the new core image.
 */
int r, sep_id, round, proc_s, hdrlen, load_text, allow_setuid;
vir_bytes text_bytes, data_bytes, bss_bytes, pc;
phys_bytes tot_bytes; /* total space for program, including gap */
vir_bytes stack_top, vsp;
off_t off;
uid_t new_uid;
gid_t new_gid;

```

```

struct fproc *rfp;
struct inode *rip;
char *cp;
char progname[PROC_NAME_LEN];

static char mbuf[ARG_MAX];    /* buffer for stack and zeroes */

okendpt(proc_e, &proc_s);
rfp= fp= &fproc[proc_s];
who_e= proc_e;
who_p= proc_s;
super_user = (fp->fp_effuid == SU_UID ? TRUE : FALSE);    /* su? */

/* Get the exec file name. */
r= fetch_name(path, path_len, 0);
if (r != OK)
{
    printf("pm_exec: fetch_name failed\n");
    return(r);    /* file name not in user data segment */
}

/* Fetch the stack from the user before destroying the old core image. */
if (frame_len > ARG_MAX)
{
    printf("pm_exec: bad frame_len\n");
    return(ENOMEM); /* stack too big */
}
r = sys_datacopy(proc_e, (vir_bytes) frame,
                  SELF, (vir_bytes) mbuf, (phys_bytes) frame_len);
/* can't fetch stack (e.g. bad virtual addr) */
if (r != OK)
{
    printf("pm_exec: sys_datacopy failed\n");
    return(r);
}

/* The default is the keep the original user and group IDs */
new_uid= rfp->fp_effuid;
new_gid= rfp->fp_effgid;

for (round= 0; round < 2; round++)
    /* round = 0 (first attempt), or 1 (interpreted script) */
    {
        /* Save the name of the program */
        (cp= strchr(user_path, '/')) ? cp++ : (cp= user_path);

        strncpy(progname, cp, PROC_NAME_LEN-1);
        progname[PROC_NAME_LEN-1] = '\0';

#if 0
        printf("pm_exec: eat_path '%s'\n", user_path);
#endif
        rip= eat_path(user_path);
        if (rip == NIL_INODE)
        {
            return(err_code);
        }
        if ((rip->i_mode & I_TYPE) != I_REGULAR)
            r = ENOEXEC;
        else
            r = forbidden(rip, X_BIT); /* check if file is executable */
        if (r != OK) {
            put_inode(rip);
            printf("pm_exec: bad executable\n");
            return(r);
        }

        if (round == 0)
        {
            /* Deal with setuid/setgid executables */
            if (rip->i_mode & I_SET_UID_BIT)
                new_uid = rip->i_uid;
            if (rip->i_mode & I_SET_GID_BIT)
                new_gid = rip->i_gid;
        }
    }

```

```

    }

    /* Read the file header and extract the segment sizes. */
    r = read_header(rip, &sep_id, &text_bytes, &data_bytes, &bss_bytes,
        &tot_bytes, &pc, &hdrlen);
    if (r != ESCRIPT || round != 0)
        break;

    /* Get fresh copy of the file name. */
    r = fetch_name(path, path_len, 0);
    if (r != OK)
    {
        printf("pm_exec: 2nd fetch_name failed\n");
        put_inode(rip);
        return(r); /* strange */
    }
    r = patch_stack(rip, mbuf, &frame_len);
    put_inode(rip);
    if (r != OK)
    {
        printf("pm_exec: patch stack\n");
        return r;
    }
}

if (r != OK)
{
    printf("pm_exec: returning ENOEXEC, r = %d\n", r);
    return ENOEXEC;
}

r = exec_newmem(proc_e, text_bytes, data_bytes, bss_bytes, tot_bytes,
    frame_len, sep_id, rip->i_dev, rip->i_num, rip->i_ctime,
    progname, new_uid, new_gid, &stack_top, &load_text, &allow_setuid);
if (r != OK)
{
    printf("pm_exec: exec_newmap failed: %d\n", r);
    put_inode(rip);
    return r;
}

/* Patch up stack and copy it from FS to new core image. */
vsp = stack_top;
vsp -= frame_len;
patch_ptr(mbuf, vsp);
r = sys_datacopy(SELF, (vir_bytes) mbuf,
    proc_e, (vir_bytes) vsp, (phys_bytes) frame_len);
if (r != OK) panic(__FILE__, "pm_exec stack copy err on", proc_e);

off = hdrlen;

/* Read in text and data segments. */
if (load_text) {
    r = read_seg(rip, off, proc_e, T, text_bytes);
}
off += text_bytes;
if (r == OK)
    r = read_seg(rip, off, proc_e, D, data_bytes);

put_inode(rip);

if (r != OK) return r;

clo_exec(rfp);

if (allow_setuid)
{
    rfp->fp_effuid = new_uid;
    rfp->fp_effgid = new_gid;
}

/* This child has now exec()ed. */
rfp->fp_excced = 1;

```

```

/* Check if this is a driver that can now be useful. */
dmap_endpt_up(rfp->fp_endpoint);

return OK;
}

/*=====
 *                               exec_newmem                               *
 *=====*/
PRIVATE int exec_newmem(proc_e, text_bytes, data_bytes, bss_bytes, tot_bytes,
                        frame_len, sep_id, st_dev, st_ino, st_ctime, progame,
                        new_uid, new_gid, stack_top, load_textp, allow_setuidp)
int proc_e;
vir_bytes text_bytes;
vir_bytes data_bytes;
vir_bytes bss_bytes;
vir_bytes tot_bytes;
vir_bytes frame_len;
int sep_id;
dev_t st_dev;
ino_t st_ino;
time_t st_ctime;
int new_uid;
int new_gid;
char *progame;
vir_bytes *stack_top;
int *load_textp;
int *allow_setuidp;
{
    int r;
    struct exec_newmem e;
    message m;

    e.text_bytes= text_bytes;
    e.data_bytes= data_bytes;
    e.bss_bytes= bss_bytes;
    e.tot_bytes= tot_bytes;
    e.args_bytes= frame_len;
    e.sep_id= sep_id;
    e.st_dev= st_dev;
    e.st_ino= st_ino;
    e.st_ctime= st_ctime;
    e.new_uid= new_uid;
    e.new_gid= new_gid;
    strncpy(e.progame, progame, sizeof(e.progame)-1);
    e.progame[sizeof(e.progame)-1]= '\0';

    m.m_type= EXEC_NEWMEM;
    m.EXC_NM_PROC= proc_e;
    m.EXC_NM_PTR= (char *)&e;
    r= sendrec(PM_PROC_NR, &m);
    if (r != OK)
        return r;

#ifdef 0
    printf("exec_newmem: r = %d, m_type = %d\n", r, m.m_type);
#endif
    *stack_top= m.ml_i1;
    *load_textp= !(m.ml_i2 & EXC_NM_RF_LOAD_TEXT);
    *allow_setuidp= !(m.ml_i2 & EXC_NM_RF_ALLOW_SETUID);

#ifdef 0
    printf("exec_newmem: stack_top = 0x%x\n", *stack_top);
    printf("exec_newmem: load_text = %d\n", *load_textp);
#endif
    return m.m_type;
}

/*=====
 *                               read_header                               *
 *=====*/
PRIVATE int read_header(rip, sep_id, text_bytes, data_bytes, bss_bytes,
                        tot_bytes, pc, hdrlenp)
struct inode *rip;
/* inode for reading exec file */

```

```

int *sep_id;                /* true iff sep I&D */
vir_bytes *text_bytes;      /* place to return text size */
vir_bytes *data_bytes;      /* place to return initialized data size */
vir_bytes *bss_bytes;       /* place to return bss size */
phys_bytes *tot_bytes;      /* place to return total size */
vir_bytes *pc;              /* program entry point (initial PC) */
int *hdrlenp;
{
/* Read the header and extract the text, data, bss and total sizes from it. */
off_t pos;
block_t b;
struct buf *bp;
struct exec hdr;           /* a.out header is read in here */

/* Read the header and check the magic number. The standard MINIX header
 * is defined in <a.out.h>. It consists of 8 chars followed by 6 longs.
 * Then come 4 more longs that are not used here.
 *   Byte 0: magic number 0x01
 *   Byte 1: magic number 0x03
 *   Byte 2: normal = 0x10 (not checked, 0 is OK), separate I/D = 0x20
 *   Byte 3: CPU type, Intel 16 bit = 0x04, Intel 32 bit = 0x10,
 *           Motorola = 0x0B, Sun SPARC = 0x17
 *   Byte 4: Header length = 0x20
 *   Bytes 5-7 are not used.
 *
 * Now come the 6 longs
 *   Bytes 8-11: size of text segments in bytes
 *   Bytes 12-15: size of initialized data segment in bytes
 *   Bytes 16-19: size of bss in bytes
 *   Bytes 20-23: program entry point
 *   Bytes 24-27: total memory allocated to program (text, data + stack)
 *   Bytes 28-31: size of symbol table in bytes
 * The longs are represented in a machine dependent order,
 * little-endian on the 8088, big-endian on the 68000.
 * The header is followed directly by the text and data segments, and the
 * symbol table (if any). The sizes are given in the header. Only the
 * text and data segments are copied into memory by exec. The header is
 * used here only. The symbol table is for the benefit of a debugger and
 * is ignored here.
 */

pos = 0;                   /* Read from the start of the file */
b = read_map(rip, pos);    /* get block number */

if (b == 0)                /* Hole */
    return ENOEXEC;

bp = get_block(rip->i_dev, b, NORMAL);    /* get block */

/* Interpreted script? */
if (bp->b_data[0] == '#' && bp->b_data[1] == '!' && rip->i_size >= 2)
{
    put_block(bp, FULL_DATA_BLOCK);
    return ESCRIPT;
}

memcpy(&hdr, bp->b_data, sizeof(hdr));
put_block(bp, FULL_DATA_BLOCK);

if (rip->i_size < A_MINHDR) return(ENOEXEC);

/* Check magic number, cpu type, and flags. */
if (BADMAG(hdr)) return(ENOEXEC);
#if CHIP == INTEL && _WORD_SIZE == 2
    if (hdr.a_cpu != A_I8086) return(ENOEXEC);
#endif
#if CHIP == INTEL && _WORD_SIZE == 4
    if (hdr.a_cpu != A_I80386) return(ENOEXEC);
#endif
if ((hdr.a_flags & ~(A_NSYM | A_EXEC | A_SEP)) != 0) return(ENOEXEC);

*sep_id = !(hdr.a_flags & A_SEP);    /* separate I & D or not */

/* Get text and data sizes. */

```

```

*text_bytes = (vir_bytes) hdr.a_text; /* text size in bytes */
*data_bytes = (vir_bytes) hdr.a_data; /* data size in bytes */
*bss_bytes = (vir_bytes) hdr.a_bss; /* bss size in bytes */
*tot_bytes = hdr.a_total; /* total bytes to allocate for prog */
if (*tot_bytes == 0) return(ENOEXEC);

if (!*sep_id) {
    /* If I & D space is not separated, it is all considered data. Text=0*/
    *data_bytes += *text_bytes;
    *text_bytes = 0;
}
*pc = hdr.a_entry; /* initial address to start execution */
*hdrlenp = hdr.a_hdrlen & BYTE; /* header length */

return(OK);
}

/*=====
*
* patch_stack
*=====*/
PRIVATE int patch_stack(rip, stack, stk_bytes)
struct inode *rip; /* pointer for open script file */
char stack[ARG_MAX]; /* pointer to stack image within FS */
vir_bytes *stk_bytes; /* size of initial stack */
{
    /* Patch the argument vector to include the path name of the script to be
    * interpreted, and all strings on the #! line. Returns the path name of
    * the interpreter.
    */
    enum { INSERT=FALSE, REPLACE=TRUE };
    int n;
    off_t pos;
    block_t b;
    struct buf *bp;
    char *sp, *interp = NULL;

    /* Make user_path the new argv[0]. */
    if (!insert_arg(stack, stk_bytes, user_path, REPLACE)) return(ENOMEM);

    pos = 0; /* Read from the start of the file */
    b = read_map(rip, pos); /* get block number */
    if (b == 0) /* Hole */
        return ENOEXEC;

    bp = get_block(rip->i_dev, b, NORMAL); /* get block */
    n = rip->i_size;
    if (n > rip->i_sp->s_block_size)
        n = rip->i_sp->s_block_size;
    if (n < 2)
    {
        put_block(bp, FULL_DATA_BLOCK);
        return ENOEXEC;
    }
    sp = bp->b_data+2; /* just behind the #! */
    n -= 2;
    if (n > PATH_MAX)
        n = PATH_MAX;

    /* Use the user_path variable for temporary storage */
    memcpy(user_path, sp, n);
    put_block(bp, FULL_DATA_BLOCK);

    if ((sp = memchr(user_path, '\n', n)) == NULL) /* must be a proper line */
        return(ENOEXEC);

    /* Move sp backwards through script[], prepending each string to stack. */
    for (;;) {
        /* skip spaces behind argument. */
        while (sp > user_path && (*--sp == ' ' || *sp == '\t')) {}
        if (sp == user_path) break;

        sp[1] = 0;
        /* Move to the start of the argument. */
        while (sp > user_path && sp[-1] != ' ' && sp[-1] != '\t') --sp;

```



```

    interp = sp;
    if (!insert_arg(stack, stk_bytes, sp, INSERT)) return(ENOMEM);
}

/* Round *stk_bytes up to the size of a pointer for alignment constraints. */
*stk_bytes= ((*stk_bytes + PTRSIZE - 1) / PTRSIZE) * PTRSIZE;

if (interp != user_path)
    memmove(user_path, interp, strlen(interp)+1);
return(OK);
}

/*=====
*
*                               insert_arg
*=====*/
PRIVATE int insert_arg(stack, stk_bytes, arg, replace)
char stack[ARG_MAX];          /* pointer to stack image within PM */
vir_bytes *stk_bytes;         /* size of initial stack */
char *arg;                    /* argument to prepend/replace as new argv[0] */
int replace;
{
/* Patch the stack so that arg will become argv[0]. Be careful, the stack may
* be filled with garbage, although it normally looks like this:
*     nargs argv[0] ... argv[nargs-1] NULL envp[0] ... NULL
* followed by the strings "pointed" to by the argv[i] and the envp[i]. The
* pointers are really offsets from the start of stack.
* Return true iff the operation succeeded.
*/
    int offset, a0, a1, old_bytes = *stk_bytes;

    /* Prepending arg adds at least one string and a zero byte. */
    offset = strlen(arg) + 1;

    a0 = (int) ((char **) stack)[1]; /* argv[0] */
    if (a0 < 4 * PTRSIZE || a0 >= old_bytes) return(FALSE);

    a1 = a0; /* a1 will point to the strings to be moved */
    if (replace) {
        /* Move a1 to the end of argv[0][] (argv[1] if nargs > 1). */
        do {
            if (a1 == old_bytes) return(FALSE);
            --offset;
        } while (stack[a1++] != 0);
    } else {
        offset += PTRSIZE; /* new argv[0] needs new pointer in argv[] */
        a0 += PTRSIZE; /* location of new argv[0][]. */
    }

    /* stack will grow by offset bytes (or shrink by -offset bytes) */
    if ((*stk_bytes += offset) > ARG_MAX) return(FALSE);

    /* Reposition the strings by offset bytes */
    memmove(stack + a1 + offset, stack + a1, old_bytes - a1);

    strcpy(stack + a0, arg); /* Put arg in the new space. */

    if (!replace) {
        /* Make space for a new argv[0]. */
        memmove(stack + 2 * PTRSIZE, stack + 1 * PTRSIZE, a0 - 2 * PTRSIZE);

        ((char **) stack)[0]++; /* nargs++; */
    }
    /* Now patch up argv[] and envp[] by offset. */
    patch_ptr(stack, (vir_bytes) offset);
    ((char **) stack)[1] = (char *) a0; /* set argv[0] correctly */
    return(TRUE);
}

/*=====
*
*                               patch_ptr
*=====*/
PRIVATE void patch_ptr(stack, base)
char stack[ARG_MAX];          /* pointer to stack image within PM */

```

```

vir_bytes base;                /* virtual address of stack base inside user */
{
/* When doing an exec(name, argv, envp) call, the user builds up a stack
 * image with arg and env pointers relative to the start of the stack. Now
 * these pointers must be relocated, since the stack is not positioned at
 * address 0 in the user's address space.
 */

    char **ap, flag;
    vir_bytes v;

    flag = 0;                  /* counts number of 0-pointers seen */
    ap = (char **) stack;      /* points initially to 'nargs' */
    ap++;                      /* now points to argv[0] */
    while (flag < 2) {
        if (ap >= (char **) &stack[ARG_MAX]) return; /* too bad */
        if (*ap != NULL) {
            v = (vir_bytes) *ap; /* v is relative pointer */
            v += base;           /* relocate it */
            *ap = (char *) v;    /* put it back */
        } else {
            flag++;
        }
        ap++;
    }
}

/*=====
 *                      read_seg                      *
 *=====*/
PRIVATE int read_seg(rip, off, proc_e, seg, seg_bytes)
struct inode *rip;             /* inode descriptor to read from */
off_t off;                     /* offset in file */
int proc_e;                    /* process number (endpoint) */
int seg;                       /* T, D, or S */
phys_bytes seg_bytes;          /* how much is to be transferred? */
{
/*
 * The byte count on read is usually smaller than the segment count, because
 * a segment is padded out to a click multiple, and the data segment is only
 * partially initialized.
 */

    int r, block_size;
    off_t n, o, b_off, seg_off;
    block_t b;
    struct buf *bp;

    /* Make sure that the file is big enough */
    if (rip->i_size < off+seg_bytes)
        return EIO;

    block_size= rip->i_sp->s_block_size;
    seg_off= 0;
    for (o= off - (off % block_size); o < off+seg_bytes; o += block_size)
    {
        b= read_map(rip, o);
        if (b == NO_BLOCK)
            return EIO; /* Executables don't have holes */

        bp = get_block(rip->i_dev, b, NORMAL); /* get block */
        if (o < off)
            b_off= off-o;
        else
            b_off= 0;
        n= block_size-b_off;
        if (o+b_off+n > off+seg_bytes)
            n= off+seg_bytes-(o+b_off);
        r= sys_vircopy(SELFS, D, (vir_bytes)bp->b_data+b_off,
            proc_e, seg, seg_off, n);
        put_block(bp, FULL_DATA_BLOCK);

        if (r != OK)
            return r;
    }
}

```

```
        seg_off += n;
    }

    return OK;
}

/*=====
 *                                clo_exec                                *
 *=====*/
PRIVATE void clo_exec(rfp)
struct fproc *rfp;
{
    /* Files can be marked with the FD_CLOEXEC bit (in fp->fp_cloexec).
    */
    int i, proc;
    long bitmap;

    /* The array of FD_CLOEXEC bits is in the fp_cloexec bit map. */
    bitmap = rfp->fp_cloexec;
    if (bitmap) {
        /* Check the file descriptors one by one for presence of FD_CLOEXEC. */
        for (i = 0; i < OPEN_MAX; i++) {
            if ( (bitmap >> i) & 01) (void) close_fd(rfp, i);
        }
    }
}
```

```
# Makefile for inet.

# Directories
g = generic

# Programs, flags, and libraries
CC = cc
CPPFLAGS = -I. -D_MINIX
CFLAGS = $(OPT) $(CPPFLAGS)
LDFLAGS =
LIBS = -lsys -lsysutil

.c.o:
    $(CC) $(CFLAGS) -o $@ -c $<

OBJ = buf.o clock.o inet.o inet_config.o \
    mnx_eth.o mq.o qp.o sr.o stacktrace.o \
    $g/udp.o $g/arp.o $g/eth.o $g/event.o \
    $g/icmp.o $g/io.o $g/ip.o $g/ip_ioctl.o \
    $g/ip_lib.o $g/ip_read.o $g/ip_write.o \
    $g/ipr.o $g/rand256.o $g/tcp.o $g/tcp_lib.o \
    $g/tcp_recv.o $g/tcp_send.o $g/ip_eth.o \
    $g/ip_ps.o $g/psip.o \
    minix3/queryparam.o sha2.o

all: inet

inet: $(OBJ)
    $(CC) -o $@ $(LDFLAGS) $(OBJ) version.c $(LIBS)

install: inet
    install -c $? /usr/sbin/inet

clean:
    rm -f $(OBJ) inet *.bak

depend:
    /usr/bin/mkdep "$(CC) -E $(CPPFLAGS)" *.c generic/*.c > .depend

# Include generated dependencies.
include .depend

#
# $PchId: Makefile.mnx3,v 1.1 2005/06/28 14:28:45 philip Exp $
#
```

```

/*
This file contains routines for buffer management.

Copyright 1995 Philip Homburg
*/

#define BUF_IMPLEMENTATION      1          /* Avoid some macros */

#include "inet.h"

#include <stdlib.h>
#include <string.h>

#include "generic/assert.h"
#include "generic/buf.h"
#include "generic/type.h"

THIS_FILE

#ifndef BUF_USEMALLOC
#define BUF_USEMALLOC      0
#endif

#ifndef BUF512_NR
#define BUF512_NR          512
#endif
#ifndef BUF2K_NR
#define BUF2K_NR           0
#endif
#ifndef BUF32K_NR
#define BUF32K_NR          0
#endif

#define ACC_NR              ((BUF512_NR+BUF2K_NR+BUF32K_NR)*3)
#define CLIENT_NR          7

#define DECLARE_TYPE(Tag, Type, Size)
typedef struct Tag
{
    buf_t buf_header;
    char buf_data[Size];
} Type

#if BUF_USEMALLOC
#define DECLARE_STORAGE(Type, Ident, Nitems)
PRIVATE Type *Ident

#define ALLOC_STORAGE(Ident, Nitems, Label)
do
{
    printf("buf.c: malloc %d %s\n", Nitems, Label);
    Ident= malloc(sizeof(*Ident) * Nitems);
    if (!Ident)
        ip_panic(( "unable to alloc %s", Label ));
} while(0)
#else
#define DECLARE_STORAGE(Type, Ident, Nitems)
PRIVATE Type Ident[Nitems]

#define ALLOC_STORAGE(Ident, Nitems, Label)
(void)0
#endif

#if BUF512_NR
DECLARE_TYPE(buf512, buf512_t, 512);
PRIVATE acc_t *buf512_freelist;
DECLARE_STORAGE(buf512_t, buffers512, BUF512_NR);
FORWARD void bf_512free ARGS(( acc_t *acc ));
#endif
#if BUF2K_NR
DECLARE_TYPE(buf2K, buf2K_t, (2*1024));
PRIVATE acc_t *buf2K_freelist;
DECLARE_STORAGE(buf2K_t, buffers2K, BUF2K_NR);
FORWARD void bf_2Kfree ARGS(( acc_t *acc ));

```

```
#endif
#if BUF32K_NR
DECLARE_TYPE(buf32K, buf32K_t, (32*1024));
PRIVATE acc_t *buf32K_freelist;
DECLARE_STORAGE(buf32K_t, buffers32K, BUF32K_NR);
FORWARD void bf_32Kfree ARGS(( acc_t *acc ));
#endif

PRIVATE acc_t *acc_freelist;
DECLARE_STORAGE(acc_t, accessors, ACC_NR);

PRIVATE bf_freereq_t freereq[CLIENT_NR];
PRIVATE size_t bf_buf_gran;

PUBLIC size_t bf_free_bufsize;
PUBLIC acc_t *bf_temporary_acc;
PUBLIC acc_t *bf_linkcheck_acc;

#ifdef BUF_CONSISTENCY_CHECK
int inet_buf_debug;
unsigned buf_generation;
PRIVATE bf_checkreq_t checkreq[CLIENT_NR];
#endif

#ifdef BUF_TRACK_ALLOC_FREE
FORWARD acc_t *bf_small_memreq ARGS(( size_t size ));
#else
FORWARD acc_t *_bf_small_memreq ARGS(( char *clnt_file, int clnt_line,
                                     size_t size ));
#define bf_small_memreq(a) _bf_small_memreq(clnt_file, clnt_line, a)
#endif
FORWARD void free_accs ARGS(( void ));
#ifdef BUF_CONSISTENCY_CHECK
FORWARD void count_free_bufs ARGS(( acc_t *list ));
FORWARD int report_buffer ARGS(( buf_t *buf, char *label, int i ));
#endif

PUBLIC void bf_init()
{
    int i;
    size_t buf_s;
    acc_t *acc;

    bf_buf_gran= BUF_S;
    buf_s= 0;

    for (i=0;i<CLIENT_NR;i++)
        freereq[i]=0;
#ifdef BUF_CONSISTENCY_CHECK
    for (i=0;i<CLIENT_NR;i++)
        checkreq[i]=0;
#endif

#if BUF512_NR
    ALLOC_STORAGE(buffers512, BUF512_NR, "512B-buffers");
#endif
#if BUF2K_NR
    ALLOC_STORAGE(buffers2K, BUF2K_NR, "2K-buffers");
#endif
#if BUF32K_NR
    ALLOC_STORAGE(buffers32K, BUF32K_NR, "32K-buffers");
#endif

    ALLOC_STORAGE(accessors, ACC_NR, "accs");

    acc_freelist= NULL;
    for (i=0;i<ACC_NR;i++)
    {
        memset(&accessors[i], '\0', sizeof(accessors[i]));

        accessors[i].acc_linkC= 0;
        accessors[i].acc_next= acc_freelist;
        acc_freelist= &accessors[i];
    }
}
```

```

#define INIT_BUFFERS(Ident, Nitems, Freelist, Freefunc)
do
{
    Freelist= NULL;
    for (i=0;i<Nitems;i++)
    {
        acc= acc_freelist;
        if (!acc)
            ip_panic(( "fewer accessors than buffers" )); \
        acc_freelist= acc->acc_next;
        acc->acc_linkC= 0;

        memset(&Ident[i], '\0', sizeof(Ident[i]));
        Ident[i].buf_header.buf_linkC= 0;
        Ident[i].buf_header.buf_free= Freefunc;
        Ident[i].buf_header.buf_size=
            sizeof(Ident[i].buf_data);
        Ident[i].buf_header.buf_data_p=
            Ident[i].buf_data;

        acc->acc_buffer= &Ident[i].buf_header;
        acc->acc_next= Freelist;
        Freelist= acc;
    }
    if (sizeof(Ident[0].buf_data) < bf_buf_gran)
        bf_buf_gran= sizeof(Ident[0].buf_data);
    if (sizeof(Ident[0].buf_data) > buf_s)
        buf_s= sizeof(Ident[0].buf_data);
} while(0)

#if BUF512_NR
    INIT_BUFFERS(buffers512, BUF512_NR, buf512_freelist, bf_512free);
#endif
#if BUF2K_NR
    INIT_BUFFERS(buffers2K, BUF2K_NR, buf2K_freelist, bf_2Kfree);
#endif
#if BUF32K_NR
    INIT_BUFFERS(buffers32K, BUF32K_NR, buf32K_freelist, bf_32Kfree);
#endif

#undef INIT_BUFFERS

    assert (buf_s == BUF_S);
}

#ifdef BUF_CONSISTENCY_CHECK
PUBLIC void bf_logon(func)
bf_freereq_t func;
#else
PUBLIC void bf_logon(func, checkfunc)
bf_freereq_t func;
bf_checkreq_t checkfunc;
#endif
{
    int i;

    for (i=0;i<CLIENT_NR;i++)
        if (!freereq[i])
        {
            freereq[i]=func;
#ifdef BUF_CONSISTENCY_CHECK
            checkreq[i]= checkfunc;
#endif
        }

    return;
}

ip_panic(( "buf.c: too many clients" ));
}

/*
bf_memreq
*/

#ifdef BUF_TRACK_ALLOC_FREE

```

```

PUBLIC acc_t *bf_memreq(size)
#else
PUBLIC acc_t *_bf_memreq(clnt_file, clnt_line, size)
char *clnt_file;
int clnt_line;
#endif
size_t size;
{
    acc_t *head, *tail, *new_acc;
    buf_t *buf;
    int i,j;
    size_t count;

    assert (size>0);

    head= NULL;
    tail= NULL;
    while (size)
    {
        new_acc= NULL;

        /* Note the tricky dangling else... */
#define ALLOC_BUF(Freelist, BuFSIZE)
        if (Freelist && (BuFSIZE == BUF_S || size <= BuFSIZE))
        {
            new_acc= Freelist;
            Freelist= new_acc->acc_next;

            assert(new_acc->acc_linkC == 0);
            new_acc->acc_linkC= 1;
            buf= new_acc->acc_buffer;
            assert(buf->buf_linkC == 0);
            buf->buf_linkC= 1;
        }
        else

        /* Sort attempts by buffer size */
#if BUF512_NR
        ALLOC_BUF(buf512_freelist, 512)
#endif
#if BUF2K_NR
        ALLOC_BUF(buf2K_freelist, 2*1024)
#endif
#if BUF32K_NR
        ALLOC_BUF(buf32K_freelist, 32*1024)
#endif
#undef ALLOC_BUF
        {
            DBLOCK(2, printf("freeing buffers\n"));

            bf_free_bufsize= 0;
            for (i=0; bf_free_bufsize<size && i<MAX_BUFREQ_PRI;
                i++)
            {
                for (j=0; j<CLIENT_NR; j++)
                {
                    if (freereq[j])
                        (*freereq[j])(i);
                }
            }
#if DEBUG && 0
            { acc_t *acc;
              j= 0; for(acc= buf512_freelist; acc; acc= acc->acc_next) j++;
              printf("# of free 512-bytes buffer is now %d\n", j); }
#endif
            }
#if DEBUG && 0
            { printf("last level was level %d\n", i-1); }
#endif
            if (bf_free_bufsize<size)
                ip_panic(( "not enough buffers freed" ));

            continue;
        }
    }
}

```



```

#ifdef BUF_TRACK_ALLOC_FREE
    new_acc->acc_alloc_file= clnt_file;
    new_acc->acc_alloc_line= clnt_line;
    buf->buf_alloc_file= clnt_file;
    buf->buf_alloc_line= clnt_line;
#endif

    if (!head)
        head= new_acc;
    else
        tail->acc_next= new_acc;
    tail= new_acc;

    count= tail->acc_buffer->buf_size;
    if (count > size)
        count= size;

    tail->acc_offset= 0;
    tail->acc_length= count;
    size -= count;
}
tail->acc_next= NULL;

return head;
}

/*
bf_small_memreq
*/

#ifndef BUF_TRACK_ALLOC_FREE
PRIVATE acc_t *bf_small_memreq(size)
#else
PRIVATE acc_t *_bf_small_memreq(clnt_file, clnt_line, size)
char *clnt_file;
int clnt_line;
#endif
size_t size;
{
    return bf_memreq(size);
}

#ifndef BUF_TRACK_ALLOC_FREE
PUBLIC void bf_afree(acc)
#else
PUBLIC void _bf_afree(clnt_file, clnt_line, acc)
char *clnt_file;
int clnt_line;
#endif
acc_t *acc;
{
    acc_t *next_acc;
    buf_t *buf;

    while (acc)
    {
#if defined(bf_afree)
        DIFBLOCK(1, (acc->acc_linkC <= 0),
            printf("clnt_file= %s, clnt_line= %d\n",
                clnt_file, clnt_line));
#endif

        assert (acc->acc_linkC>0);
        if (--acc->acc_linkC > 0)
            break;

#ifdef BUF_TRACK_ALLOC_FREE
        acc->acc_free_file= clnt_file;
        acc->acc_free_line= clnt_line;
#endif

        buf= acc->acc_buffer;
        assert (buf);

#if defined(bf_afree)
        DIFBLOCK(1, (buf->buf_linkC == 0),

```

```

        printf("clnt_file= %s, clnt_line= %d\n",
               clnt_file, clnt_line));
#endif
    assert(buf->buf_linkC > 0);
    if (--buf->buf_linkC > 0)
    {
        acc->acc_buffer= NULL;
        next_acc= acc->acc_next;
        acc->acc_next= acc_freelist;
        acc_freelist= acc;
#ifdef BUF_CONSISTENCY_CHECK
        if (inet_buf_debug)
        {
            acc->acc_offset= 0xdeadbeaf;
            acc->acc_length= 0xdeadbeaf;
            acc->acc_buffer= (buf_t *)0xdeadbeaf;
            acc->acc_ext_link= (acc_t *)0xdeadbeaf;
        }
#endif
        acc= next_acc;
        continue;
    }

    bf_free_bufsize += buf->buf_size;
#ifdef BUF_TRACK_ALLOC_FREE
    buf->buf_free_file= clnt_file;
    buf->buf_free_line= clnt_line;
#endif
    next_acc= acc->acc_next;
    buf->buf_free(acc);
    acc= next_acc;
    continue;
}

#ifdef BUF_TRACK_ALLOC_FREE
PUBLIC acc_t *bf_dupacc(acc_ptr)
#else
PUBLIC acc_t *_bf_dupacc(clnt_file, clnt_line, acc_ptr)
char *clnt_file;
int clnt_line;
#endif
register acc_t *acc_ptr;
{
    register acc_t *new_acc;

    if (!acc_freelist)
    {
        free_accs();
        if (!acc_freelist)
            ip_panic(("buf.c: out of accessors" ));
    }
    new_acc= acc_freelist;
    acc_freelist= new_acc->acc_next;

    *new_acc= *acc_ptr;
    if (acc_ptr->acc_next)
        acc_ptr->acc_next->acc_linkC++;
    if (acc_ptr->acc_buffer)
        acc_ptr->acc_buffer->buf_linkC++;
    new_acc->acc_linkC= 1;
#ifdef BUF_TRACK_ALLOC_FREE
    new_acc->acc_alloc_file= clnt_file;
    new_acc->acc_alloc_line= clnt_line;
#endif
    return new_acc;
}

PUBLIC size_t bf_bufsize(acc_ptr)
register acc_t *acc_ptr;
{
    register size_t size;

    assert(acc_ptr);

```

```

        size=0;

        while (acc_ptr)
        {
assert(acc_ptr >= accessors && acc_ptr <= &accessors[ACC_NR-1]);
            size += acc_ptr->acc_length;
            acc_ptr= acc_ptr->acc_next;
        }
        return size;
    }

#ifdef BUF_TRACK_ALLOC_FREE
PUBLIC acc_t *bf_packIffLess(pack, min_len)
#else
PUBLIC acc_t *_bf_packIffLess(clnt_file, clnt_line, pack, min_len)
char *clnt_file;
int clnt_line;
#endif
acc_t *pack;
int min_len;
{
    if (!pack || pack->acc_length >= min_len)
        return pack;

#ifdef DEBUG
#ifdef bf_packIffLess
    { where(); printf("calling bf_pack because of %s %d: %d\n", bf_pack_file,
        bf_pack_line, min_len); }
#endif
#endif
    return bf_pack(pack);
}

#ifdef BUF_TRACK_ALLOC_FREE
PUBLIC acc_t *bf_pack(old_acc)
#else
PUBLIC acc_t *_bf_pack(clnt_file, clnt_line, old_acc)
char *clnt_file;
int clnt_line;
#endif
acc_t *old_acc;
{
    acc_t *new_acc, *acc_ptr_old, *acc_ptr_new;
    size_t size, offset_old, offset_new, block_size, block_size_old;

    /* Check if old acc is good enough. */
    if (!old_acc || (!old_acc->acc_next && old_acc->acc_linkC == 1 &&
        old_acc->acc_buffer->buf_linkC == 1))
    {
        return old_acc;
    }

    size= bf_bufsize(old_acc);
    assert(size > 0);
    new_acc= bf_memreq(size);
    acc_ptr_old= old_acc;
    acc_ptr_new= new_acc;
    offset_old= 0;
    offset_new= 0;
    while (size)
    {
        assert (acc_ptr_old);
        if (offset_old == acc_ptr_old->acc_length)
        {
            offset_old= 0;
            acc_ptr_old= acc_ptr_old->acc_next;
            continue;
        }
        assert (offset_old < acc_ptr_old->acc_length);
        block_size_old= acc_ptr_old->acc_length - offset_old;
        assert (acc_ptr_new);
        if (offset_new == acc_ptr_new->acc_length)
        {

```

```

        offset_new= 0;
        acc_ptr_new= acc_ptr_new->acc_next;
        continue;
    }
    assert (offset_new < acc_ptr_new->acc_length);
    block_size= acc_ptr_new->acc_length - offset_new;
    if (block_size > block_size_old)
        block_size= block_size_old;
    memcpy(ptr2acc_data(acc_ptr_new)+offset_new,
           ptr2acc_data(acc_ptr_old)+offset_old, block_size);
    offset_new += block_size;
    offset_old += block_size;
    size -= block_size;
}
bf_afree(old_acc);
return new_acc;
}

#ifdef BUF_TRACK_ALLOC_FREE
PUBLIC acc_t *bf_cut (data, offset, length)
#else
PUBLIC acc_t *_bf_cut (clnt_file, clnt_line, data, offset, length)
char *clnt_file;
int clnt_line;
#endif
register acc_t *data;
register unsigned offset;
register unsigned length;
{
    register acc_t *head, *tail;

    if (!data && !offset && !length)
        return NULL;
#ifdef BUF_TRACK_ALLOC_FREE
    assert(data ||
           (printf("from %s, %d: %u, %u\n",
                    clnt_file, clnt_line, offset, length), 0));
#else
    assert(data);
#endif

    assert(data);

    if (!length)
    {
        head= bf_dupacc(data);
        bf_afree(head->acc_next);
        head->acc_next= NULL;
        head->acc_length= 0;
        return head;
    }
    while (data && offset>=data->acc_length)
    {
        offset -= data->acc_length;
        data= data->acc_next;
    }

    assert (data);

    head= bf_dupacc(data);
    bf_afree(head->acc_next);
    head->acc_next= NULL;
    head->acc_offset += offset;
    head->acc_length -= offset;
    if (length >= head->acc_length)
        length -= head->acc_length;
    else
    {
        head->acc_length= length;
        length= 0;
    }
    tail= head;
    data= data->acc_next;
    while (data && length && length>=data->acc_length)

```

```

        {
            tail->acc_next= bf_dupacc(data);
            tail= tail->acc_next;
            bf_afree(tail->acc_next);
            tail->acc_next= NULL;
            data= data->acc_next;
            length -= tail->acc_length;
        }
        if (length)
        {
#ifdef bf_cut
            assert (data ||
                (printf("bf_cut called from %s:%d\n",
                    clnt_file, clnt_line), 0));
#else
            assert (data);
#endif

            tail->acc_next= bf_dupacc(data);
            tail= tail->acc_next;
            bf_afree(tail->acc_next);
            tail->acc_next= NULL;
            tail->acc_length= length;
        }
        return head;
    }

#ifndef BUF_TRACK_ALLOC_FREE
    PUBLIC acc_t *bf_delhead (data, offset)
#else
    PUBLIC acc_t *_bf_delhead (clnt_file, clnt_line, data, offset)
    char *clnt_file;
    int clnt_line;
#endif
    register acc_t *data;
    register unsigned offset;
    {
        acc_t *new_acc;

        assert(data);

        /* Find the acc we need to modify. */
        new_acc= data;
        while(offset >= new_acc->acc_length)
        {
            offset -= new_acc->acc_length;
            new_acc= new_acc->acc_next;
#ifdef BUF_TRACK_ALLOC_FREE
                assert(new_acc || (printf("called from %s, %d\n",
                    clnt_file, clnt_line), 0));
#else
                assert(new_acc);
#endif
        }

        /* Discard the old acc(s) */
        if (new_acc != data)
        {
            new_acc->acc_linkC++;
            bf_afree(data);
            data= new_acc;
        }

        /* Make sure that acc_linkC == 1 */
        if (data->acc_linkC != 1)
        {
            new_acc= bf_dupacc(data);
            bf_afree(data);
            data= new_acc;
        }

        /* Delete the last bit by modifying acc_offset and acc_length */
        data->acc_offset += offset;
        data->acc_length -= offset;
        return data;
    }

```

```
}

/*
bf_append
*/

#ifdef BUF_TRACK_ALLOC_FREE
PUBLIC acc_t *bf_append(data_first, data_second)
#else
PUBLIC acc_t *_bf_append(clnt_file, clnt_line, data_first, data_second)
char *clnt_file;
int clnt_line;
#endif
acc_t *data_first;
acc_t *data_second;
{
    acc_t *head, *tail, *new_acc, *acc_ptr_new, tmp_acc, *curr;
    char *src_ptr, *dst_ptr;
    size_t size, offset_old, offset_new, block_size_old, block_size;

    if (!data_first)
        return data_second;
    if (!data_second)
        return data_first;

    head= NULL;
    tail= NULL;
    while (data_first)
    {
        if (data_first->acc_linkC == 1)
            curr= data_first;
        else
        {
            curr= bf_dupacc(data_first);
            assert (curr->acc_linkC == 1);
            bf_afree(data_first);
        }
        data_first= curr->acc_next;
        if (!curr->acc_length)
        {
            curr->acc_next= NULL;
            bf_afree(curr);
            continue;
        }
        if (!head)
            head= curr;
        else
            tail->acc_next= curr;
        tail= curr;
    }
    if (!head)
        return data_second;
    tail->acc_next= NULL;

    while (data_second && !data_second->acc_length)
    {
        curr= data_second;
        data_second= data_second->acc_next;
        if (data_second)
            data_second->acc_linkC++;
        bf_afree(curr);
    }
    if (!data_second)
        return head;

    if (tail->acc_length + data_second->acc_length >
        tail->acc_buffer->buf_size)
    {
        tail->acc_next= data_second;
        return head;
    }

    if (tail->acc_buffer->buf_size == bf_buf_gran &&
        tail->acc_buffer->buf_linkC == 1)
```

```
{
    if (tail->acc_offset)
    {
        memmove(tail->acc_buffer->buf_data_p,
                ptr2acc_data(tail), tail->acc_length);
        tail->acc_offset= 0;
    }
    dst_ptr= ptr2acc_data(tail) + tail->acc_length;
    src_ptr= ptr2acc_data(data_second);
    memcpy(dst_ptr, src_ptr, data_second->acc_length);
    tail->acc_length += data_second->acc_length;
    tail->acc_next= data_second->acc_next;
    if (data_second->acc_next)
        data_second->acc_next->acc_linkC++;
    bf_afree(data_second);
    return head;
}

new_acc= bf_small_memreq(tail->acc_length+data_second->acc_length);
acc_ptr_new= new_acc;
offset_old= 0;
offset_new= 0;
size= tail->acc_length;
while (size)
{
assert (acc_ptr_new);
    if (offset_new == acc_ptr_new->acc_length)
    {
        offset_new= 0;
        acc_ptr_new= acc_ptr_new->acc_next;
        continue;
    }
assert (offset_new < acc_ptr_new->acc_length);
assert (offset_old < tail->acc_length);
    block_size_old= tail->acc_length - offset_old;
    block_size= acc_ptr_new->acc_length - offset_new;
    if (block_size > block_size_old)
        block_size= block_size_old;
    memcpy(ptr2acc_data(acc_ptr_new)+offset_new,
           ptr2acc_data(tail)+offset_old, block_size);
    offset_new += block_size;
    offset_old += block_size;
    size -= block_size;
}
offset_old= 0;
size= data_second->acc_length;
while (size)
{
assert (acc_ptr_new);
    if (offset_new == acc_ptr_new->acc_length)
    {
        offset_new= 0;
        acc_ptr_new= acc_ptr_new->acc_next;
        continue;
    }
assert (offset_new < acc_ptr_new->acc_length);
assert (offset_old < data_second->acc_length);
    block_size_old= data_second->acc_length - offset_old;
    block_size= acc_ptr_new->acc_length - offset_new;
    if (block_size > block_size_old)
        block_size= block_size_old;
    memcpy(ptr2acc_data(acc_ptr_new)+offset_new,
           ptr2acc_data(data_second)+offset_old, block_size);
    offset_new += block_size;
    offset_old += block_size;
    size -= block_size;
}
tmp_acc= *tail;
*tail= *new_acc;
*new_acc= tmp_acc;

bf_afree(new_acc);
while (tail->acc_next)
    tail= tail->acc_next;
```

```

        tail->acc_next= data_second->acc_next;
        if (data_second->acc_next)
            data_second->acc_next->acc_linkC++;
        bf_afree(data_second);
        return head;
    }

#if BUF512_NR
PRIVATE void bf_512free(acc)
acc_t *acc;
{
#ifdef BUF_CONSISTENCY_CHECK
    if (inet_buf_debug)
        memset(acc->acc_buffer->buf_data_p, 0xa5, 512);
#endif
    acc->acc_next= buf512_freelist;
    buf512_freelist= acc;
}
#endif
#if BUF2K_NR
PRIVATE void bf_2Kfree(acc)
acc_t *acc;
{
#ifdef BUF_CONSISTENCY_CHECK
    if (inet_buf_debug)
        memset(acc->acc_buffer->buf_data_p, 0xa5, 2*1024);
#endif
    acc->acc_next= buf2K_freelist;
    buf2K_freelist= acc;
}
#endif
#if BUF32K_NR
PRIVATE void bf_32Kfree(acc)
acc_t *acc;
{
#ifdef BUF_CONSISTENCY_CHECK
    if (inet_buf_debug)
        memset(acc->acc_buffer->buf_data_p, 0xa5, 32*1024);
#endif
    acc->acc_next= buf32K_freelist;
    buf32K_freelist= acc;
}
#endif

#ifdef BUF_CONSISTENCY_CHECK
PUBLIC int bf_consistency_check()
{
    acc_t *acc;
    int silent;
    int error;
    int i;

    buf_generation++;

    for (i=0; i<CLIENT_NR; i++)
    {
        if (checkreq[i])
            (*checkreq[i])();
    }

    /* Add information about free accessors */
    for(acc= acc_freelist; acc; acc= acc->acc_next)
    {
        if (acc->acc_generation == buf_generation-1)
        {
            acc->acc_generation= buf_generation;
            acc->acc_check_linkC= 0;
        }
        else
        {
            assert(acc->acc_generation == buf_generation &&
                acc->acc_check_linkC > 0);
            acc->acc_check_linkC= -acc->acc_check_linkC;
        }
    }
}
#endif

```



```

    }
}

#if BUF512_NR
    count_free_bufs(buf512_freelist);
#endif
#if BUF2K_NR
    count_free_bufs(buf2K_freelist);
#endif
#if BUF32K_NR
    count_free_bufs(buf32K_freelist);
#endif

    error= 0;

    /* Report about accessors */
    silent= 0;
    for (i=0, acc= accessors; i<ACC_NR; i++, acc++)
    {
        if (acc->acc_generation != buf_generation)
        {
            error++;
            assert(acc->acc_generation == buf_generation-1);
            acc->acc_generation= buf_generation;
            if (!silent)
            {
                printf(
"acc[%d] (%p) has been lost with count %d, last allocated at %s, %d\n",
i, acc, acc->acc_linkC, acc->acc_alloc_file, acc->acc_alloc_line);
#if 0
                    silent= 1;
#endif

                }
                continue;
            }
            if (acc->acc_check_linkC == acc->acc_linkC)
                continue;
            error++;
            if (acc->acc_check_linkC < 0)
            {
                if (!silent)
                {
                    printf(
"acc[%d] is freed but still in use, allocated at %s, %d, freed at %s, %d\n",
i, acc->acc_alloc_file, acc->acc_alloc_line,
acc->acc_free_file, acc->acc_free_line);
                }
                acc->acc_check_linkC= -acc->acc_check_linkC;
                if (acc->acc_check_linkC == acc->acc_linkC)
                {
                    silent= 1;
                    continue;
                }
            }
            if (!silent)
            {
                printf(
"# of tracked links (%d) for acc[%d] don't match with stored link count %d\n",
acc->acc_check_linkC, i, acc->acc_linkC);
                printf("acc[%d] was allocated at %s, %d\n",
i, acc->acc_alloc_file, acc->acc_alloc_line);
                silent=1;
            }
        }
    }

    /* Report about buffers */
#if BUF512_NR
    {
        for (i= 0; i<BUF512_NR; i++)
        {
            error |= report_buffer(&buffers512[i].buf_header,
"512-buffer", i);
        }
    }

```

```

#endif
#if BUF2K_NR
{
    for (i= 0; i<BUF2K_NR; i++)
    {
        error |= report_buffer(&buffers2K[i].buf_header,
                               "2K-buffer", i);
    }
}
#endif
#if BUF32K_NR
{
    for (i= 0; i<BUF32K_NR; i++)
    {
        error |= report_buffer(&buffers32K[i].buf_header,
                               "32K-buffer", i);
    }
}
#endif

return !error;
}

PRIVATE void count_free_bufs(list)
acc_t *list;
{
    acc_t *acc;
    buf_t *buf;

    for(acc= list; acc; acc= acc->acc_next)
    {
        if (acc->acc_generation != buf_generation-1)
        {
            assert(acc->acc_generation == buf_generation &&
                   acc->acc_check_linkC > 0);
            acc->acc_check_linkC= -acc->acc_check_linkC;
            continue;
        }
        acc->acc_generation= buf_generation;
        acc->acc_check_linkC= 0;

        buf= acc->acc_buffer;
        if (buf->buf_generation == buf_generation-1)
        {
            buf->buf_generation= buf_generation;
            buf->buf_check_linkC= 0;
            continue;
        }
        assert(buf->buf_generation == buf_generation &&
               buf->buf_check_linkC > 0);
        buf->buf_check_linkC= -buf->buf_check_linkC;
    }
}

PRIVATE int report_buffer(buf, label, i)
buf_t *buf;
char *label;
int i;
{
    if (buf->buf_generation != buf_generation)
    {
        assert(buf->buf_generation == buf_generation-1);
        buf->buf_generation= buf_generation;
        printf(
"%s[%d] (%p) has been lost with count %d, last allocated at %s, %d\n" ,
                label, i, buf,
                buf->buf_linkC, buf->buf_alloc_file,
                buf->buf_alloc_line);

        return 1;
    }
    if (buf->buf_check_linkC == buf->buf_linkC)
        return 0;
    if (buf->buf_check_linkC < 0)
    {

```

```
        printf(
"%s[%d] is freed but still in use, allocated at %s, %d, freed at %s, %d\n",
        label, i, buf->buf_alloc_file, buf->buf_alloc_line,
        buf->buf_free_file, buf->buf_free_line);
        buf->buf_check_linkC= -buf->buf_check_linkC;
        if (buf->buf_check_linkC == buf->buf_linkC)
            return 1;
    }
    printf(
"# of tracked links (%d) for %s[%d] don't match with stored link count %d\n",
        buf->buf_check_linkC, label, i, buf->buf_linkC);
    printf("%s[%d] was allocated at %s, %d\n",
        label, i, buf->buf_alloc_file, buf->buf_alloc_line);
    return 1;
}

PUBLIC void bf_check_acc(acc)
acc_t *acc;
{
    buf_t *buf;

    while(acc != NULL)
    {
        if (acc->acc_generation == buf_generation)
        {
            assert(acc->acc_check_linkC > 0);
            acc->acc_check_linkC++;
            return;
        }
        assert(acc->acc_generation == buf_generation-1);
        acc->acc_generation= buf_generation;
        acc->acc_check_linkC= 1;

        buf= acc->acc_buffer;
        if (buf->buf_generation == buf_generation)
        {
            assert(buf->buf_check_linkC > 0);
            buf->buf_check_linkC++;
        }
        else
        {
            assert(buf->buf_generation == buf_generation-1);
            buf->buf_generation= buf_generation;
            buf->buf_check_linkC= 1;
        }

        acc= acc->acc_next;
    }
}

PUBLIC void _bf_mark_lacc(clnt_file, clnt_line, acc)
char *clnt_file;
int clnt_line;
acc_t *acc;
{
    acc->acc_alloc_file= clnt_file;
    acc->acc_alloc_line= clnt_line;
}

PUBLIC void _bf_mark_acc(clnt_file, clnt_line, acc)
char *clnt_file;
int clnt_line;
acc_t *acc;
{
    buf_t *buf;

    for (; acc; acc= acc->acc_next)
    {
        acc->acc_alloc_file= clnt_file;
        acc->acc_alloc_line= clnt_line;
        buf= acc->acc_buffer;
        buf->buf_alloc_file= clnt_file;
        buf->buf_alloc_line= clnt_line;
    }
}
```

```

}
#endif

PUBLIC int bf_linkcheck(acc)
acc_t *acc;
{
    int i;

    buf_t *buffer;
    for (i= 0; i<ACC_NR && acc; i++, acc= acc->acc_next)
    {
        if (acc->acc_linkC <= 0)
        {
            printf("wrong acc_linkC (%d) for acc %p\n",
                acc->acc_linkC, acc);
            return 0;
        }
        if (acc->acc_offset < 0)
        {
            printf("wrong acc_offset (%d) for acc %p\n",
                acc->acc_offset, acc);
            return 0;
        }
        if (acc->acc_length < 0)
        {
            printf("wrong acc_length (%d) for acc %p\n",
                acc->acc_length, acc);
            return 0;
        }
        buffer= acc->acc_buffer;
        if (buffer == NULL)
        {
            printf("no buffer for acc %p\n", acc);
            return 0;
        }
        if (buffer->buf_linkC <= 0)
        {
            printf(
                "wrong buf_linkC (%d) for buffer %p, from acc %p\n",
                buffer->buf_linkC, buffer, acc);
            return 0;
        }
        if (acc->acc_offset + acc->acc_length > buffer->buf_size)
        {
            printf("%d + %d > %d for buffer %p, and acc %p\n",
                acc->acc_offset, acc->acc_length,
                buffer->buf_size, buffer, acc);
            return 0;
        }
    }
    if (acc != NULL)
    {
        printf("loop\n");
        return 0;
    }
    return 1;
}

PRIVATE void free_accs()
{
    int i, j;

    DBLOCK(1, printf("free_accs\n"));

    assert(bf_linkcheck(bf_linkcheck_acc));
    for (i=0; !acc_freelist && i<MAX_BUFREQ_PRI; i++)
    {
        for (j=0; j<CLIENT_NR; j++)
        {
            bf_free_bufsize= 0;
            if (freereq[j])
            {
                (*freereq[j])(i);
                assert(bf_linkcheck(bf_linkcheck_acc) ||

```

```

                                (printf("just called %p\n",
                                freereq[i]),0));
                                }
                                }
}
#if DEBUG
    printf("last level was level %d\n", i-1);
#endif
}

#ifndef BUF_TRACK_ALLOC_FREE
PUBLIC acc_t *bf_align(acc, size, alignment)
#else
PUBLIC acc_t *_bf_align(clnt_file, clnt_line, acc, size, alignment)
char *clnt_file;
int clnt_line;
#endif
acc_t *acc;
size_t size;
size_t alignment;
{
    char *ptr;
    size_t buf_size;
    acc_t *head, *tail;

    /* Fast check if the buffer is aligned already. */
    if (acc->acc_length >= size)
    {
        ptr= ptr2acc_data(acc);
        if (((unsigned)ptr & (alignment-1)) == 0)
            return acc;
    }
    buf_size= bf_bufsize(acc);
#ifdef bf_align
    assert((size != 0 && buf_size != 0) ||
        (printf("bf_align(..., %d, %d) from %s, %d\n",
            size, alignment, clnt_file, clnt_line),0));
#else
    assert(size != 0 && buf_size != 0);
#endif
    if (buf_size <= size)
    {
        acc= bf_pack(acc);
        return acc;
    }
    head= bf_cut(acc, 0, size);
    tail= bf_cut(acc, size, buf_size-size);
    bf_afree(acc);
    head= bf_pack(head);
    assert(head->acc_next == NULL);
    head->acc_next= tail;
    return head;
}

#if 0
int chk_acc(acc)
acc_t *acc;
{
    int acc_nr;

    if (!acc)
        return 1;
    if (acc < accessors || acc >= &accessors[ACC_NR])
        return 0;
    acc_nr= acc-accessors;
    return acc == &accessors[acc_nr];
}
#endif

/*
 * $PchId: buf.c,v 1.19 2003/09/10 08:54:23 philip Exp $
 */

```

```

/*
clock.c

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "proto.h"
#include "generic/assert.h"
#include "generic/buf.h"
#include "generic/clock.h"
#include "generic/type.h"

THIS_FILE

PUBLIC int clk_call_expire;

PRIVATE time_t curr_time;
PRIVATE time_t prev_time;
PRIVATE timer_t *timer_chain;
PRIVATE time_t next_timeout;
#ifdef __minix_vmd
PRIVATE int clk_tasknr= ANY;
#endif

FORWARD _PROTOTYPE( void clk_fast_release, (timer_t *timer) );
FORWARD _PROTOTYPE( void set_timer, (void) );

PUBLIC void clk_init()
{
    int r;

    clk_call_expire= 0;
    curr_time= 0;
    prev_time= 0;
    next_timeout= 0;
    timer_chain= 0;

#ifdef __minix_vmd
    r= sys_findproc(CLOCK_NAME, &clk_tasknr, 0);
    if (r != OK)
        ip_panic(( "unable to find clock task: %d\n", r ));
#endif
}

PUBLIC time_t get_time()
{
    if (!curr_time)
    {
#ifdef __minix_vmd
        static message mess;

        mess.m_type= GET_UPTIME;
        if (sendrec (clk_tasknr, &mess) < 0)
            ip_panic(( "unable to sendrec" ));
        if (mess.m_type != OK)
            ip_panic(( "can't read clock" ));
        curr_time= mess.NEW_TIME;
#else /* Minix 3 */
        int s;
        if ((s=getuptime(&curr_time)) != OK)
            ip_panic(( "can't read clock" ));
#endif
        assert(curr_time >= prev_time);
    }
    return curr_time;
}

PUBLIC void set_time (tim)
time_t tim;
{
    if (!curr_time && tim >= prev_time)
    {
        /* Some code assumes that no time elapses while it is

```

```
        * running.
        */
        curr_time= tim;
    }
    else if (!curr_time)
    {
        DBLOCK(0x20, printf("set_time: new time %ld < prev_time %ld\n",
            tim, prev_time));
    }
}

PUBLIC void reset_time()
{
    prev_time= curr_time;
    curr_time= 0;
}

PUBLIC void clk_timer(timer, timeout, func, fd)
timer_t *timer;
time_t timeout;
timer_func_t func;
int fd;
{
    timer_t *timer_index;

    if (timer->tim_active)
        clk_fast_release(timer);
    assert(!timer->tim_active);

    timer->tim_next= 0;
    timer->tim_func= func;
    timer->tim_ref= fd;
    timer->tim_time= timeout;
    timer->tim_active= 1;

    if (!timer_chain)
        timer_chain= timer;
    else if (timeout < timer_chain->tim_time)
    {
        timer->tim_next= timer_chain;
        timer_chain= timer;
    }
    else
    {
        timer_index= timer_chain;
        while (timer_index->tim_next &&
            timer_index->tim_next->tim_time < timeout)
            timer_index= timer_index->tim_next;
        timer->tim_next= timer_index->tim_next;
        timer_index->tim_next= timer;
    }
    if (next_timeout == 0 || timer_chain->tim_time < next_timeout)
        set_timer();
}

PUBLIC void clk_tick (mess)
message *mess;
{
    next_timeout= 0;
    set_timer();
}

PRIVATE void clk_fast_release (timer)
timer_t *timer;
{
    timer_t *timer_index;

    if (!timer->tim_active)
        return;

    if (timer == timer_chain)
        timer_chain= timer_chain->tim_next;
    else
    {

```

```

        timer_index= timer_chain;
        while (timer_index && timer_index->tim_next != timer)
            timer_index= timer_index->tim_next;
        assert(timer_index);
        timer_index->tim_next= timer->tim_next;
    }
    timer->tim_active= 0;
}

PRIVATE void set_timer()
{
    time_t new_time;
    time_t curr_time;

    if (!timer_chain)
        return;

    curr_time= get_time();
    new_time= timer_chain->tim_time;
    if (new_time <= curr_time)
    {
        clk_call_expire= 1;
        return;
    }

    if (next_timeout == 0 || new_time < next_timeout)
    {
#ifdef __minix_vmd
        static message mess;

        next_timeout= new_time;

        new_time -= curr_time;

        mess.m_type= SET_SYNC_AL;
        mess.CLOCK_PROC_NR= this_proc;
        mess.DELTA_TICKS= new_time;
        if (sendrec (clk_tasknr, &mess) < 0)
            ip_panic(("unable to sendrec"));
        if (mess.m_type != OK)
            ip_panic(("can't set timer"));
#else /* Minix 3 */
        next_timeout= new_time;
        new_time -= curr_time;

        if (sys_setalarm(new_time, 0) != OK)
            ip_panic(("can't set timer"));
#endif
    }
}

PUBLIC void clk_untimer (timer)
timer_t *timer;
{
    clk_fast_release (timer);
    set_timer();
}

PUBLIC void clk_expire_timers()
{
    time_t curr_time;
    timer_t *timer_index;

    clk_call_expire= 0;

    if (timer_chain == NULL)
        return;

    curr_time= get_time();
    while (timer_chain && timer_chain->tim_time<=curr_time)
    {
        assert(timer_chain->tim_active);
        timer_chain->tim_active= 0;
        timer_index= timer_chain;
    }
}

```



```
        timer_chain= timer_chain->tim_next;
        (*timer_index->tim_func)(timer_index->tim_ref, timer_index);
    }
    set_timer();
}

/*
 * $PchId: clock.c,v 1.10 2005/06/28 14:23:40 philip Exp $
 */
```

```
/*
inet/const.h

Created:      Dec 30, 1991 by Philip Homburg

Copyright 1995 Philip Homburg
*/

#ifndef INET__CONST_H
#define INET__CONST_H

#ifndef DEBUG
#define DEBUG    0
#endif

#ifndef NDEBUG
#define NDEBUG    0
#endif

#define CLOCK_GRAN    1        /* in HZ */

#define where() printf("%s,%d: ", __FILE__, __LINE__)

#define NW_SUSPEND        SUSPEND
#define NW_WOULDBLOCK    EWOULDBLOCK
#define NW_OK            OK

#define BUF_S            512

#endif /* INET__CONST_H */

/*
 * $PchId: const.h,v 1.7 2000/08/12 09:21:44 philip Exp $
 */
```

```
/*      this file contains the interface of the network software with rest of
        minix. Furthermore it contains the main loop of the network task.
```

Copyright 1995 Philip Homburg

The valid messages and their parameters are:

from FS:

m_type	DEVICE	PROC_NR	COUNT	POSITION	ADDRESS
NW_OPEN	minor dev	proc nr	mode		
NW_CLOSE	minor dev	proc nr			
NW_IOCTL	minor dev	proc nr		NWIO..	address
NW_READ	minor dev	proc nr	count		address
NW_WRITE	minor dev	proc nr	count		address
NW_CANCEL	minor dev	proc nr			

from DL_ETH:

m_type	DL_PORT	DL_PROC	DL_COUNT	DL_STAT	DL_TIME
DL_INIT_REPLY	minor dev	proc nr	rd_count	0 stat	time
DL_TASK_REPLY	minor dev	proc nr	rd_count	err stat	time

```
*/
```

```
#include "inet.h"
```

```
#define _MINIX_SOURCE 1
```

```
#include <fcntl.h>
```

```
#include <time.h>
```

```
#include <unistd.h>
```

```
#include <sys/svrctl.h>
```

```
#include "mq.h"
```

```
#include "qp.h"
```

```
#include "proto.h"
```

```
#include "generic/type.h"
```

```
#include "generic/arp.h"
```

```
#include "generic/assert.h"
```

```
#include "generic/buf.h"
```

```
#include "generic/clock.h"
```

```
#include "generic/eth.h"
```

```
#include "generic/event.h"
```

```
#include "generic/ip.h"
```

```
#include "generic/psip.h"
```

```
#include "generic/rand256.h"
```

```
#include "generic/sr.h"
```

```
#include "generic/tcp.h"
```

```
#include "generic/udp.h"
```

```
THIS_FILE
```

```
#define RANDOM_DEV_NAME "/dev/random"
```

```

int this_proc;          /* Process number of this server. */

#ifdef __minix_vmd
static int synal_tasknr= ANY;
#endif

/* Killing Solaris */
int killer_inet= 0;

#ifdef BUF_CONSISTENCY_CHECK
extern int inet_buf_debug;
#endif

_PROTOTYPE( void main, (void) );

FORWARD _PROTOTYPE( void nw_conf, (void) );
FORWARD _PROTOTYPE( void nw_init, (void) );

PUBLIC void main()
{
    mq_t *mq;
    int r;
    int source, timerand, fd;
    struct fssignon device;
#ifdef __minix_vmd
    struct systaskinfo info;
#endif
    u8_t randbits[32];
    struct timeval tv;

    #if DEBUG
        printf("Starting inet...\n");
        printf("%s\n", version);
    #endif

    /* Read configuration. */
    nw_conf();

    /* Get a random number */
    timerand= 1;
    fd= open(RANDOM_DEV_NAME, O_RDONLY | O_NONBLOCK);
    if (fd != -1)
    {
        r= read(fd, randbits, sizeof(randbits));
        if (r == sizeof(randbits))
            timerand= 0;
        else
        {
            printf("unable to read random data from %s: %s\n",
                RANDOM_DEV_NAME, r == -1 ? strerror(errno) :
                r == 0 ? "EOF" : "not enough data");
        }
        close(fd);
    }
    else
    {
        printf("unable to open random device %s: %s\n",
            RANDOM_DEV_NAME, strerror(errno));
    }
    if (timerand)
    {
        printf("using current time for random-number seed\n");
#ifdef __minix_vmd
        r= sysutime(UTIME_TIMEOFDAY, &tv);
#else /* Minix 3 */
        r= gettimeofday(&tv, NULL);
#endif
        if (r == -1)
        {
            printf("sysutime failed: %s\n", strerror(errno));
            exit(1);
        }
        memcpy(randbits, &tv, sizeof(tv));
    }
}

```

```

    }
    init_rand256(randbits);
#ifdef __minix_vmd
    if (svrctl(SYSSIGNON, (void *) &info) == -1) pause();

    /* Our new identity as a server. */
    this_proc = info.proc_nr;
#else /* Minix 3 */

    /* Our new identity as a server. */
    if ((this_proc = getprocnr()) < 0)
        ip_panic(( "unable to get own process nr\n" ));
#endif

    /* Register the device group. */
    device.dev= ip_dev;
    device.style= STYLE_CLONE;
    if (svrctl(FSSIGNON, (void *) &device) == -1) {
        printf("inet: error %d on registering ethernet devices\n",
            errno);
        pause();
    }

#ifdef BUF_CONSISTENCY_CHECK
    inet_buf_debug= (getenv("inetbufdebug") &&
        (strcmp(getenv("inetbufdebug"), "on") == 0));
    inet_buf_debug= 100;
    if (inet_buf_debug)
    {
        ip_warning(( "buffer consistency check enabled" ));
    }
#endif

    if (getenv("killerinet"))
    {
        ip_warning(( "killer inet active" ));
        killer_inet= 1;
    }

#ifdef __minix_vmd
    r= sys_findproc(SYN_AL_NAME, &synal_tasknr, 0);
    if (r != OK)
        ip_panic(( "unable to find synchronous alarm task: %d\n", r ));
#endif

    nw_init();
    while (TRUE)
    {
#ifdef BUF_CONSISTENCY_CHECK
        if (inet_buf_debug)
        {
            static int buf_debug_count= 0;

            if (++buf_debug_count >= inet_buf_debug)
            {
                buf_debug_count= 0;
                if (!bf_consistency_check())
                    break;
            }
        }
#endif

        if (ev_head)
        {
            ev_process();
            continue;
        }
        if (clk_call_expire)
        {
            clk_expire_timers();
            continue;
        }
        mq= mq_get();
        if (!mq)

```

```

        ip_panic(( "out of messages" ));

        r= receive (ANY, &mq->mq_mess);
        if (r<0)
        {
            ip_panic(( "unable to receive: %d", r));
        }
        reset_time();
        source= mq->mq_mess.m_source;
        if (source == FS_PROC_NR)
        {
            sr_rec(mq);
        }
#ifdef __minix_vmd
        else if (source == synal_tasknr)
        {
            clk_tick (&mq->mq_mess);
            mq_free(mq);
        }
#else /* Minix 3 */
        else if (mq->mq_mess.m_type == SYN_ALARM)
        {
            clk_tick(&mq->mq_mess);
            mq_free(mq);
        }
        else if (mq->mq_mess.m_type == PROC_EVENT)
        {
            /* signaled */
            /* probably SIGTERM */
            mq_free(mq);
        }
        else if (mq->mq_mess.m_type & NOTIFY_MESSAGE)
        {
            /* A driver is (re)started. */
            eth_check_drivers(&mq->mq_mess);
            mq_free(mq);
        }
#endif
        else
        {
            compare(mq->mq_mess.m_type, ==, DL_TASK_REPLY);
            eth_rec(&mq->mq_mess);
            mq_free(mq);
        }
    }
    ip_panic(( "task is not allowed to terminate" ));
}

PRIVATE void nw_conf()
{
    read_conf();
    eth_prep();
    arp_prep();
    psip_prep();
    ip_prep();
    tcp_prep();
    udp_prep();
}

PRIVATE void nw_init()
{
    mq_init();
    qp_init();
    bf_init();
    clk_init();
    sr_init();
    eth_init();
    arp_init();
    psip_init();
    ip_init();
    tcp_init();
    udp_init();
}

```

```
PUBLIC void panic0(file, line)
char *file;
int line;
{
    printf("panic at %s,%d: ", file, line);
}

PUBLIC void inet_panic()
{
    printf("\ninet stacktrace: ");
    stacktrace();
#ifdef __minix_vmd
    sys_abort(RBT_PANIC);
#else /* Minix 3 */
    (panic)("INET", "aborted due to a panic", NO_NUM);
#endif
    for(;;);
}

#if !NDEBUG
PUBLIC void bad_assertion(file, line, what)
char *file;
int line;
char *what;
{
    panic0(file, line);
    printf("assertion \"%s\" failed", what);
    panic();
}

PUBLIC void bad_compare(file, line, lhs, what, rhs)
char *file;
int line;
int lhs;
char *what;
int rhs;
{
    panic0(file, line);
    printf("compare (%d) %s (%d) failed", lhs, what, rhs);
    panic();
}
#endif /* !NDEBUG */

/*
 * $PchId: inet.c,v 1.23 2005/06/28 14:27:22 philip Exp $
 */
```

```
/*
inet/inet.h

Created:          Dec 30, 1991 by Philip Homburg

Copyright 1995 Philip Homburg
*/

#ifndef INET__INET_H
#define INET__INET_H

#define _SYSTEM 1          /* get OK and negative error codes */

#include <sys/types.h>
#include <errno.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>

#ifdef __minix_vmd

#include <minix/ansi.h>
#include <minix/cfg_public.h>
#include <minix/type.h>

#else /* Assume at least Minix 3.x */

#include <unistd.h>
#include <sys/ioc_file.h>
#include <sys/time.h>
#include <minix/config.h>
#include <minix/type.h>

#define _NORETURN          /* Should be non empty for GCC */

typedef int ioreq_t;

#endif

#include <minix/const.h>
#include <minix/com.h>
#include <minix/syslib.h>
#include <minix/sysutil.h>
#include <net/hton.h>
#include <net/gen/ether.h>
#include <net/gen/eth_hdr.h>
#include <net/gen/eth_io.h>
#include <net/gen/in.h>
#include <net/gen/ip_hdr.h>
#include <net/gen/ip_io.h>
#include <net/gen/icmp.h>
#include <net/gen/icmp_hdr.h>
#include <net/gen/oneCsum.h>
#include <net/gen/psip_hdr.h>
#include <net/gen/psip_io.h>
#include <net/gen/route.h>
#include <net/gen/tcp.h>
#include <net/gen/tcp.h>
#include <net/gen/tcp_hdr.h>
#include <net/gen/tcp_io.h>
#include <net/gen/udp.h>
#include <net/gen/udp_hdr.h>
#include <net/gen/udp_io.h>

#include <net/gen/arp_io.h>
#include <net/ioctl.h>

#include "const.h"
#include "inet_config.h"

#define PUBLIC
#define EXTERN extern
#define PRIVATE static
#define FORWARD static
```



```
#define THIS_FILE static char *this_file= __FILE__;  
  
_PROTOTYPE( void panic0, (char *file, int line) );  
_PROTOTYPE( void inet_panic, (void) ) _NORETURN;  
  
#define ip_panic(print_list) \  
    (panic0(this_file, __LINE__), printf print_list, panic())  
#define panic() inet_panic()  
  
#if DEBUG  
#define ip_warning(print_list) \  
    ( \  
        printf("warning at %s,%d: ", this_file, __LINE__), \  
        printf print_list, \  
        printf("\ninet stacktrace: "), \  
        stacktrace() \  
    )  
#else  
#define ip_warning(print_list) ((void) 0)  
#endif  
  
#define DBLOCK(level, code) \  
    do { if ((level) & DEBUG) { where(); code; } } while(0)  
#define DIFBLOCK(level, condition, code) \  
    do { if (((level) & DEBUG) && (condition)) \  
        { where(); code; } } while(0)  
  
#if _ANSI  
#define ARGS(x) x  
#else /* _ANSI */  
#define ARGS(x) ()  
#endif /* _ANSI */  
  
extern int this_proc;  
extern char version[];  
  
void stacktrace ARGS(( void ));  
  
#endif /* INET__INET_H */  
  
/*  
 * $PchId: inet.h,v 1.16 2005/06/28 14:27:54 philip Exp $  
 */
```

```
/*
inet/inet_config.c

Created:      Nov 11, 1992 by Philip Homburg

Modified:     Apr 07, 2001 by Kees J. Bot
              Read the configuration file and fill in the xx_conf[] arrays.

Copyright 1995 Philip Homburg
*/

#define _MINIX_SOURCE 1
#define _POSIX_SOURCE 1

#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <minix/type.h>
#include <minix/sysutil.h>
#include <minix/syslib.h>
#include "inet_config.h"

struct eth_conf eth_conf[IP_PORT_MAX];
struct psip_conf psip_conf[IP_PORT_MAX];
struct ip_conf ip_conf[IP_PORT_MAX];
struct tcp_conf tcp_conf[IP_PORT_MAX];
struct udp_conf udp_conf[IP_PORT_MAX];
dev_t ip_dev;

int eth_conf_nr;
int psip_conf_nr;
int ip_conf_nr;
int tcp_conf_nr;
int udp_conf_nr;

int ip_forward_directed_bcast= 0;      /* Default is off */

static u8_t iftype[IP_PORT_MAX];      /* Interface in use as? */
static int ifdefault= -1;              /* Default network interface. */

static void fatal(char *label)
{
    printf("init: %s: %s\n", label, strerror(errno));
    exit(1);
}

static void check_rm(char *device)
/* Check if a device is not among the living. */
{
    if (unlink(device) < 0) {
        if (errno == ENOENT) return;
        fatal(device);
    }
    printf("rm %s\n", device);
}

static void check_mknod(char *device, mode_t mode, int minor)
/* Check if a device exists with the proper device number. */
{
    struct stat st;
    dev_t dev;

    dev= (ip_dev & 0xFF00) | minor;

    if (stat(device, &st) < 0) {
        if (errno != ENOENT) fatal(device);
    } else {
        if (S_ISCHR(st.st_mode) && st.st_rdev == dev) return;
        if (unlink(device) < 0) fatal(device);
    }
}
```

```

    if (mknod(device, S_IFCHR | mode, dev) < 0) fatal(device);
    printf("mknod %s c %d %d\n", device, (ip_dev >> 8), minor);
}

static void check_ln(char *old, char *new)
/* Check if 'old' and 'new' are still properly linked. */
{
    struct stat st_old, st_new;

    if (stat(old, &st_old) < 0) fatal(old);
    if (stat(new, &st_new) < 0) {
        if (errno != ENOENT) fatal(new);
    } else {
        if (st_new.st_dev == st_old.st_dev
            && st_new.st_ino == st_old.st_ino) {
            return;
        }
        if (unlink(new) < 0) fatal(new);
    }

    if (link(old, new) < 0) fatal(new);
    printf("ln %s %s\n", old, new);
}

static void check_dev(int type, int ifno)
/* Check if the device group with interface number 'ifno' exists and has the
 * proper device numbers. If 'type' is -1 then the device group must be
 * removed.
 */
{
    static struct devlist {
        char      *defname;
        mode_t    mode;
        u8_t      minor_off;
    } devlist[] = {
        { "/dev/eth",      0600,    ETH_DEV_OFF    },
        { "/dev/psip",     0600,    PSIP_DEV_OFF   },
        { "/dev/ip",       0600,    IP_DEV_OFF      },
        { "/dev/tcp",      0666,    TCP_DEV_OFF     },
        { "/dev/udp",      0666,    UDP_DEV_OFF     },
    };
    struct devlist *dvp;
    int i;
    char device[sizeof("/dev/psip99")];
    char *dp;

    for (i = 0; i < sizeof(devlist) / sizeof(devlist[0]); i++) {
        dvp = &devlist[i];
        strcpy(device, dvp->defname);
        dp = device + strlen(device);
        if (ifno >= 10) *dp++ = '0' + (ifno / 10);
        *dp++ = '0' + (ifno % 10);
        *dp = 0;

        if (type == 0
            || (i == 0 && type != NETTYPE_ETH)
            || (i == 1 && type != NETTYPE_PSIP)
        ) {
            check_rm(device);
            if (ifno == ifdefault) check_rm(dvp->defname);
        } else {
            check_mknod(device, dvp->mode,
                if2minor(ifno, dvp->minor_off));
            if (ifno == ifdefault) check_ln(device, dvp->defname);
        }
    }
}

static int cfg_fd;
static char word[16];
static unsigned line;

static void error(void)

```

```

{
    printf("inet: error on line %u\n", line);
    exit(1);
}

static void token(int need)
{
    /* Read a word from the configuration file. Return a null string on
     * EOF. Return a punctuation as a one character word. If 'need' is
     * true then an actual word is expected at this point, so err out if
     * not.
     */
    unsigned char *wp;
    static unsigned char c= '\n';

    wp= (unsigned char *) word;
    *wp = 0;

    while (c <= ' ') {
        if (c == '\n') line++;
        if (read(cfg_fd, &c, 1) != 1) {
            if (need) error();
            return;
        }
    }

    do {
        if (wp < (unsigned char *) word + sizeof(word)-1) *wp++ = c;
        if (read(cfg_fd, &c, 1) != 1) c= ' ';
        if (word[0] == ';' || word[0] == '{' || word[0] == '}') {
            if (need) error();
            break;
        }
    } while (c > ' ' && c != ';' && c != '{' && c != '}');
    *wp = 0;
}

static unsigned number(char *str, unsigned max)
{
    /* Interpret a string as an unsigned decimal number, no bigger than
     * 'max'. Return this number.
     */
    char *s;
    unsigned n, d;

    s= str;
    n= 0;
    while ((d= (*s - '0')) < 10 && n <= max) {
        n= n * 10 + d;
        s++;
    }
    if (*s != 0 || n > max) {
        printf("inet: '%s' is not a number <= %u\n", str, max);
        error();
    }
    return n;
}

void read_conf(void)
{
    int i, j, ifno, type, port, enable;
    struct eth_conf *ecp;
    struct psip_conf *pcp;
    struct ip_conf *icp;
    struct stat st;

    /* Open the configuration file. */
    if ((cfg_fd= open(PATH_INET_CONF, O_RDONLY)) == -1)
        fatal(PATH_INET_CONF);

    ecp= eth_conf;
    pcp= psip_conf;
    icp= ip_conf;

```

```

while (token(0), word[0] != 0) {
    if (strncmp(word, "eth", 3) == 0) {
        ecp->ec_ifno= ifno= number(word+3, IP_PORT_MAX-1);
        type= NETTYPE_ETH;
        port= eth_conf_nr;
        token(1);
        if (strcmp(word, "vlan") == 0) {
            token(1);
            ecp->ec_vlan= number(word, (1<<12)-1);
            token(1);
            if (strncmp(word, "eth", 3) != 0) {
                printf(
                    "inet: VLAN eth%d can't be built on %s\n",
                        ifno, word);
                exit(1);
            }
            ecp->ec_port= number(word+3, IP_PORT_MAX-1);
        } else {
            ecp->ec_task= alloc(strlen(word)+1);
            strcpy(ecp->ec_task, word);
            token(1);
            ecp->ec_port= number(word, IP_PORT_MAX-1);
        }
        ecp++;
        eth_conf_nr++;
    } else
    if (strncmp(word, "psip", 4) == 0) {
        pcp->pc_ifno= ifno= number(word+4, IP_PORT_MAX-1);
        type= NETTYPE_PSIP;
        port= psip_conf_nr;
        pcp++;
        psip_conf_nr++;
    } else {
        printf("inet: Unknown device '%s'\n", word);
        error();
    }
    iftype[ifno]= type;
    icp->ic_ifno= ifno;
    icp->ic_devtype= type;
    icp->ic_port= port;
    tcp_conf[tcp_conf_nr].tc_port= ip_conf_nr;
    udp_conf[udp_conf_nr].uc_port= ip_conf_nr;

    enable= 7;          /* 1 = IP, 2 = TCP, 4 = UDP */

    token(0);
    if (word[0] == '{') {
        token(0);
        while (word[0] != '}') {
            if (strcmp(word, "default") == 0) {
                if (ifdefault != -1) {
                    printf(
                        "inet: ip%d and ip%d can't both be default\n",
                            ifdefault, ifno);
                    error();
                }
                ifdefault= ifno;
                token(0);
            } else
            if (strcmp(word, "no") == 0) {
                token(1);
                if (strcmp(word, "ip") == 0) {
                    enable= 0;
                } else
                if (strcmp(word, "tcp") == 0) {
                    enable &= ~2;
                } else
                if (strcmp(word, "udp") == 0) {
                    enable &= ~4;
                } else {
                    printf(
                        "inet: Can't do 'no %s'\n",
                            word);
                    exit(1);
                }
            }
        }
    }
}

```

```

        }
        token(0);
    } else {
        printf("inet: Unknown option '%s'\n",
            word);
        exit(1);
    }
    if (word[0] == ';') token(0);
    else
        if (word[0] != '}') error();
    }
    token(0);
}
if (word[0] != ';' && word[0] != 0) error();

if (enable & 1) icp++, ip_conf_nr++;
if (enable & 2) tcp_conf_nr++;
if (enable & 4) udp_conf_nr++;
}

if (ifdefault == -1) {
    printf("inet: No networks or no default network defined\n");
    exit(1);
}

/* Translate VLAN network references to port numbers. */
for (i= 0; i < eth_conf_nr; i++) {
    ecp= &eth_conf[i];
    if (eth_is_vlan(ecp)) {
        for (j= 0; j < eth_conf_nr; j++) {
            if (eth_conf[j].ec_ifno == ecp->ec_port
                && !eth_is_vlan(&eth_conf[j]))
            {
                ecp->ec_port= j;
                break;
            }
        }
        if (j == eth_conf_nr) {
            printf(
                "inet: VLAN eth%d can't be built on eth%d\n",
                ecp->ec_ifno, ecp->ec_port);
            exit(1);
        }
    }
}

/* Set umask 0 so we can creat mode 666 devices. */
(void) umask(0);

/* See what the device number of /dev/ip is. That's what we
 * used last time for the network devices, so we keep doing so.
 */
if (stat("/dev/ip", &st) < 0) fatal("/dev/ip");
ip_dev= st.st_rdev;

for (i= 0; i < IP_PORT_MAX; i++) {
    /* Create network devices. */
    check_dev(itype[i], i);
}
}

void *alloc(size_t size)
{
    /* Allocate memory on the heap with sbrk(). */

    return sbrk((size + (sizeof(char *) - 1)) & ~(sizeof(char *) - 1));
}

/*
 * $PchId: inet_config.c,v 1.10 2003/08/21 09:26:02 philip Exp $
 */

```

```
/*
inet/inet_config.h

Created:      Nov 11, 1992 by Philip Homburg

Defines values for configurable parameters. The structure definitions for
configuration information are also here.

Copyright 1995 Philip Homburg
*/

#ifndef INET__INET_CONFIG_H
#define INET__INET_CONFIG_H

/* Inet configuration file. */
#define PATH_INET_CONF  "/etc/inet.conf"

#define IP_PORT_MAX      32      /* Up to this many network devices */
extern int eth_conf_nr;      /* Number of ethernetets */
extern int psip_conf_nr;    /* Number of Pseudo IP networks */
extern int ip_conf_nr;      /* Number of configured IP layers */
extern int tcp_conf_nr;     /* Number of configured TCP layers */
extern int udp_conf_nr;     /* Number of configured UDP layers */

extern dev_t ip_dev;        /* Device number of /dev/ip */

struct eth_conf
{
    char *ec_task;           /* Kernel ethernet task name if nonnull */
    u8_t ec_port;            /* Task port (!vlan) or Ethernet port (vlan) */
    u8_t ec_ifno;            /* Interface number of /dev/eth* */
    u16_t ec_vlan;           /* VLAN number of this net if task == NULL */
};
#define eth_is_vlan(ecp)    ((ecp)->ec_task == NULL)

struct psip_conf
{
    u8_t pc_ifno;            /* Interface number of /dev/psip* */
};

struct ip_conf
{
    u8_t ic_devtype;         /* Underlying device type: Ethernet / PSIP */
    u8_t ic_port;            /* Port of underlying device */
    u8_t ic_ifno;           /* Interface number of /dev/ip*, tcp*, udp* */
};

struct tcp_conf
{
    u8_t tc_port;            /* IP port number */
};

struct udp_conf
{
    u8_t uc_port;            /* IP port number */
};

/* Types of networks. */
#define NETTYPE_ETH        1
#define NETTYPE_PSIP      2

/* To compute the minor device number for a device on an interface. */
#define if2minor(ifno, dev)  ((ifno) * 8 + (dev))

/* Offsets of the minor device numbers within a group per interface. */
#define ETH_DEV_OFF        0
#define PSIP_DEV_OFF       0
#define IP_DEV_OFF         1
#define TCP_DEV_OFF        2
#define UDP_DEV_OFF        3

extern struct eth_conf eth_conf[IP_PORT_MAX];
extern struct psip_conf psip_conf[IP_PORT_MAX];
extern struct ip_conf ip_conf[IP_PORT_MAX];
```

```
extern struct tcp_conf tcp_conf[IP_PORT_MAX];
extern struct udp_conf udp_conf[IP_PORT_MAX];
void read_conf(void);
extern char *sbrk(int);
void *alloc(size_t size);

/* Options */
extern int ip_forward_directed_bcast;

#endif /* INET__INET_CONFIG_H */

/*
 * $PchId: inet_config.h,v 1.10 2003/08/21 09:24:33 philip Exp $
 */
```



```

/*
inet/mnx_eth.c

Created:      Jan 2, 1992 by Philip Homburg

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "proto.h"
#include "osdep_eth.h"
#include "generic/type.h"

#include "generic/assert.h"
#include "generic/buf.h"
#include "generic/clock.h"
#include "generic/eth.h"
#include "generic/eth_int.h"
#include "generic/sr.h"

THIS_FILE

static int recv_debug= 0;

FORWARD _PROTOTYPE( void setup_read, (eth_port_t *eth_port) );
FORWARD _PROTOTYPE( void read_int, (eth_port_t *eth_port, int count) );
FORWARD _PROTOTYPE( void write_int, (eth_port_t *eth_port) );
FORWARD _PROTOTYPE( void eth_recvev, (event_t *ev, ev_arg_t ev_arg) );
FORWARD _PROTOTYPE( void eth_sendev, (event_t *ev, ev_arg_t ev_arg) );
FORWARD _PROTOTYPE( eth_port_t *find_port, (message *m) );
FORWARD _PROTOTYPE( void eth_restart, (eth_port_t *eth_port, int tasknr) );

PUBLIC void osdep_eth_init()
{
    int i, r, tasknr, rport;
    struct eth_conf *ecp;
    eth_port_t *eth_port, *rep;
    message mess;

    /* First initialize normal ethernet interfaces */
    for (i= 0, ecp= eth_conf, eth_port= eth_port_table;
         i<eth_conf_nr; i++, ecp++, eth_port++)
    {
        if (eth_is_vlan(ecp))
            continue;
#ifdef __minix_vmd
        r= sys_findproc(ecp->ec_task, &tasknr, 0);
#else /* Minix 3 */
        r = _pm_findproc(ecp->ec_task, &tasknr);
#endif
        if (r != OK)
        {
            /* Eventually, we expect ethernet drivers to be
             * started after INET. So we always end up here. And
             * the findproc can be removed.
             */
            printf("eth%d: unable to find task %s: %d\n",
                   i, ecp->ec_task, r);
            tasknr= ANY;
        }

        eth_port->etp_osdep.etp_port= ecp->ec_port;
        eth_port->etp_osdep.etp_task= tasknr;
        ev_init(&eth_port->etp_osdep.etp_recvev);

        mess.m_type= DL_INIT;
        mess.DL_PORT= eth_port->etp_osdep.etp_port;
        mess.DL_PROC= this_proc;
        mess.DL_MODE= DL_NOMODE;

        if (tasknr == ANY)
            r= ENXIO;
        else
        {

```

```

        r= send(eth_port->etp_osdep.etp_task, &mess);
        if (r<0)
        {
            printf(
"osdep_eth_init: unable to send to ethernet task, error= %d\n" ,
                r);
        }
    }

    if (r == OK)
    {
        r= receive(eth_port->etp_osdep.etp_task, &mess);
        if (r<0)
        {
            printf(
"osdep_eth_init: unable to receive from ethernet task, error= %d\n" ,
                r);
        }
    }

    if (r == OK)
    {
        r= mess.m3_i1;
        if (r == ENXIO)
        {
            printf(
"osdep_eth_init: no ethernet device at task=%d,port=%d\n" ,
                eth_port->etp_osdep.etp_task,
                eth_port->etp_osdep.etp_port);
        }
        else if (r < 0)
        {
            ip_panic((
"osdep_eth_init: DL_INIT returned error %d\n" ,
                r));
        }
        else if (mess.m3_i1 != eth_port->etp_osdep.etp_port)
        {
            ip_panic((
"osdep_eth_init: got reply for wrong port (got %d, expected %d)\n" ,
                mess.m3_i1,
                eth_port->etp_osdep.etp_port));
        }
    }

    sr_add_minor(if2minor(ecp->ec_ifno, ETH_DEV_OFF),
        i, eth_open, eth_close, eth_read,
        eth_write, eth_ioctl, eth_cancel, eth_select);

    eth_port->etp_flags |= EPF_ENABLED;
    eth_port->etp_vlan= 0;
    eth_port->etp_vlan_port= NULL;
    eth_port->etp_wr_pack= 0;
    eth_port->etp_rd_pack= 0;
    if (r == OK)
    {
        eth_port->etp_ethaddr= *(ether_addr_t *)mess.m3_cal;
        eth_port->etp_flags |= EPF_GOT_ADDR;
        setup_read (eth_port);
    }
}

/* And now come the VLANs */
for (i= 0, ecp= eth_conf, eth_port= eth_port_table;
    i<eth_conf_nr; i++, ecp++, eth_port++)
{
    if (!eth_is_vlan(ecp))
        continue;

    eth_port->etp_osdep.etp_port= ecp->ec_port;
    eth_port->etp_osdep.etp_task= ANY;
    ev_init(&eth_port->etp_osdep.etp_recvev);

    rport= eth_port->etp_osdep.etp_port;

```

```

    assert(rport >= 0 && rport < eth_conf_nr);
    rep= &eth_port_table[rport];
    if (!(rep->etp_flags & EPF_ENABLED))
    {
        printf(
            "eth%d: underlying ethernet device %d not enabled",
            i, rport);
        continue;
    }
    if (rep->etp_vlan != 0)
    {
        printf(
            "eth%d: underlying ethernet device %d is a VLAN",
            i, rport);
        continue;
    }

    if (rep->etp_flags & EPF_GOT_ADDR)
    {
        eth_port->etp_ethaddr= rep->etp_ethaddr;
        eth_port->etp_flags |= EPF_GOT_ADDR;
    }

    sr_add_minor(if2minor(ecp->ec_ifno, ETH_DEV_OFF),
        i, eth_open, eth_close, eth_read,
        eth_write, eth_ioctl, eth_cancel, eth_select);

    eth_port->etp_flags |= EPF_ENABLED;
    eth_port->etp_vlan= ecp->ec_vlan;
    eth_port->etp_vlan_port= rep;
    assert(eth_port->etp_vlan != 0);
    eth_port->etp_wr_pack= 0;
    eth_port->etp_rd_pack= 0;
    eth_reg_vlan(rep, eth_port);
}

}

PUBLIC void eth_write_port(eth_port, pack)
eth_port_t *eth_port;
acc_t *pack;
{
    eth_port_t *loc_port;
    message mess1, block_msg;
    int i, pack_size;
    acc_t *pack_ptr;
    iovec_t *iovec;
    u8_t *eth_dst_ptr;
    int multicast, r;
    ev_arg_t ev_arg;

    assert(!no_ethWritePort);
    assert(!eth_port->etp_vlan);

    assert(eth_port->etp_wr_pack == NULL);
    eth_port->etp_wr_pack= pack;

    iovec= eth_port->etp_osdep.etp_wr_iovec;
    pack_size= 0;
    for (i=0, pack_ptr= pack; i<IOVEC_NR && pack_ptr; i++,
        pack_ptr= pack_ptr->acc_next)
    {
        iovec[i].iov_addr= (vir_bytes)ptr2acc_data(pack_ptr);
        pack_size += iovec[i].iov_size= pack_ptr->acc_length;
    }
    if (i>= IOVEC_NR)
    {
        pack= bf_pack(pack);          /* packet is too fragmented */
        eth_port->etp_wr_pack= pack;
        pack_size= 0;
        for (i=0, pack_ptr= pack; i<IOVEC_NR && pack_ptr;
            i++, pack_ptr= pack_ptr->acc_next)
        {
            iovec[i].iov_addr= (vir_bytes)ptr2acc_data(pack_ptr);
            pack_size += iovec[i].iov_size= pack_ptr->acc_length;
        }
    }
}

```

```

    }
}
assert (i < IOVEC_NR);
assert (pack_size >= ETH_MIN_PACK_SIZE);

if (i == 1)
{
    /* simple packets can be sent using DL_WRITE instead of
     * DL_WRITEV.
     */
    mess1.DL_COUNT= iovec[0].iov_size;
    mess1.DL_ADDR= (char *)iovec[0].iov_addr;
    mess1.m_type= DL_WRITE;
}
else
{
    mess1.DL_COUNT= i;
    mess1.DL_ADDR= (char *)iovec;
    mess1.m_type= DL_WRITEV;
}
mess1.DL_PORT= eth_port->etp_osdep.etp_port;
mess1.DL_PROC= this_proc;
mess1.DL_MODE= DL_NOMODE;

for (;;)
{
    r= sendrec(eth_port->etp_osdep.etp_task, &mess1);
    if (r != ELOCKED)
        break;

    /* ethernet task is sending to this task, I hope */
    r= receive(eth_port->etp_osdep.etp_task, &block_msg);
    if (r < 0)
        ip_panic(("unable to receive"));

    loc_port= eth_port;
    if (loc_port->etp_osdep.etp_port != block_msg.DL_PORT ||
        loc_port->etp_osdep.etp_task != block_msg.m_source)
    {
        loc_port= find_port(&block_msg);
    }
    assert(block_msg.DL_STAT & (DL_PACK_SEND|DL_PACK_RECV));
    if (block_msg.DL_STAT & DL_PACK_SEND)
    {
        assert(loc_port != eth_port);
        loc_port->etp_osdep.etp_sendrepl= block_msg;
        ev_arg.ev_ptr= loc_port;
        ev_enqueue(&loc_port->etp_sendev, eth_sendev, ev_arg);
    }
    if (block_msg.DL_STAT & DL_PACK_RECV)
    {
        if (recv_debug)
        {
            printf(
                "eth_write_port(block_msg): eth%d got DL_PACK_RECV\n",
                loc_port-eth_port_table);
        }
        loc_port->etp_osdep.etp_recvrepl= block_msg;
        ev_arg.ev_ptr= loc_port;
        ev_enqueue(&loc_port->etp_osdep.etp_recvev,
            eth_recvev, ev_arg);
    }
}

if (r < 0)
{
    printf("eth_write_port: sendrec to %d failed: %d\n",
        eth_port->etp_osdep.etp_task, r);
    return;
}

assert(mess1.m_type == DL_TASK_REPLY &&
    mess1.DL_PORT == eth_port->etp_osdep.etp_port &&
    mess1.DL_PROC == this_proc);

```

```

assert((mess1.DL_STAT >> 16) == OK);

if (mess1.DL_STAT & DL_PACK_RECV)
{
    if (recv_debug)
    {
        printf(
            "eth_write_port(mess1): eth%d got DL_PACK_RECV\n",
            mess1.DL_PORT);
    }
    eth_port->etp_osdep.etp_recvrepl= mess1;
    ev_arg.ev_ptr= eth_port;
    ev_enqueue(&eth_port->etp_osdep.etp_recvev, eth_recvev,
        ev_arg);
}
if (!(mess1.DL_STAT & DL_PACK_SEND))
{
    /* Packet is not yet sent. */
    return;
}

/* If the port is in promiscuous mode or the packet is
 * broad- or multicast, enqueue the reply packet.
 */
eth_dst_ptr= (u8_t *)ptr2acc_data(pack);
multicast= (*eth_dst_ptr & 1); /* low order bit indicates multicast */
if (multicast || (eth_port->etp_osdep.etp_recvconf & NWE0_EN_PROMISC))
{
    eth_port->etp_osdep.etp_sendrepl= mess1;
    ev_arg.ev_ptr= eth_port;
    ev_enqueue(&eth_port->etp_sendev, eth_sendev, ev_arg);

    /* Pretend that we didn't get a reply. */
    return;
}

/* packet is sent */
bf_afree(eth_port->etp_wr_pack);
eth_port->etp_wr_pack= NULL;
}

PUBLIC void eth_rec(m)
message *m;
{
    int i;
    eth_port_t *loc_port;
    int stat;

    assert(m->m_type == DL_TASK_REPLY);

    set_time (m->DL_CLK);

    for (i=0, loc_port= eth_port_table; i<eth_conf_nr; i++, loc_port++)
    {
        if (loc_port->etp_osdep.etp_port == m->DL_PORT &&
            loc_port->etp_osdep.etp_task == m->m_source)
            break;
    }
    if (i == eth_conf_nr)
    {
        ip_panic(("message from unknown source: %d:%d",
            m->m_source, m->DL_PORT));
    }

    stat= m->DL_STAT & 0xffff;

    assert(stat & (DL_PACK_SEND|DL_PACK_RECV));
    if (stat & DL_PACK_SEND)
        write_int(loc_port);
    if (stat & DL_PACK_RECV)
    {
        if (recv_debug)
        {
            printf("eth_rec: eth%d got DL_PACK_RECV\n",

```

```

        m->DL_PORT);
    }
    read_int(loc_port, m->DL_COUNT);
}

PUBLIC void eth_check_drivers(m)
message *m;
{
    int i, r, tasknr;
    struct eth_conf *ecp;
    eth_port_t *eth_port;
    char *drivename;

    tasknr= m->m_source;
    printf("eth_check_drivers: got a notification from %d\n", tasknr);

    m->m_type= DL_GETNAME;
    r= sendrec(tasknr, m);
    if (r != OK)
    {
        printf("eth_check_drivers: sendrec to %d failed: %d\n",
            tasknr, r);
        return;
    }
    if (m->m_type != DL_NAME_REPLY)
    {
        printf(
            "eth_check_drivers: got bad getname reply (%d) from %d\n",
            m->m_type, tasknr);
        return;
    }

    drivename= m->m3_cal;
    printf("eth_check_drivers: got name: %s\n", drivename);

    /* Re-init ethernet interfaces */
    for (i= 0, ecp= eth_conf, eth_port= eth_port_table;
        i<eth_conf_nr; i++, ecp++, eth_port++)
    {
        if (eth_is_vlan(ecp))
            continue;

        if (strcmp(ecp->ec_task, drivename) != 0)
        {
            /* Wrong driver */
            continue;
        }

        eth_restart(eth_port, tasknr);
    }
}

PUBLIC int eth_get_stat(eth_port, eth_stat)
eth_port_t *eth_port;
eth_stat_t *eth_stat;
{
    int r;
    message mess, mlocked;

    assert(!eth_port->etp_vlan);

    mess.m_type= DL_GETSTAT;
    mess.DL_PORT= eth_port->etp_osdep.etp_port;
    mess.DL_PROC= this_proc;
    mess.DL_ADDR= (char *)eth_stat;

    for (;;)
    {
        r= sendrec(eth_port->etp_osdep.etp_task, &mess);
        if (r != ELOCKED)
            break;

        r= receive(eth_port->etp_osdep.etp_task, &mlocked);
    }
}

```

```

        assert(r == OK);

        compare(mlocked.m_type, ==, DL_TASK_REPLY);
        eth_rec(&mlocked);
    }

    if (r != OK)
    {
        printf("eth_get_stat: sendrec to %d failed: %d\n",
               eth_port->etp_osdep.etp_task, r);
        return EIO;
    }

    assert(mess.m_type == DL_TASK_REPLY);

    r= mess.DL_STAT >> 16;
    assert (r == 0);

    if (mess.DL_STAT)
    {
        eth_rec(&mess);
    }
    return OK;
}

PUBLIC void eth_set_rec_conf (eth_port, flags)
eth_port_t *eth_port;
u32_t flags;
{
    int r;
    unsigned dl_flags;
    message mess, repl_mess;

    assert(!eth_port->etp_vlan);

    if (!(eth_port->etp_flags & EPF_GOT_ADDR))
    {
        /* We have never seen the device. */
        printf("eth_set_rec_conf: waiting for device to appear\n");
        return;
    }

    eth_port->etp_osdep.etp_rcvconf= flags;
    dl_flags= DL_NOMODE;
    if (flags & NWE0_EN_BROAD)
        dl_flags |= DL_BROAD_REQ;
    if (flags & NWE0_EN_MULTI)
        dl_flags |= DL_MULTI_REQ;
    if (flags & NWE0_EN_PROMISC)
        dl_flags |= DL_PROMISC_REQ;

    mess.m_type= DL_INIT;
    mess.DL_PORT= eth_port->etp_osdep.etp_port;
    mess.DL_PROC= this_proc;
    mess.DL_MODE= dl_flags;

    do
    {
        r= sendrec(eth_port->etp_osdep.etp_task, &mess);
        if (r == ELOCKED) /* etp_task is sending to this task,
                           I hope */
        {
            if (receive (eth_port->etp_osdep.etp_task,
                          &repl_mess)< 0)
            {
                ip_panic(("unable to receive"));
            }

            compare(repl_mess.m_type, ==, DL_TASK_REPLY);
            eth_rec(&repl_mess);
        }
    } while (r == ELOCKED);

    if (r < 0)

```

```

    {
        printf("eth_set_rec_conf: sendrec to %d failed: %d\n",
               eth_port->etp_osdep.etp_task, r);
        return;
    }

    assert(mess.m_type == DL_INIT_REPLY);
    if(mess.m3_il != eth_port->etp_osdep.etp_port)
    {
        ip_panic(("got reply for wrong port"));
    }
}

PRIVATE void write_int(eth_port)
eth_port_t *eth_port;
{
    acc_t *pack;
    int multicast;
    u8_t *eth_dst_ptr;

    pack= eth_port->etp_wr_pack;
    eth_port->etp_wr_pack= NULL;

    eth_dst_ptr= (u8_t *)ptr2acc_data(pack);
    multicast= (*eth_dst_ptr & 1); /* low order bit indicates multicast */
    if(multicast || (eth_port->etp_osdep.etp_recvconf & NWE0_EN_PROMISC))
    {
        assert(!no_ethWritePort);
        no_ethWritePort= 1;
        eth_arrive(eth_port, pack, bf_bufsize(pack));
        assert(no_ethWritePort);
        no_ethWritePort= 0;
    }
    else
        bf_afree(pack);

    eth_restart_write(eth_port);
}

PRIVATE void read_int(eth_port, count)
eth_port_t *eth_port;
int count;
{
    acc_t *pack, *cut_pack;

    pack= eth_port->etp_rd_pack;
    eth_port->etp_rd_pack= NULL;

    cut_pack= bf_cut(pack, 0, count);
    bf_afree(pack);

    assert(!no_ethWritePort);
    no_ethWritePort= 1;
    eth_arrive(eth_port, cut_pack, count);
    assert(no_ethWritePort);
    no_ethWritePort= 0;

    eth_port->etp_flags &= ~(EPF_READ_IP|EPF_READ_SP);
    setup_read(eth_port);
}

PRIVATE void setup_read(eth_port)
eth_port_t *eth_port;
{
    eth_port_t *loc_port;
    acc_t *pack, *pack_ptr;
    message mess1, block_msg;
    iovec_t *iovec;
    ev_arg_t ev_arg;
    int i, r;

    assert(!eth_port->etp_vlan);
    assert(!(eth_port->etp_flags & (EPF_READ_IP|EPF_READ_SP)));

```



```

do
{
    assert (!eth_port->etp_rd_pack);

    iovec= eth_port->etp_osdep.etp_rd_iovec;
    pack= bf_memreq (ETH_MAX_PACK_SIZE_TAGGED);

    for (i=0, pack_ptr= pack; i<RD_IOVEC && pack_ptr;
        i++, pack_ptr= pack_ptr->acc_next)
    {
        iovec[i].iov_addr= (vir_bytes)ptr2acc_data(pack_ptr);
        iovec[i].iov_size= (vir_bytes)pack_ptr->acc_length;
    }
    assert (!pack_ptr);

    mess1.m_type= DL_READV;
    mess1.DL_PORT= eth_port->etp_osdep.etp_port;
    mess1.DL_PROC= this_proc;
    mess1.DL_COUNT= i;
    mess1.DL_ADDR= (char *)iovec;

    for (;;)
    {
        if (recv_debug)
        {
            printf("eth%d: sending DL_READV\n",
                mess1.DL_PORT);
        }
        r= sendrec(eth_port->etp_osdep.etp_task, &mess1);
        if (r != ELOCKED)
            break;

        /* ethernet task is sending to this task, I hope */
        r= receive(eth_port->etp_osdep.etp_task, &block_msg);
        if (r < 0)
            ip_panic(("unable to receive"));

        loc_port= eth_port;
        if (loc_port->etp_osdep.etp_port != block_msg.DL_PORT ||
            loc_port->etp_osdep.etp_task !=
                block_msg.m_source)
        {
            loc_port= find_port(&block_msg);
        }
        assert(block_msg.DL_STAT &
            (DL_PACK_SEND|DL_PACK_RECV));
        if (block_msg.DL_STAT & DL_PACK_SEND)
        {
            loc_port->etp_osdep.etp_sendrepl= block_msg;
            ev_arg.ev_ptr= loc_port;
            ev_enqueue(&loc_port->etp_sendev, eth_sendev,
                ev_arg);
        }
        if (block_msg.DL_STAT & DL_PACK_RECV)
        {
            if (recv_debug)
            {
                printf(
                    "setup_read(block_msg): eth%d got DL_PACK_RECV\n",
                    block_msg.DL_PORT);
            }
            assert(loc_port != eth_port);
            loc_port->etp_osdep.etp_recvrepl= block_msg;
            ev_arg.ev_ptr= loc_port;
            ev_enqueue(&loc_port->etp_osdep.etp_recvev,
                eth_recvev, ev_arg);
        }
    }

    if (r < 0)
    {
        printf("mnx_eth'setup_read: sendrec to %d failed: %d\n",
            eth_port->etp_osdep.etp_task, r);
        eth_port->etp_rd_pack= pack;
    }
}

```

```

        eth_port->etp_flags |= EPF_READ_IP;
        continue;
    }

    assert (mess1.m_type == DL_TASK_REPLY &&
            mess1.DL_PORT == mess1.DL_PORT &&
            mess1.DL_PROC == this_proc);
    compare((mess1.DL_STAT >> 16), ==, OK);

    if (mess1.DL_STAT & DL_PACK_RECV)
    {
        if (recv_debug)
        {
            printf(
                "setup_read(mess1): eth%d: got DL_PACK_RECV\n",
                mess1.DL_PORT);
        }
        /* packet received */
        pack_ptr= bf_cut(pack, 0, mess1.DL_COUNT);
        bf_afree(pack);

        assert(!no_ethWritePort);
        no_ethWritePort= 1;
        eth_arrive(eth_port, pack_ptr, mess1.DL_COUNT);
        assert(no_ethWritePort);
        no_ethWritePort= 0;
    }
    else
    {
        /* no packet received */
        eth_port->etp_rd_pack= pack;
        eth_port->etp_flags |= EPF_READ_IP;
    }

    if (mess1.DL_STAT & DL_PACK_SEND)
    {
        eth_port->etp_osdep.etp_sendrepl= mess1;
        ev_arg.ev_ptr= eth_port;
        ev_enqueue(&eth_port->etp_sendev, eth_sendev, ev_arg);
    }
    } while (!(eth_port->etp_flags & EPF_READ_IP));
    eth_port->etp_flags |= EPF_READ_SP;
}

```

```

PRIVATE void eth_recvev(ev, ev_arg)
event_t *ev;
ev_arg_t ev_arg;
{
    eth_port_t *eth_port;
    message *m_ptr;

    eth_port= ev_arg.ev_ptr;
    assert(ev == &eth_port->etp_osdep.etp_recvev);
    m_ptr= &eth_port->etp_osdep.etp_recvrepl;

    assert(m_ptr->m_type == DL_TASK_REPLY);
    assert(eth_port->etp_osdep.etp_port == m_ptr->DL_PORT &&
            eth_port->etp_osdep.etp_task == m_ptr->m_source);

    assert(m_ptr->DL_STAT & DL_PACK_RECV);
    m_ptr->DL_STAT &= ~DL_PACK_RECV;

    if (recv_debug)
    {
        printf("eth_recvev: eth%d got DL_PACK_RECV\n", m_ptr->DL_PORT);
    }

    read_int(eth_port, m_ptr->DL_COUNT);
}

```

```

PRIVATE void eth_sendev(ev, ev_arg)
event_t *ev;
ev_arg_t ev_arg;
{

```

```

    eth_port_t *eth_port;
    message *m_ptr;

    eth_port= ev_arg.ev_ptr;
    assert(ev == &eth_port->etp_sendev);
    m_ptr= &eth_port->etp_osdep.etp_sendrepl;

    assert (m_ptr->m_type == DL_TASK_REPLY);
    assert(eth_port->etp_osdep.etp_port == m_ptr->DL_PORT &&
           eth_port->etp_osdep.etp_task == m_ptr->m_source);

    assert(m_ptr->DL_STAT & DL_PACK_SEND);
    m_ptr->DL_STAT &= ~DL_PACK_SEND;

    /* packet is sent */
    write_int(eth_port);
}

PRIVATE eth_port_t *find_port(m)
message *m;
{
    eth_port_t *loc_port;
    int i;

    for (i=0, loc_port= eth_port_table; i<eth_conf_nr; i++, loc_port++)
    {
        if (loc_port->etp_osdep.etp_port == m->DL_PORT &&
            loc_port->etp_osdep.etp_task == m->m_source)
            break;
    }
    assert (i<eth_conf_nr);
    return loc_port;
}

static void eth_restart(eth_port, tasknr)
eth_port_t *eth_port;
int tasknr;
{
    int i, r;
    unsigned flags, dl_flags;
    message mess;
    eth_port_t *loc_port;

    printf("eth_restart: restarting eth%d, task %d, port %d\n",
           eth_port-eth_port_table, tasknr,
           eth_port->etp_osdep.etp_port);

    eth_port->etp_osdep.etp_task= tasknr;

    flags= eth_port->etp_osdep.etp_rcvconf;
    dl_flags= DL_NOMODE;
    if (flags & NWE0_EN_BROAD)
        dl_flags |= DL_BROAD_REQ;
    if (flags & NWE0_EN_MULTI)
        dl_flags |= DL_MULTI_REQ;
    if (flags & NWE0_EN_PROMISC)
        dl_flags |= DL_PROMISC_REQ;
    mess.m_type= DL_INIT;
    mess.DL_PORT= eth_port->etp_osdep.etp_port;
    mess.DL_PROC= this_proc;
    mess.DL_MODE= dl_flags;

    r= sendrec(eth_port->etp_osdep.etp_task, &mess);
    /* YYY */
    if (r<0)
    {
        printf(
            "eth_restart: sendrec to ethernet task %d failed: %d\n",
            eth_port->etp_osdep.etp_task, r);
        return;
    }

    if (mess.m3_il == ENXIO)
    {

```

```
        printf(
"osdep_eth_init: no ethernet device at task=%d,port=%d\n",
        eth_port->etp_osdep.etp_task,
        eth_port->etp_osdep.etp_port);
        return;
    }
    if (mess.m3_i1 < 0)
        ip_panic(("osdep_eth_init: DL_INIT returned error %d\n",
            mess.m3_i1));

    if (mess.m3_i1 != eth_port->etp_osdep.etp_port)
    {
        ip_panic(("
osdep_eth_init: got reply for wrong port (got %d, expected %d)\n",
            mess.m3_i1, eth_port->etp_osdep.etp_port));
    }

    eth_port->etp_flags |= EPF_ENABLED;

    eth_port->etp_ethaddr= *(ether_addr_t *)mess.m3_cal;
    if (!(eth_port->etp_flags & EPF_GOT_ADDR))
    {
        eth_port->etp_flags |= EPF_GOT_ADDR;
        eth_restart_ioctl(eth_port);

        /* Also update any VLANs on this device */
        for (i=0, loc_port= eth_port_table; i<eth_conf_nr;
            i++, loc_port++)
        {
            if (!(loc_port->etp_flags & EPF_ENABLED))
                continue;
            if (loc_port->etp_vlan_port != eth_port)
                continue;

            loc_port->etp_ethaddr= eth_port->etp_ethaddr;
            loc_port->etp_flags |= EPF_GOT_ADDR;
            eth_restart_ioctl(loc_port);
        }
    }

    if (eth_port->etp_wr_pack)
    {
        bf_afree(eth_port->etp_wr_pack);
        eth_port->etp_wr_pack= NULL;
        eth_restart_write(eth_port);
    }
    if (eth_port->etp_rd_pack)
    {
        bf_afree(eth_port->etp_rd_pack);
        eth_port->etp_rd_pack= NULL;
        eth_port->etp_flags &= ~(EPF_READ_IP|EPF_READ_SP);
    }
    setup_read (eth_port);
}

/*
 * $PchId: mnx_eth.c,v 1.16 2005/06/28 14:24:37 philip Exp $
 */
```

```
/*
inet/mq.c

Created:      Jan 3, 1992 by Philip Homburg

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "mq.h"
#include "generic/assert.h"

THIS_FILE

#define MQ_SIZE      128

PRIVATE mq_t mq_list[MQ_SIZE];
PRIVATE mq_t *mq_freelist;

void mq_init()
{
    int i;

    mq_freelist= NULL;
    for (i= 0; i<MQ_SIZE; i++)
    {
        mq_list[i].mq_next= mq_freelist;
        mq_freelist= &mq_list[i];
        mq_list[i].mq_allocated= 0;
    }
}

mq_t *mq_get()
{
    mq_t *mq;

    mq= mq_freelist;
    assert(mq != NULL);

    mq_freelist= mq->mq_next;
    mq->mq_next= NULL;
    assert(mq->mq_allocated == 0);
    mq->mq_allocated= 1;
    return mq;
}

void mq_free(mq)
mq_t *mq;
{
    mq->mq_next= mq_freelist;
    mq_freelist= mq;
    assert(mq->mq_allocated == 1);
    mq->mq_allocated= 0;
}

/*
 * $PchId: mq.c,v 1.7 1998/10/23 20:10:47 philip Exp $
 */
```

```
/*
inet/mq.h

Created:      Jan 3, 1992 by Philip Homburg

Copyright 1995 Philip Homburg
*/

#ifndef INET__MQ_H
#define INET__MQ_H

typedef struct mq
{
    message mq_mess;
    struct mq *mq_next;
    int mq_allocated;
} mq_t;

_PROTOTYPE( mq_t *mq_get, (void) );
_PROTOTYPE( void mq_free, (mq_t *mq) );
_PROTOTYPE( void mq_init, (void) );

#endif /* INET__MQ_H */

/*
 * $PchId: mq.h,v 1.4 1995/11/21 06:40:30 philip Exp $
 */
```

```
/*
inet/osdep_eth.h

Created:          Dec 30, 1991 by Philip Homburg

Copyright 1995 Philip Homburg
*/

#ifndef INET__OSDEP_ETH_H
#define INET__OSDEP_ETH_H

#include "generic/event.h"

#define IOVEC_NR          16
#define RD_IOVEC          ((ETH_MAX_PACK_SIZE + BUF_S - 1)/BUF_S)

typedef struct osdep_eth_port
{
    int etp_task;
    int etp_port;
    int etp_recvconf;
    iovec_t etp_wr_iovec[IOVEC_NR];
    iovec_t etp_rd_iovec[RD_IOVEC];
    event_t etp_recvev;
    message etp_sendrepl;
    message etp_recvrepl;
} osdep_eth_port_t;

#endif /* INET__OSDEP_ETH_H */

/*
 * $PchId: osdep_eth.h,v 1.6 2001/04/20 06:39:54 philip Exp $
 */
```

```
/*
inet/proto.h

Created:      Jan 2, 1992 by Philip Homburg

Copyright 1995 Philip Homburg
*/

/* clock.c */

_PROTOTYPE( void clck_tick, (message *mess) );

/* mnx_eth.c */

_PROTOTYPE( void eth_rec, (message *m) );
_PROTOTYPE( void eth_check_drivers, (message *m) );

/* sr.c */

struct mq;
_PROTOTYPE( void sr_rec, (struct mq *m) );

/*
 * $PchId: proto.h,v 1.4 1995/11/21 06:36:37 philip Exp $
 */
```



```
/*
inet/qp.c

Query parameters

Created:      June 1995 by Philip Homburg <philip@f-mnx.phicoh.com>
*/

#include "inet.h"

#include <sys/svrctl.h>
#ifdef __minix_vmd
#include <minix/queryparam.h>
#else /* Minix 3 */
#include <minix3/queryparam.h>
#endif

#include "generic/buf.h"
#include "generic/clock.h"
#include "generic/event.h"
#include "generic/type.h"
#include "generic/sr.h"

#include "generic/tcp_int.h"
#include "generic/udp_int.h"
#include "mq.h"
#include "qp.h"
#include "sr_int.h"

FORWARD int get_userdata ARGS(( int proc, vir_bytes vaddr, vir_bytes vlen,
void *buffer ));
FORWARD int put_userdata ARGS(( int proc, vir_bytes vaddr, vir_bytes vlen,
void *buffer ));
FORWARD int iqp_getc ARGS(( void ));
FORWARD void iqp_putc ARGS(( int c ));

PRIVATE struct export_param_list inet_ex_list[]=
{
    QP_VARIABLE(sr_fd_table),
    QP_VARIABLE(ip_dev),
    QP_VARIABLE(tcp_fd_table),
    QP_VARIABLE(tcp_conn_table),
    QP_VARIABLE(tcp_cancel_f),
    QP_VECTOR(udp_port_table, udp_port_table, ip_conf_nr),
    QP_VARIABLE(udp_fd_table),
    QP_END()
};

PRIVATE struct export_params inet_ex_params= { inet_ex_list, NULL };

PRIVATE struct queryvars {
    int proc;
    struct svrqueryparam qpar;
    char parbuf[256], valbuf[256];
    char *param, *value;
    int r;
} *qvars;

PUBLIC void qp_init()
{
    qp_export(&inet_ex_params);
}

PUBLIC int qp_query(proc, argp)
int proc;
vir_bytes argp;
{
    /* Return values, sizes, or addresses of variables in MM space. */

    struct queryvars qv;
    void *addr;
    size_t n, size;
    int byte;
    int more;
}
```

```

static char hex[] = "0123456789ABCDEF";

qv.r = get_userdata(proc, argp, sizeof(qv.qpar), &qv.qpar);

/* Export these to mq_getc() and mq_putc(). */
qvars = &qv;
qv.proc = proc;
qv.param = qv.parbuf + sizeof(qv.parbuf);
qv.value = qv.valbuf;

do {
    more = queryparam(iqp_getc, &addr, &size);
    for (n = 0; n < size; n++) {
        byte = ((u8_t *) addr)[n];
        iqp_putc(hex[byte >> 4]);
        iqp_putc(hex[byte & 0x0F]);
    }
    iqp_putc(more ? ',' : 0);
} while (more);
return qv.r;
}

PRIVATE int iqp_getc()
{
    /* Return one character of the names to search for. */
    struct queryvars *qv = qvars;
    size_t n;

    if (qv->r != OK || qv->qpar.psize == 0) return 0;
    if (qv->param == qv->parbuf + sizeof(qv->parbuf)) {
        /* Need to fill the parameter buffer. */
        n = sizeof(qv->parbuf);
        if (qv->qpar.psize < n) n = qv->qpar.psize;
        qv->r = get_userdata(qv->proc, (vir_bytes) qv->qpar.param, n,
                           qv->parbuf);

        if (qv->r != OK) return 0;
        qv->qpar.param += n;
        qv->param = qv->parbuf;
    }
    qv->qpar.psize--;
    return (u8_t) *qv->param++;
}

PRIVATE void iqp_putc(c)
int c;
{
    /* Send one character back to the user. */
    struct queryvars *qv = qvars;
    size_t n;

    if (qv->r != OK || qv->qpar.vsize == 0) return;
    *qv->value++ = c;
    qv->qpar.vsize--;
    if (qv->value == qv->valbuf + sizeof(qv->valbuf)
        || c == 0 || qv->qpar.vsize == 0) {
        /* Copy the value buffer to user space. */
        n = qv->value - qv->valbuf;
        qv->r = put_userdata(qv->proc, (vir_bytes) qv->qpar.value, n,
                           qv->valbuf);

        qv->qpar.value += n;
        qv->value = qv->valbuf;
    }
}

PRIVATE int get_userdata(proc, vaddr, vlen, buffer)
int proc;
vir_bytes vaddr;
vir_bytes vlen;
void *buffer;
{
#ifdef __minix_vmd
    return sys_copy(proc, SEG_D, (phys_bytes)vaddr, this_proc, SEG_D,

```

```
        (phys_bytes)buffer, (phys_bytes)vlen);
#else /* Minix 3 */
    return sys_vircopy(proc, D, vaddr, SELF, D, (vir_bytes)buffer, vlen);
#endif
}

PRIVATE int put_userdata(proc, vaddr, vlen, buffer)
int proc;
vir_bytes vaddr;
vir_bytes vlen;
void *buffer;
{
#ifdef __minix_vmd
    return sys_copy(this_proc, SEG_D, (phys_bytes)buffer,
        proc, SEG_D, (phys_bytes)vaddr, (phys_bytes)vlen);
#else /* Minix 3 */
    return sys_vircopy(SELF, D, (vir_bytes)buffer, proc, D, vaddr, vlen);
#endif
}

/*
 * $PchId: qp.c,v 1.7 2005/06/28 14:25:25 philip Exp $
 */
```

```
/*
inet/qp.h

Handle queryparams requests

Created:      June 1995 by Philip Homburg <philip@f-mnx.phicoh.com>

Copyright 1995 Philip Homburg
*/

#ifndef INET__QP_H
#define INET__QP_H

void qp_init ARGS(( void ));
int qp_query ARGS(( int proc, vir_bytes argp ));

#endif /* INET__QP_H */

/*
 * $PchId: qp.h,v 1.4 2005/01/29 18:08:06 philip Exp $
 */
```

```
/*
 * sha2.c
 *
 * Version 1.0.0beta1
 *
 * Written by Aaron D. Gifford <me@aarongifford.com>
 *
 * Copyright 2000 Aaron D. Gifford. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the copyright holder nor the names of contributors
 * may be used to endorse or promote products derived from this software
 * without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR(S) AND CONTRIBUTOR(S) ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR(S) OR CONTRIBUTOR(S) BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */

#include <sys/types.h>
/* #include <sys/time.h> */
/* #include <sys/system.h> */
/* #include <machine/endian.h> */
#include "sha2.h"

/*
 * ASSERT NOTE:
 * Some sanity checking code is included using assert(). On my FreeBSD
 * system, this additional code can be removed by compiling with NDEBUG
 * defined. Check your own systems manpage on assert() to see how to
 * compile WITHOUT the sanity checking code on your system.
 *
 * UNROLLED TRANSFORM LOOP NOTE:
 * You can define SHA2_UNROLL_TRANSFORM to use the unrolled transform
 * loop version for the hash transform rounds (defined using macros
 * later in this file). Either define on the command line, for example:
 *
 * cc -DSHA2_UNROLL_TRANSFORM -o sha2 sha2.c sha2prog.c
 *
 * or define below:
 *
 * #define SHA2_UNROLL_TRANSFORM
 */

#if defined(__bsdi__) || defined(__FreeBSD__)
#define assert(x)
#endif

/** SHA-256/384/512 Machine Architecture Definitions *****/
/*
 * SHA2_BYTE_ORDER NOTE:
 *
 * Please make sure that your system defines SHA2_BYTE_ORDER. If your
 * architecture is little-endian, make sure it also defines
 */
```

```

* SHA2_LITTLE_ENDIAN and that the two (SHA2_BYTE_ORDER and
* SHA2_LITTLE_ENDIAN) are equivalent.
*
* If your system does not define the above, then you can do so by
* hand like this:
*
* #define SHA2_LITTLE_ENDIAN 1234
* #define SHA2_BIG_ENDIAN 4321
*
* And for little-endian machines, add:
*
* #define SHA2_BYTE_ORDER SHA2_LITTLE_ENDIAN
*
* Or for big-endian machines:
*
* #define SHA2_BYTE_ORDER SHA2_BIG_ENDIAN
*
* The FreeBSD machine this was written on defines BYTE_ORDER
* appropriately by including <sys/types.h> (which in turn includes
* <machine/endian.h> where the appropriate definitions are actually
* made).
*/
#if !defined(SHA2_BYTE_ORDER) || (SHA2_BYTE_ORDER != SHA2_LITTLE_ENDIAN && SHA2_BYTE_ORDER != SHA2_BIG_ENDIAN)
#error Define SHA2_BYTE_ORDER to be equal to either SHA2_LITTLE_ENDIAN or SHA2_BIG_ENDIAN
#endif

/*
* Define the followingsha2_* types to types of the correct length on
* the native architecture. Most BSD systems and Linux define u_intXX_t
* types. Machines with very recent ANSI C headers, can use the
* uintXX_t definitions from inttypes.h by defining SHA2_USE_INTTYPES_H
* during compile or in the sha.h header file.
*
* Machines that support neither u_intXX_t nor inttypes.h's uintXX_t
* will need to define these three typedefs below (and the appropriate
* ones in sha.h too) by hand according to their system architecture.
*
* Thank you, Jun-ichiro itojun Hagino, for suggesting using u_intXX_t
* types and pointing out recent ANSI C support for uintXX_t in inttypes.h.
*/
#if 0 /*def SHA2_USE_INTTYPES_H*/

typedef uint8_t sha2_byte; /* Exactly 1 byte */
typedef uint32_t sha2_word32; /* Exactly 4 bytes */
typedef uint64_t sha2_word64; /* Exactly 8 bytes */

#else /* SHA2_USE_INTTYPES_H */

typedef u_int8_t sha2_byte; /* Exactly 1 byte */
typedef u_int32_t sha2_word32; /* Exactly 4 bytes */
typedef u_int64_t sha2_word64; /* Exactly 8 bytes */

#endif /* SHA2_USE_INTTYPES_H */

/** SHA-256/384/512 Various Length Definitions *****/
/* NOTE: Most of these are in sha2.h */
#define SHA256_SHORT_BLOCK_LENGTH (SHA256_BLOCK_LENGTH - 8)
#define SHA384_SHORT_BLOCK_LENGTH (SHA384_BLOCK_LENGTH - 16)
#define SHA512_SHORT_BLOCK_LENGTH (SHA512_BLOCK_LENGTH - 16)

/** ENDIAN REVERSAL MACROS *****/
#if SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN
#define REVERSE32(w,x) { \
    sha2_word32 tmp = (w); \
    tmp = (tmp >> 16) | (tmp << 16); \
    (x) = ((tmp & 0xff00ff00UL) >> 8) | ((tmp & 0x00ff00ffUL) << 8); \
}
#define REVERSE64(w,x) { \
    sha2_word64 tmp = (w); \
    tmp = (tmp >> 32) | (tmp << 32); \
    tmp = ((tmp & 0xff00ff00ff00ff00ULL) >> 8) | \

```

```

        ((tmp & 0x00ff00ff00ff00ffULL) << 8); \
        (x) = ((tmp & 0xffff0000ffff0000ULL) >> 16) | \
        ((tmp & 0x0000ffff0000ffffULL) << 16); \
    }
#ifdef MINIX_64BIT
#undef REVERSE64
#define REVERSE64(w,x) { \
    u32_t hi, lo; \
    REVERSE32(ex64hi((w)), lo); \
    REVERSE32(ex64lo((w)), hi); \
    (x) = make64(lo, hi); \
}
#endif /* MINIX_64BIT */
#endif /* SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN */

/*
 * Macro for incrementally adding the unsigned 64-bit integer n to the
 * unsigned 128-bit integer (represented using a two-element array of
 * 64-bit words):
 */
#define ADDINC128(w,n) { \
    (w)[0] += (sha2_word64)(n); \
    if ((w)[0] < (n)) { \
        (w)[1]++; \
    } \
}

/** THE SIX LOGICAL FUNCTIONS *****/
/*
 * Bit shifting and rotation (used by the six SHA-XYZ logical functions:
 *
 * NOTE: The naming of R and S appears backwards here (R is a SHIFT and
 * S is a ROTATION) because the SHA-256/384/512 description document
 * (see http://csrc.nist.gov/cryptval/shs/sha256-384-512.pdf) uses this
 * same "backwards" definition.
 */
/* Shift-right (used in SHA-256, SHA-384, and SHA-512): */
#define R(b,x) ((x) >> (b))
/* 32-bit Rotate-right (used in SHA-256): */
#define S32(b,x) (((x) >> (b)) | ((x) << (32 - (b))))
/* 64-bit Rotate-right (used in SHA-384 and SHA-512): */
#define S64(b,x) (((x) >> (b)) | ((x) << (64 - (b))))

/* Two of six logical functions used in SHA-256, SHA-384, and SHA-512: */
#define Ch(x,y,z) (((x) & (y)) ^ ((~(x)) & (z)))
#define Maj(x,y,z) (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))

/* Four of six logical functions used in SHA-256: */
#define Sigma0_256(x) (S32(2, (x)) ^ S32(13, (x)) ^ S32(22, (x)))
#define Sigma1_256(x) (S32(6, (x)) ^ S32(11, (x)) ^ S32(25, (x)))
#define sigma0_256(x) (S32(7, (x)) ^ S32(18, (x)) ^ R(3, (x)))
#define sigma1_256(x) (S32(17, (x)) ^ S32(19, (x)) ^ R(10, (x)))

/* Four of six logical functions used in SHA-384 and SHA-512: */
#define Sigma0_512(x) (S64(28, (x)) ^ S64(34, (x)) ^ S64(39, (x)))
#define Sigma1_512(x) (S64(14, (x)) ^ S64(18, (x)) ^ S64(41, (x)))
#define sigma0_512(x) (S64( 1, (x)) ^ S64( 8, (x)) ^ R( 7, (x)))
#define sigma1_512(x) (S64(19, (x)) ^ S64(61, (x)) ^ R( 6, (x)))

/** INTERNAL FUNCTION PROTOTYPES *****/
/* NOTE: These should not be accessed directly from outside this
 * library -- they are intended for private internal visibility/use
 * only.
 */
void SHA512_Last(SHA512_CTX*);
void SHA256_Transform(SHA256_CTX*, const sha2_word32*);
void SHA512_Transform(SHA512_CTX*, const sha2_word64*);

/** SHA-XYZ INITIAL HASH VALUES AND CONSTANTS *****/
/* Hash constant words K for SHA-256: */
const static sha2_word32 K256[64] = {
    0x428a2f98UL, 0x71374491UL, 0xb5c0fbcfUL, 0xe9b5dba5UL,
    0x3956c25bUL, 0x59f111f1UL, 0x923f82a4UL, 0xab1c5ed5UL,

```

```
    0xd807aa98UL, 0x12835b01UL, 0x243185beUL, 0x550c7dc3UL,
    0x72be5d74UL, 0x80deb1feUL, 0x9bdc06a7UL, 0xc19bf174UL,
    0xe49b69c1UL, 0xefbe4786UL, 0x0fc19dc6UL, 0x240ca1ccUL,
    0x2de92c6fUL, 0x4a7484aaUL, 0x5cb0a9dcUL, 0x76f988daUL,
    0x983e5152UL, 0xa831c66dUL, 0xb00327c8UL, 0xbf597fc7UL,
    0xc6e00bf3UL, 0xd5a79147UL, 0x06ca6351UL, 0x14292967UL,
    0x27b70a85UL, 0x2e1b2138UL, 0x4d2c6dfcUL, 0x53380d13UL,
    0x650a7354UL, 0x766a0abbUL, 0x81c2c92eUL, 0x92722c85UL,
    0xa2bfe8a1UL, 0xa81a664bUL, 0xc24b8b70UL, 0xc76c51a3UL,
    0xd192e819UL, 0xd6990624UL, 0xf40e3585UL, 0x106aa070UL,
    0x19a4c116UL, 0x1e376c08UL, 0x2748774cUL, 0x34b0bcb5UL,
    0x391c0cb3UL, 0x4ed8aa4aUL, 0x5b9cca4fUL, 0x682e6fff3UL,
    0x748f82eeUL, 0x78a5636fUL, 0x84c87814UL, 0x8cc70208UL,
    0x90beffffaUL, 0xa4506cebUL, 0xbef9a3f7UL, 0xc67178f2UL
};

/* Initial hash value H for SHA-256: */
const static sha2_word32 sha256_initial_hash_value[8] = {
    0x6a09e667UL,
    0xbb67ae85UL,
    0x3c6ef372UL,
    0xa54ff53aUL,
    0x510e527fUL,
    0x9b05688cUL,
    0x1f83d9abUL,
    0x5be0cd19UL
};

#if !NO_64BIT
/* Hash constant words K for SHA-384 and SHA-512: */
const static sha2_word64 K512[80] = {
    0x428a2f98d728ae22ULL, 0x7137449123ef65cdULL,
    0xb5c0fbcfec4d3b2fULL, 0xe9b5dba58189dbbcULL,
    0x3956c25bf348b538ULL, 0x59f111f1b605d019ULL,
    0x923f82a4af194f9bULL, 0xab1c5ed5da6d8118ULL,
    0xd807aa98a3030242ULL, 0x12835b0145706fbeULL,
    0x243185be4ee4b28cULL, 0x550c7dc3d5fffb4eULL,
    0x72be5d74f27b896fULL, 0x80deb1fe3b1696b1ULL,
    0x9bdc06a725c71235ULL, 0xc19bf174cf692694ULL,
    0xe49b69c19ef14ad2ULL, 0xefbe4786384f25e3ULL,
    0x0fc19dc68b8cd5b5ULL, 0x240ca1cc77ac9c65ULL,
    0x2de92c6f592b0275ULL, 0x4a7484aa6eae6483ULL,
    0x5cb0a9dcdb41fbd4ULL, 0x76f988da831153b5ULL,
    0x983e5152ee66dfabULL, 0xa831c66d2db43210ULL,
    0xb00327c898fb213fULL, 0xbf597fc7beef0ee4ULL,
    0xc6e00bf33da88fc2ULL, 0xd5a79147930aa725ULL,
    0x06ca6351e003826fULL, 0x142929670a0e6e70ULL,
    0x27b70a8546d22ffcULL, 0x2e1b21385c26c926ULL,
    0x4d2c6dfc5ac42aedULL, 0x53380d139d95b3dfULL,
    0x650a73548baf63deULL, 0x766a0abb3c77b2a8ULL,
    0x81c2c92e47edaee6ULL, 0x92722c851482353bULL,
    0xa2bfe8a14cf10364ULL, 0xa81a664bbc423001ULL,
    0xc24b8b70d0f89791ULL, 0xc76c51a30654be30ULL,
    0xd192e819d6ef5218ULL, 0xd69906245565a910ULL,
    0xf40e35855771202aULL, 0x106aa07032bbd1b8ULL,
    0x19a4c116b8d2d0c8ULL, 0x1e376c085141ab53ULL,
    0x2748774cdf8eeb99ULL, 0x34b0bcb5e19b48a8ULL,
    0x391c0cb3c5c95a63ULL, 0x4ed8aa4ae3418acbULL,
    0x5b9cca4f7763e373ULL, 0x682e6fff3d6b2b8a3ULL,
    0x748f82ee5defb2fcULL, 0x78a5636f43172f60ULL,
    0x84c87814a1f0ab72ULL, 0x8cc702081a6439ecULL,
    0x90beffffa23631e28ULL, 0xa4506cebde82bde9ULL,
    0xbef9a3f7b2c67915ULL, 0xc67178f2e372532bULL,
    0xca273eceea26619cULL, 0xd186b8c721c0c207ULL,
    0xeadad7dd6cde0eb1eULL, 0xf57d4f7fee6ed178ULL,
    0x06f067aa72176fbaULL, 0x0a637dc5a2c898a6ULL,
    0x113f9804bef90daeULL, 0x1b710b35131c471bULL,
    0x28db77f523047d84ULL, 0x32caab7b40c72493ULL,
    0x3c9ebe0a15c9bebcULL, 0x431d67c49c100d4cULL,
    0x4cc5d4becb3e42b6ULL, 0x597f299cfc657e2aULL,
    0x5fcb6fab3ad6faecULL, 0x6c44198c4a475817ULL
};

/* Initial hash value H for SHA-384 */
```



```
const static sha2_word64 sha384_initial_hash_value[8] = {
    0xcbbb9d5dc1059ed8ULL,
    0x629a292a367cd507ULL,
    0x9159015a3070dd17ULL,
    0x152fec8f70e5939ULL,
    0x67332667ffc00b31ULL,
    0x8eb44a8768581511ULL,
    0xdb0c2e0d64f98fa7ULL,
    0x47b5481dbefa4fa4ULL
};

/* Initial hash value H for SHA-512 */
const static sha2_word64 sha512_initial_hash_value[8] = {
    0x6a09e667f3bcc908ULL,
    0xbb67ae8584caa73bULL,
    0x3c6ef372fe94f82bULL,
    0xa54fff53a5f1d36f1ULL,
    0x510e527fade682d1ULL,
    0x9b05688c2b3e6c1fULL,
    0x1f83d9abfb41bd6bULL,
    0x5be0cd19137e2179ULL
};
#endif /* !NO_64BIT */

/*
 * Constant used by SHA256/384/512_End() functions for converting the
 * digest to a readable hexadecimal character string:
 */
static const char *sha2_hex_digits = "0123456789abcdef";

**** SHA-256: ****
void SHA256_Init(SHA256_CTX* context) {
    if (context == (SHA256_CTX*)0) {
        return;
    }
    bcopy(sha256_initial_hash_value, context->state, SHA256_DIGEST_LENGTH);
    bzero(context->buffer, SHA256_BLOCK_LENGTH);
#if MINIX_64BIT
    context->bitcount = cvu64(0);
#else /* !MINIX_64BIT */
    context->bitcount = 0;
#endif /* MINIX_64BIT */
}

#ifdef SHA2_UNROLL_TRANSFORM

/* Unrolled SHA-256 round macros: */

#if SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN

#define ROUND256_0_TO_15(a,b,c,d,e,f,g,h) \
    REVERSE32(*data++, W256[j]); \
    T1 = (h) + Sigma1_256(e) + Ch((e), (f), (g)) + \
        K256[j] + W256[j]; \
    (d) += T1; \
    (h) = T1 + Sigma0_256(a) + Maj((a), (b), (c)); \
    j++

#else /* SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN */

#define ROUND256_0_TO_15(a,b,c,d,e,f,g,h) \
    T1 = (h) + Sigma1_256(e) + Ch((e), (f), (g)) + \
        K256[j] + (W256[j] = *data++); \
    (d) += T1; \
    (h) = T1 + Sigma0_256(a) + Maj((a), (b), (c)); \
    j++

#endif /* SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN */

#define ROUND256(a,b,c,d,e,f,g,h) \
    s0 = W256[(j+1)&0x0f]; \
    s0 = sigma0_256(s0); \
```

```

    s1 = W256[(j+14)&0x0f]; \
    s1 = sigma1_256(s1); \
    T1 = (h) + Sigma1_256(e) + Ch((e), (f), (g)) + K256[j] + \
        (W256[j&0x0f] += s1 + W256[(j+9)&0x0f] + s0); \
    (d) += T1; \
    (h) = T1 + Sigma0_256(a) + Maj((a), (b), (c)); \
    j++;

void SHA256_Transform(SHA256_CTX* context, const sha2_word32* data) {
    sha2_word32    a, b, c, d, e, f, g, h, s0, s1;
    sha2_word32    T1, *W256;
    int            j;

    W256 = (sha2_word32*)context->buffer;

    /* Initialize registers with the prev. intermediate value */
    a = context->state[0];
    b = context->state[1];
    c = context->state[2];
    d = context->state[3];
    e = context->state[4];
    f = context->state[5];
    g = context->state[6];
    h = context->state[7];

    j = 0;
    do {
        /* Rounds 0 to 15 (unrolled): */
        ROUND256_0_TO_15(a,b,c,d,e,f,g,h);
        ROUND256_0_TO_15(h,a,b,c,d,e,f,g);
        ROUND256_0_TO_15(g,h,a,b,c,d,e,f);
        ROUND256_0_TO_15(f,g,h,a,b,c,d,e);
        ROUND256_0_TO_15(e,f,g,h,a,b,c,d);
        ROUND256_0_TO_15(d,e,f,g,h,a,b,c);
        ROUND256_0_TO_15(c,d,e,f,g,h,a,b);
        ROUND256_0_TO_15(b,c,d,e,f,g,h,a);
    } while (j < 16);

    /* Now for the remaining rounds to 64: */
    do {
        ROUND256(a,b,c,d,e,f,g,h);
        ROUND256(h,a,b,c,d,e,f,g);
        ROUND256(g,h,a,b,c,d,e,f);
        ROUND256(f,g,h,a,b,c,d,e);
        ROUND256(e,f,g,h,a,b,c,d);
        ROUND256(d,e,f,g,h,a,b,c);
        ROUND256(c,d,e,f,g,h,a,b);
        ROUND256(b,c,d,e,f,g,h,a);
    } while (j < 64);

    /* Compute the current intermediate hash value */
    context->state[0] += a;
    context->state[1] += b;
    context->state[2] += c;
    context->state[3] += d;
    context->state[4] += e;
    context->state[5] += f;
    context->state[6] += g;
    context->state[7] += h;

    /* Clean up */
    a = b = c = d = e = f = g = h = T1 = 0;
}

#else /* SHA2_UNROLL_TRANSFORM */

void SHA256_Transform(SHA256_CTX* context, const sha2_word32* data) {
    sha2_word32    a, b, c, d, e, f, g, h, s0, s1;
    sha2_word32    T1, T2, *W256;
    int            j;

    W256 = (sha2_word32*)context->buffer;

    /* Initialize registers with the prev. intermediate value */

```

```

    a = context->state[0];
    b = context->state[1];
    c = context->state[2];
    d = context->state[3];
    e = context->state[4];
    f = context->state[5];
    g = context->state[6];
    h = context->state[7];

    j = 0;
    do {
#ifdef SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN
        /* Copy data while converting to host byte order */
        REVERSE32(*data++, W256[j]);
        /* Apply the SHA-256 compression function to update a..h */
        T1 = h + Sigma1_256(e) + Ch(e, f, g) + K256[j] + W256[j];
#else /* SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN */
        /* Apply the SHA-256 compression function to update a..h with copy */
        T1 = h + Sigma1_256(e) + Ch(e, f, g) + K256[j] + (W256[j] = *data++);
#endif /* SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN */
        T2 = Sigma0_256(a) + Maj(a, b, c);
        h = g;
        g = f;
        f = e;
        e = d + T1;
        d = c;
        c = b;
        b = a;
        a = T1 + T2;

        j++;
    } while (j < 16);

    do {
        /* Part of the message block expansion: */
        s0 = W256[(j+1)&0x0f];
        s0 = sigma0_256(s0);
        s1 = W256[(j+14)&0x0f];
        s1 = sigma1_256(s1);

        /* Apply the SHA-256 compression function to update a..h */
        T1 = h + Sigma1_256(e) + Ch(e, f, g) + K256[j] +
            (W256[j&0x0f] += s1 + W256[(j+9)&0x0f] + s0);
        T2 = Sigma0_256(a) + Maj(a, b, c);
        h = g;
        g = f;
        f = e;
        e = d + T1;
        d = c;
        c = b;
        b = a;
        a = T1 + T2;

        j++;
    } while (j < 64);

    /* Compute the current intermediate hash value */
    context->state[0] += a;
    context->state[1] += b;
    context->state[2] += c;
    context->state[3] += d;
    context->state[4] += e;
    context->state[5] += f;
    context->state[6] += g;
    context->state[7] += h;

    /* Clean up */
    a = b = c = d = e = f = g = h = T1 = T2 = 0;
}

#endif /* SHA2_UNROLL_TRANSFORM */

void SHA256_Update(SHA256_CTX* context, const sha2_byte *data, size_t len) {
    unsigned int    freespace, usedspace;

```

```

    if (len == 0) {
        /* Calling with no data is valid - we do nothing */
        return;
    }

    /* Sanity check: */
    assert(context != (SHA256_CTX*)0 && data != (sha2_byte*)0);

#if MINIX_64BIT
    usedspace= rem64u(context->bitcount, SHA256_BLOCK_LENGTH*8)/8;
#else /* !MINIX_64BIT */
    usedspace = (context->bitcount >> 3) % SHA256_BLOCK_LENGTH;
#endif /* MINIX_64BIT */
    if (usedspace > 0) {
        /* Calculate how much free space is available in the buffer */
        freespace = SHA256_BLOCK_LENGTH - usedspace;

        if (len >= freespace) {
            /* Fill the buffer completely and process it */
            bcopy(data, &context->buffer[usedspace], freespace);
#if MINIX_64BIT
            context->bitcount= add64u(context->bitcount,
                                     freespace << 3);
#else /* !MINIX_64BIT */
            context->bitcount += freespace << 3;
#endif /* MINIX_64BIT */
            len -= freespace;
            data += freespace;
            SHA256_Transform(context, (sha2_word32*)context->buffer);
        } else {
            /* The buffer is not yet full */
            bcopy(data, &context->buffer[usedspace], len);
#if MINIX_64BIT
            context->bitcount= add64u(context->bitcount, len << 3);
#else /* !MINIX_64BIT */
            context->bitcount += len << 3;
#endif /* MINIX_64BIT */
            /* Clean up: */
            usedspace = freespace = 0;
            return;
        }
    }

    while (len >= SHA256_BLOCK_LENGTH) {
        /* Process as many complete blocks as we can */
        SHA256_Transform(context, (const sha2_word32*)data);
#if MINIX_64BIT
        context->bitcount= add64u(context->bitcount,
                                SHA256_BLOCK_LENGTH << 3);
#else /* !MINIX_64BIT */
        context->bitcount += SHA256_BLOCK_LENGTH << 3;
#endif /* MINIX_64BIT */
        len -= SHA256_BLOCK_LENGTH;
        data += SHA256_BLOCK_LENGTH;
    }

    if (len > 0) {
        /* There's left-overs, so save 'em */
        bcopy(data, context->buffer, len);
#if MINIX_64BIT
        context->bitcount= add64u(context->bitcount, len << 3);
#else /* !MINIX_64BIT */
        context->bitcount += len << 3;
#endif /* MINIX_64BIT */
    }
    /* Clean up: */
    usedspace = freespace = 0;
}

void SHA256_Final(sha2_byte digest[], SHA256_CTX* context) {
    sha2_word32    *d = (sha2_word32*)digest;
    unsigned int    usedspace;

    /* Sanity check: */
    assert(context != (SHA256_CTX*)0);

```

```

        /* If no digest buffer is passed, we don't bother doing this: */
        if (digest != (sha2_byte*)0) {
# if MINIX_64BIT
            usedspace= rem64u(context->bitcount, SHA256_BLOCK_LENGTH*8);
# else /* !MINIX_64BIT */
            usedspace = (context->bitcount >> 3) % SHA256_BLOCK_LENGTH;
# endif /* MINIX_64BIT */
# if SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN
            /* Convert FROM host byte order */
            REVERSE64(context->bitcount, context->bitcount);
# endif

            if (usedspace > 0) {
                /* Begin padding with a 1 bit: */
                context->buffer[usedspace++] = 0x80;

                if (usedspace <= SHA256_SHORT_BLOCK_LENGTH) {
                    /* Set-up for the last transform: */
                    bzero(&context->buffer[usedspace], SHA256_SHORT_BLOCK_LEN
GTH - usedspace);
                } else {
                    if (usedspace < SHA256_BLOCK_LENGTH) {
                        bzero(&context->buffer[usedspace], SHA256_BLOCK_L
ENGTH - usedspace);
                    }
                    /* Do second-to-last transform: */
                    SHA256_Transform(context, (sha2_word32*)context->buffer);

                    /* And set-up for the last transform: */
                    bzero(context->buffer, SHA256_SHORT_BLOCK_LENGTH);
                }
            } else {
                /* Set-up for the last transform: */
                bzero(context->buffer, SHA256_SHORT_BLOCK_LENGTH);

                /* Begin padding with a 1 bit: */
                *context->buffer = 0x80;
            }
            /* Set the bit count: */
            *(sha2_word64*)&context->buffer[SHA256_SHORT_BLOCK_LENGTH] = context->bit
count;

            /* Final transform: */
            SHA256_Transform(context, (sha2_word32*)context->buffer);
# if SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN
            {
                /* Convert TO host byte order */
                int j;
                for (j = 0; j < 8; j++) {
                    REVERSE32(context->state[j], context->state[j]);
                    *d++ = context->state[j];
                }
            }
# else
            bcopy(context->state, d, SHA256_DIGEST_LENGTH);
# endif
        }

        /* Clean up state data: */
        bzero(context, sizeof(context));
        usedspace = 0;
    }

char *SHA256_End(SHA256_CTX* context, char buffer[]) {
    sha2_byte digest[SHA256_DIGEST_LENGTH], *d = digest;
    int i;

    /* Sanity check: */
    assert(context != (SHA256_CTX*)0);

    if (buffer != (char*)0) {
        SHA256_Final(digest, context);
    }

```

```

        for (i = 0; i < SHA256_DIGEST_LENGTH; i++) {
            *buffer++ = sha2_hex_digits[(*d & 0xf0) >> 4];
            *buffer++ = sha2_hex_digits[*d & 0x0f];
            d++;
        }
        *buffer = (char)0;
    } else {
        bzero(context, sizeof(context));
    }
    bzero(digest, SHA256_DIGEST_LENGTH);
    return buffer;
}

char* SHA256_Data(const sha2_byte* data, size_t len, char digest[SHA256_DIGEST_STRING_LENGTH]) {
    SHA256_CTX      context;

    SHA256_Init(&context);
    SHA256_Update(&context, data, len);
    return SHA256_End(&context, digest);
}

#ifdef NO_64BIT

/** SHA-512: *****/
void SHA512_Init(SHA512_CTX* context) {
    if (context == (SHA512_CTX*)0) {
        return;
    }
    bcopy(sha512_initial_hash_value, context->state, SHA512_DIGEST_LENGTH);
    bzero(context->buffer, SHA512_BLOCK_LENGTH);
    context->bitcount[0] = context->bitcount[1] = 0;
}

#ifdef SHA2_UNROLL_TRANSFORM

/* Unrolled SHA-512 round macros: */
#if SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN

#define ROUND512_0_TO_15(a,b,c,d,e,f,g,h) \
    REVERSE64(*data++, W512[j]); \
    T1 = (h) + Sigma1_512(e) + Ch((e), (f), (g)) + \
        K512[j] + W512[j]; \
    (d) += T1, \
    (h) = T1 + Sigma0_512(a) + Maj((a), (b), (c)), \
    j++

#else /* SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN */

#define ROUND512_0_TO_15(a,b,c,d,e,f,g,h) \
    T1 = (h) + Sigma1_512(e) + Ch((e), (f), (g)) + \
        K512[j] + (W512[j] = *data++); \
    (d) += T1; \
    (h) = T1 + Sigma0_512(a) + Maj((a), (b), (c)); \
    j++

#endif /* SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN */

#define ROUND512(a,b,c,d,e,f,g,h) \
    s0 = W512[(j+1)&0x0f]; \
    s0 = sigma0_512(s0); \
    s1 = W512[(j+14)&0x0f]; \
    s1 = sigma1_512(s1); \
    T1 = (h) + Sigma1_512(e) + Ch((e), (f), (g)) + K512[j] + \
        (W512[j&0x0f] += s1 + W512[(j+9)&0x0f] + s0); \
    (d) += T1; \
    (h) = T1 + Sigma0_512(a) + Maj((a), (b), (c)); \
    j++

void SHA512_Transform(SHA512_CTX* context, const sha2_word64* data) {
    sha2_word64    a, b, c, d, e, f, g, h, s0, s1;
    sha2_word64    T1, *W512 = (sha2_word64*)context->buffer;
    int            j;

```

```

/* Initialize registers with the prev. intermediate value */
a = context->state[0];
b = context->state[1];
c = context->state[2];
d = context->state[3];
e = context->state[4];
f = context->state[5];
g = context->state[6];
h = context->state[7];

j = 0;
do {
    ROUND512_0_TO_15(a,b,c,d,e,f,g,h);
    ROUND512_0_TO_15(h,a,b,c,d,e,f,g);
    ROUND512_0_TO_15(g,h,a,b,c,d,e,f);
    ROUND512_0_TO_15(f,g,h,a,b,c,d,e);
    ROUND512_0_TO_15(e,f,g,h,a,b,c,d);
    ROUND512_0_TO_15(d,e,f,g,h,a,b,c);
    ROUND512_0_TO_15(c,d,e,f,g,h,a,b);
    ROUND512_0_TO_15(b,c,d,e,f,g,h,a);
} while (j < 16);

/* Now for the remaining rounds up to 79: */
do {
    ROUND512(a,b,c,d,e,f,g,h);
    ROUND512(h,a,b,c,d,e,f,g);
    ROUND512(g,h,a,b,c,d,e,f);
    ROUND512(f,g,h,a,b,c,d,e);
    ROUND512(e,f,g,h,a,b,c,d);
    ROUND512(d,e,f,g,h,a,b,c);
    ROUND512(c,d,e,f,g,h,a,b);
    ROUND512(b,c,d,e,f,g,h,a);
} while (j < 80);

/* Compute the current intermediate hash value */
context->state[0] += a;
context->state[1] += b;
context->state[2] += c;
context->state[3] += d;
context->state[4] += e;
context->state[5] += f;
context->state[6] += g;
context->state[7] += h;

/* Clean up */
a = b = c = d = e = f = g = h = T1 = 0;
}

#else /* SHA2_UNROLL_TRANSFORM */

void SHA512_Transform(SHA512_CTX* context, const sha2_word64* data) {
    sha2_word64    a, b, c, d, e, f, g, h, s0, s1;
    sha2_word64    T1, T2, *W512 = (sha2_word64*)context->buffer;
    int            j;

    /* Initialize registers with the prev. intermediate value */
    a = context->state[0];
    b = context->state[1];
    c = context->state[2];
    d = context->state[3];
    e = context->state[4];
    f = context->state[5];
    g = context->state[6];
    h = context->state[7];

    j = 0;
    do {
#if SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN
        /* Convert TO host byte order */
        REVERSE64(*data++, W512[j]);
        /* Apply the SHA-512 compression function to update a..h */
        T1 = h + Sigmal_512(e) + Ch(e, f, g) + K512[j] + W512[j];
#else /* SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN */

```

```

        /* Apply the SHA-512 compression function to update a..h with copy */
        T1 = h + Sigma1_512(e) + Ch(e, f, g) + K512[j] + (W512[j] = *data++);
#endif /* SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN */
        T2 = Sigma0_512(a) + Maj(a, b, c);
        h = g;
        g = f;
        f = e;
        e = d + T1;
        d = c;
        c = b;
        b = a;
        a = T1 + T2;

        j++;
    } while (j < 16);

do {
    /* Part of the message block expansion: */
    s0 = W512[(j+1)&0x0f];
    s0 = sigma0_512(s0);
    s1 = W512[(j+14)&0x0f];
    s1 = sigma1_512(s1);

    /* Apply the SHA-512 compression function to update a..h */
    T1 = h + Sigma1_512(e) + Ch(e, f, g) + K512[j] +
        (W512[j&0x0f] += s1 + W512[(j+9)&0x0f] + s0);
    T2 = Sigma0_512(a) + Maj(a, b, c);
    h = g;
    g = f;
    f = e;
    e = d + T1;
    d = c;
    c = b;
    b = a;
    a = T1 + T2;

    j++;
} while (j < 80);

/* Compute the current intermediate hash value */
context->state[0] += a;
context->state[1] += b;
context->state[2] += c;
context->state[3] += d;
context->state[4] += e;
context->state[5] += f;
context->state[6] += g;
context->state[7] += h;

/* Clean up */
a = b = c = d = e = f = g = h = T1 = T2 = 0;
}

#endif /* SHA2_UNROLL_TRANSFORM */

void SHA512_Update(SHA512_CTX* context, const sha2_byte *data, size_t len) {
    unsigned int    freespace, usedspace;

    if (len == 0) {
        /* Calling with no data is valid - we do nothing */
        return;
    }

    /* Sanity check: */
    assert(context != (SHA512_CTX*)0 && data != (sha2_byte*)0);

    usedspace = (context->bitcount[0] >> 3) % SHA512_BLOCK_LENGTH;
    if (usedspace > 0) {
        /* Calculate how much free space is available in the buffer */
        freespace = SHA512_BLOCK_LENGTH - usedspace;

        if (len >= freespace) {
            /* Fill the buffer completely and process it */
            bcopy(data, &context->buffer[usedspace], freespace);

```



```

        ADDINC128(context->bitcount, freespace << 3);
        len -= freespace;
        data += freespace;
        SHA512_Transform(context, (sha2_word64*)context->buffer);
    } else {
        /* The buffer is not yet full */
        bcopy(data, &context->buffer[usedspace], len);
        ADDINC128(context->bitcount, len << 3);
        /* Clean up: */
        usedspace = freespace = 0;
        return;
    }
}
while (len >= SHA512_BLOCK_LENGTH) {
    /* Process as many complete blocks as we can */
    SHA512_Transform(context, (const sha2_word64*)data);
    ADDINC128(context->bitcount, SHA512_BLOCK_LENGTH << 3);
    len -= SHA512_BLOCK_LENGTH;
    data += SHA512_BLOCK_LENGTH;
}
if (len > 0) {
    /* There's left-overs, so save 'em */
    bcopy(data, context->buffer, len);
    ADDINC128(context->bitcount, len << 3);
}
/* Clean up: */
usedspace = freespace = 0;
}

void SHA512_Last(SHA512_CTX* context) {
    unsigned int    usedspace;

    usedspace = (context->bitcount[0] >> 3) % SHA512_BLOCK_LENGTH;
#ifdef SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN
    /* Convert FROM host byte order */
    REVERSE64(context->bitcount[0], context->bitcount[0]);
    REVERSE64(context->bitcount[1], context->bitcount[1]);
#endif
    if (usedspace > 0) {
        /* Begin padding with a 1 bit: */
        context->buffer[usedspace++] = 0x80;

        if (usedspace <= SHA512_SHORT_BLOCK_LENGTH) {
            /* Set-up for the last transform: */
            bzero(&context->buffer[usedspace], SHA512_SHORT_BLOCK_LENGTH - us
edgespace);
        } else {
            if (usedspace < SHA512_BLOCK_LENGTH) {
                bzero(&context->buffer[usedspace], SHA512_BLOCK_LENGTH -
usedspace);
            }
            /* Do second-to-last transform: */
            SHA512_Transform(context, (sha2_word64*)context->buffer);

            /* And set-up for the last transform: */
            bzero(context->buffer, SHA512_BLOCK_LENGTH - 2);
        }
    } else {
        /* Prepare for final transform: */
        bzero(context->buffer, SHA512_SHORT_BLOCK_LENGTH);

        /* Begin padding with a 1 bit: */
        *context->buffer = 0x80;
    }
    /* Store the length of input data (in bits): */
    *(sha2_word64*)&context->buffer[SHA512_SHORT_BLOCK_LENGTH] = context->bitcount[1]
;
    *(sha2_word64*)&context->buffer[SHA512_SHORT_BLOCK_LENGTH+8] = context->bitcount[
0];

    /* Final transform: */
    SHA512_Transform(context, (sha2_word64*)context->buffer);
}

```

```

void SHA512_Final(sha2_byte digest[], SHA512_CTX* context) {
    sha2_word64    *d = (sha2_word64*)digest;

    /* Sanity check: */
    assert(context != (SHA512_CTX*)0);

    /* If no digest buffer is passed, we don't bother doing this: */
    if (digest != (sha2_byte*)0) {
        SHA512_Last(context);

        /* Save the hash data for output: */
#ifdef SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN
        {
            /* Convert TO host byte order */
            int    j;
            for (j = 0; j < 8; j++) {
                REVERSE64(context->state[j], context->state[j]);
                *d++ = context->state[j];
            }
        }
#else
        bcopy(context->state, d, SHA512_DIGEST_LENGTH);
#endif
    }

    /* Zero out state data */
    bzero(context, sizeof(context));
}

char *SHA512_End(SHA512_CTX* context, char buffer[]) {
    sha2_byte    digest[SHA512_DIGEST_LENGTH], *d = digest;
    int          i;

    /* Sanity check: */
    assert(context != (SHA512_CTX*)0);

    if (buffer != (char*)0) {
        SHA512_Final(digest, context);

        for (i = 0; i < SHA512_DIGEST_LENGTH; i++) {
            *buffer++ = sha2_hex_digits[(*d & 0xf0) >> 4];
            *buffer++ = sha2_hex_digits[*d & 0x0f];
            d++;
        }
        *buffer = (char)0;
    } else {
        bzero(context, sizeof(context));
    }
    bzero(digest, SHA512_DIGEST_LENGTH);
    return buffer;
}

char* SHA512_Data(const sha2_byte* data, size_t len, char digest[SHA512_DIGEST_STRING_LENGTH]) {
    SHA512_CTX    context;

    SHA512_Init(&context);
    SHA512_Update(&context, data, len);
    return SHA512_End(&context, digest);
}

/** SHA-384: *****/
void SHA384_Init(SHA384_CTX* context) {
    if (context == (SHA384_CTX*)0) {
        return;
    }
    bcopy(sha384_initial_hash_value, context->state, SHA512_DIGEST_LENGTH);
    bzero(context->buffer, SHA384_BLOCK_LENGTH);
    context->bitcount[0] = context->bitcount[1] = 0;
}

void SHA384_Update(SHA384_CTX* context, const sha2_byte* data, size_t len) {
    SHA512_Update((SHA512_CTX*)context, data, len);
}

```

```

}

void SHA384_Final(sha2_byte digest[], SHA384_CTX* context) {
    sha2_word64    *d = (sha2_word64*)digest;

    /* Sanity check: */
    assert(context != (SHA384_CTX*)0);

    /* If no digest buffer is passed, we don't bother doing this: */
    if (digest != (sha2_byte*)0) {
        SHA512_Last((SHA512_CTX*)context);

        /* Save the hash data for output: */
#ifdef SHA2_BYTE_ORDER == SHA2_LITTLE_ENDIAN
        {
            /* Convert TO host byte order */
            int    j;
            for (j = 0; j < 6; j++) {
                REVERSE64(context->state[j], context->state[j]);
                *d++ = context->state[j];
            }
        }
#else
        bcopy(context->state, d, SHA384_DIGEST_LENGTH);
#endif
    }

    /* Zero out state data */
    bzero(context, sizeof(context));
}

char *SHA384_End(SHA384_CTX* context, char buffer[]) {
    sha2_byte    digest[SHA384_DIGEST_LENGTH], *d = digest;
    int          i;

    /* Sanity check: */
    assert(context != (SHA384_CTX*)0);

    if (buffer != (char*)0) {
        SHA384_Final(digest, context);

        for (i = 0; i < SHA384_DIGEST_LENGTH; i++) {
            *buffer++ = sha2_hex_digits[(d[i] & 0xf0) >> 4];
            *buffer++ = sha2_hex_digits[d[i] & 0x0f];
            d++;
        }
        *buffer = (char)0;
    } else {
        bzero(context, sizeof(context));
    }
    bzero(digest, SHA384_DIGEST_LENGTH);
    return buffer;
}

char* SHA384_Data(const sha2_byte* data, size_t len, char digest[SHA384_DIGEST_STRING_LENGTH]) {
    SHA384_CTX    context;

    SHA384_Init(&context);
    SHA384_Update(&context, data, len);
    return SHA384_End(&context, digest);
}

#endif /* !NO_64BIT */

/*
 * $PchId: sha2.c,v 1.1 2005/06/28 14:29:23 philip Exp $
 */

```

```
/*      $FreeBSD: src/sys/crypto/sha2/sha2.h,v 1.1.2.1 2001/07/03 11:01:36 ume Exp $      */
/*
/*      $KAME: sha2.h,v 1.3 2001/03/12 08:27:48 itojun Exp $      */

/*
 * sha2.h
 *
 * Version 1.0.0beta1
 *
 * Written by Aaron D. Gifford <me@aarongifford.com>
 *
 * Copyright 2000 Aaron D. Gifford. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the copyright holder nor the names of contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR(S) AND CONTRIBUTOR(S) ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR(S) OR CONTRIBUTOR(S) BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */

#ifndef __SHA2_H__
#define __SHA2_H__

#ifdef __cplusplus
extern "C" {
#endif

/** SHA-256/384/512 Various Length Definitions *****/
#define SHA256_BLOCK_LENGTH      64
#define SHA256_DIGEST_LENGTH    32
#define SHA256_DIGEST_STRING_LENGTH (SHA256_DIGEST_LENGTH * 2 + 1)
#define SHA384_BLOCK_LENGTH     128
#define SHA384_DIGEST_LENGTH    48
#define SHA384_DIGEST_STRING_LENGTH (SHA384_DIGEST_LENGTH * 2 + 1)
#define SHA512_BLOCK_LENGTH     128
#define SHA512_DIGEST_LENGTH    64
#define SHA512_DIGEST_STRING_LENGTH (SHA512_DIGEST_LENGTH * 2 + 1)

#ifdef __minix
#include <assert.h>
#include <string.h>
#include <sys/types.h>
#include <minix/u64.h>

typedef u8_t u_int8_t; /* 1-byte (8-bits) */
typedef u32_t u_int32_t; /* 4-bytes (32-bits) */
typedef u64_t u_int64_t; /* 8-bytes (64-bits) */

#ifndef __P
#define __P(x) x
#endif

#define NO_64BIT      1
#define MINIX_64BIT  1

```

```

#define SHA2_BYTE_ORDER          0x04030201
#define SHA2_LITTLE_ENDIAN      0x04030201
#define SHA2_BIG_ENDIAN         0x01020204
#define bcopy(s,d,l)            (memmove((d),(s),(l)))
#define bzero(d,l)              (memset((d), '0', (l)))
#endif

/** SHA-256/384/512 Context Structures *****/
/* NOTE: If your architecture does not define either u_intXX_t types or
 * uintXX_t (from inttypes.h), you may need to define things by hand
 * for your system:
 */
#if 0
typedef unsigned char u_int8_t;      /* 1-byte (8-bits) */
typedef unsigned int u_int32_t;      /* 4-bytes (32-bits) */
typedef unsigned long long u_int64_t; /* 8-bytes (64-bits) */
#endif
/*
 * Most BSD systems already define u_intXX_t types, as does Linux.
 * Some systems, however, like Compaq's Tru64 Unix instead can use
 * uintXX_t types defined by very recent ANSI C standards and included
 * in the file:
 *
 * #include <inttypes.h>
 *
 * If you choose to use <inttypes.h> then please define:
 *
 * #define SHA2_USE_INTTYPES_H
 *
 * Or on the command line during compile:
 *
 * cc -DSHA2_USE_INTTYPES_H ...
 */
#if 0 /*def SHA2_USE_INTTYPES_H*/

typedef struct _SHA256_CTX {
    uint32_t      state[8];
    uint64_t      bitcount;
    uint8_t buffer[SHA256_BLOCK_LENGTH];
} SHA256_CTX;
typedef struct _SHA512_CTX {
    uint64_t      state[8];
    uint64_t      bitcount[2];
    uint8_t buffer[SHA512_BLOCK_LENGTH];
} SHA512_CTX;

#else /* SHA2_USE_INTTYPES_H */

typedef struct _SHA256_CTX {
    u_int32_t      state[8];
    u_int64_t      bitcount;
    u_int8_t      buffer[SHA256_BLOCK_LENGTH];
} SHA256_CTX;
typedef struct _SHA512_CTX {
    u_int64_t      state[8];
    u_int64_t      bitcount[2];
    u_int8_t      buffer[SHA512_BLOCK_LENGTH];
} SHA512_CTX;

#endif /* SHA2_USE_INTTYPES_H */

typedef SHA512_CTX SHA384_CTX;

/** SHA-256/384/512 Function Prototypes *****/

void SHA256_Init __P((SHA256_CTX *));
void SHA256_Update __P((SHA256_CTX*, const u_int8_t*, size_t));
void SHA256_Final __P((u_int8_t[SHA256_DIGEST_LENGTH], SHA256_CTX*));
char* SHA256_End __P((SHA256_CTX*, char[SHA256_DIGEST_STRING_LENGTH]));
char* SHA256_Data __P((const u_int8_t*, size_t, char[SHA256_DIGEST_STRING_LENGTH]));

void SHA384_Init __P((SHA384_CTX*));

```

```
void SHA384_Update __P((SHA384_CTX*, const u_int8_t*, size_t));
void SHA384_Final __P((u_int8_t[SHA384_DIGEST_LENGTH], SHA384_CTX*));
char* SHA384_End __P((SHA384_CTX*, char[SHA384_DIGEST_STRING_LENGTH]));
char* SHA384_Data __P((const u_int8_t*, size_t, char[SHA384_DIGEST_STRING_LENGTH]));

void SHA512_Init __P((SHA512_CTX*));
void SHA512_Update __P((SHA512_CTX*, const u_int8_t*, size_t));
void SHA512_Final __P((u_int8_t[SHA512_DIGEST_LENGTH], SHA512_CTX*));
char* SHA512_End __P((SHA512_CTX*, char[SHA512_DIGEST_STRING_LENGTH]));
char* SHA512_Data __P((const u_int8_t*, size_t, char[SHA512_DIGEST_STRING_LENGTH]));

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* __SHA2_H__ */

/*
 * $PchId: sha2.h,v 1.1 2005/06/28 14:29:33 philip Exp $
 */
```

```

/*      this file contains the interface of the network software with the file
*      system.
*
* Copyright 1995 Philip Homburg
*
* The valid messages and their parameters are:
*
* Requests:
*
*      m_type      NDEV_MINOR      NDEV_PROC      NDEV_REF      NDEV_MODE
*      -----
*      | DEV_OPEN   |minor dev   | proc nr   | fd         | mode       |
*      |-----+-----+-----+-----+-----+
*      | DEV_CLOSE  |minor dev   | proc nr   | fd         |             |
*      |-----+-----+-----+-----+-----+
*
*      m_type      NDEV_MINOR      NDEV_PROC      NDEV_REF      NDEV_COUNT NDEV_BUFFER
*      -----
*      | DEV_READ   |minor dev   | proc nr   | fd         | count      | buf ptr    |
*      |-----+-----+-----+-----+-----+
*      | DEV_WRITE  |minor dev   | proc nr   | fd         | count      | buf ptr    |
*      |-----+-----+-----+-----+-----+
*
*      m_type      NDEV_MINOR      NDEV_PROC      NDEV_REF      NDEV_IOCTL NDEV_BUFFER
*      -----
*      | DEV_IOCTL3 |minor dev   | proc nr   | fd         | command    | buf ptr    |
*      |-----+-----+-----+-----+-----+
*
*      m_type      NDEV_MINOR      NDEV_PROC      NDEV_REF      NDEV_OPERATION
*      -----
*      | DEV_CANCEL |minor dev   | proc nr   | fd         | which operation|
*      |-----+-----+-----+-----+-----+
*
* Replies:
*
*      m_type      REP_PROC_NR      REP_STATUS      REP_REF      REP_OPERATION
*      -----
*      | DEVICE_REPLY |   proc nr   | status     | fd         | which operation|
*      |-----+-----+-----+-----+-----+
*/

#include "inet.h"

#ifndef __minix_vmd /* Minix 3 */
#include <sys/select.h>
#endif
#include <sys/svrctl.h>
#include <minix/callnr.h>

#include "mq.h"
#include "qp.h"
#include "proto.h"
#include "generic/type.h"

#include "generic/assert.h"
#include "generic/buf.h"
#include "generic/event.h"
#include "generic/sr.h"
#include "sr_int.h"

#ifndef __minix_vmd /* Minix 3 */
#define DEV_CANCEL NW_CANCEL
#define DEVICE_REPLY REVIVE
#define DEV_IOCTL3 DEV_IOCTL
#define NDEV_BUFFER ADDRESS
#define NDEV_COUNT COUNT
#define NDEV_IOCTL REQUEST
#define NDEV_MINOR DEVICE
#define NDEV_PROC IO_ENDPT
#endif

THIS_FILE

PUBLIC sr_fd_t sr_fd_table[FD_NR];

```

```

PRIVATE mq_t *repl_queue, *repl_queue_tail;
#ifdef __minix_vmd
PRIVATE cpvec_t cpvec[CPVEC_NR];
#else /* Minix 3 */
PRIVATE struct vir_cp_req vir_cp_req[CPVEC_NR];
#endif

FORWARD _PROTOTYPE ( int sr_open, (message *m) );
FORWARD _PROTOTYPE ( void sr_close, (message *m) );
FORWARD _PROTOTYPE ( int sr_rwio, (mq_t *m) );
FORWARD _PROTOTYPE ( int sr_restart_read, (sr_fd_t *fdp) );
FORWARD _PROTOTYPE ( int sr_restart_write, (sr_fd_t *fdp) );
FORWARD _PROTOTYPE ( int sr_restart_ioctl, (sr_fd_t *fdp) );
FORWARD _PROTOTYPE ( int sr_cancel, (message *m) );
#ifdef __minix_vmd /* Minix 3 */
FORWARD _PROTOTYPE ( int sr_select, (message *m) );
FORWARD _PROTOTYPE ( void sr_status, (message *m) );
#endif
FORWARD _PROTOTYPE ( void sr_reply_, (mq_t *m, int reply, int is_revive) );
FORWARD _PROTOTYPE ( sr_fd_t *sr_getchannel, (int minor));
FORWARD _PROTOTYPE ( acc_t *sr_get_userdata, (int fd, vir_bytes offset,
                                              vir_bytes count, int for_ioctl) );
FORWARD _PROTOTYPE ( int sr_put_userdata, (int fd, vir_bytes offset,
                                              acc_t *data, int for_ioctl) );

#ifdef __minix_vmd
#define sr_select_res 0
#else /* Minix 3 */
FORWARD _PROTOTYPE (void sr_select_res, (int fd, unsigned ops) );
#endif
FORWARD _PROTOTYPE ( int sr_repl_queue, (int proc, int ref, int operation) );
FORWARD _PROTOTYPE ( int walk_queue, (sr_fd_t *sr_fd, mq_t **q_head_ptr,
mq_t **q_tail_ptr, int type, int proc_nr, int ref, int first_flag) );
FORWARD _PROTOTYPE ( void process_req_q, (mq_t *mq, mq_t *tail,
mq_t **tail_ptr) );
FORWARD _PROTOTYPE ( void sr_event, (event_t *evp, ev_arg_t arg) );
FORWARD _PROTOTYPE ( int cp_u2b, (int proc, char *src, acc_t **var_acc_ptr,
int size) );
FORWARD _PROTOTYPE ( int cp_b2u, (acc_t *acc_ptr, int proc, char *dest) );

PUBLIC void sr_init()
{
    int i;

    for (i=0; i<FD_NR; i++)
    {
        sr_fd_table[i].srf_flags= SFF_FREE;
        ev_init(&sr_fd_table[i].srf_ioctl_ev);
        ev_init(&sr_fd_table[i].srf_read_ev);
        ev_init(&sr_fd_table[i].srf_write_ev);
    }
    repl_queue= NULL;
}

PUBLIC void sr_rec(m)
mq_t *m;
{
    int result;
    int send_reply, free_mess;

    if (repl_queue)
    {
        if (m->mq_mess.m_type == DEV_CANCEL)
        {
#ifdef __minix_vmd
            result= sr_repl_queue(m->mq_mess.NDEV_PROC,
m->mq_mess.NDEV_REF,
m->mq_mess.NDEV_OPERATION);
#else /* Minix 3 */
            result= sr_repl_queue(m->mq_mess.IO_ENDPT, 0, 0);
#endif
            if (result)
            {
                mq_free(m);
            }
        }
    }
}

```



```

        }
        }
        return; /* canceled request in queue */
    }
}

#if 0
    else
        sr_repl_queue(ANY, 0, 0);
#endif
}

switch (m->mq_mess.m_type)
{
case DEV_OPEN:
    result= sr_open(&m->mq_mess);
    send_reply= 1;
    free_mess= 1;
    break;
case DEV_CLOSE:
    sr_close(&m->mq_mess);
    result= OK;
    send_reply= 1;
    free_mess= 1;
    break;
case DEV_READ:
case DEV_WRITE:
case DEV_IOCTL3:
    result= sr_rwio(m);
    assert(result == OK || result == SUSPEND);
    send_reply= (result == SUSPEND);
    free_mess= 0;
    break;
case DEV_CANCEL:
    result= sr_cancel(&m->mq_mess);
    assert(result == OK || result == EINTR);
    send_reply= (result == EINTR);
    free_mess= 1;
#ifdef __minix_vmd
    m->mq_mess.m_type= m->mq_mess.NDEV_OPERATION;
#else /* Minix 3 */
    m->mq_mess.m_type= 0;
#endif
    break;
#ifndef __minix_vmd /* Minix 3 */
case DEV_SELECT:
    result= sr_select(&m->mq_mess);
    send_reply= 1;
    free_mess= 1;
    break;
case DEV_STATUS:
    sr_status(&m->mq_mess);
    send_reply= 0;
    free_mess= 1;
    break;
#endif
default:
    ip_panic(("unknown message, from %d, type %d",
            m->mq_mess.m_source, m->mq_mess.m_type));
}
if (send_reply)
{
    sr_reply(m, result, FALSE /* !is_revive */);
}
if (free_mess)
    mq_free(m);
}

PUBLIC void sr_add_minor(minor, port, openf, closef, readf, writef,
    ioctlf, cancelf, selectf)
int minor;
int port;
sr_open_t openf;
sr_close_t closef;
sr_read_t readf;
sr_write_t writef;
sr_ioctl_t ioctlf;

```

```
sr_cancel_t cancelf;
sr_select_t selectf;
{
    sr_fd_t *sr_fd;

    assert (minor>=0 && minor<FD_NR);

    sr_fd= &sr_fd_table[minor];

    assert(!(sr_fd->srf_flags & SFF_INUSE));

    sr_fd->srf_flags= SFF_INUSE | SFF_MINOR;
    sr_fd->srf_port= port;
    sr_fd->srf_open= openf;
    sr_fd->srf_close= closef;
    sr_fd->srf_write= writef;
    sr_fd->srf_read= readf;
    sr_fd->srf_ioctl= ioctlf;
    sr_fd->srf_cancel= cancelf;
    sr_fd->srf_select= selectf;
}

PRIVATE int sr_open(m)
message *m;
{
    sr_fd_t *sr_fd;

    int minor= m->NDEV_MINOR;
    int i, fd;

    if (minor<0 || minor>FD_NR)
    {
        DBLOCK(1, printf("replying EINVAL\n"));
        return EINVAL;
    }
    if (!(sr_fd_table[minor].srf_flags & SFF_MINOR))
    {
        DBLOCK(1, printf("replying ENXIO\n"));
        return ENXIO;
    }
    for (i=0; i<FD_NR && (sr_fd_table[i].srf_flags & SFF_INUSE); i++);

    if (i>=FD_NR)
    {
        DBLOCK(1, printf("replying ENFILE\n"));
        return ENFILE;
    }

    sr_fd= &sr_fd_table[i];
    *sr_fd= sr_fd_table[minor];
    sr_fd->srf_flags= SFF_INUSE;
    fd= (*sr_fd->srf_open)(sr_fd->srf_port, i, sr_get_userdata,
        sr_put_userdata, 0 /* no put_pkt */, sr_select_res);
    if (fd<0)
    {
        sr_fd->srf_flags= SFF_FREE;
        DBLOCK(1, printf("replying %d\n", fd));
        return fd;
    }
    sr_fd->srf_fd= fd;
    return i;
}

PRIVATE void sr_close(m)
message *m;
{
    sr_fd_t *sr_fd;

    sr_fd= sr_getchannel(m->NDEV_MINOR);
    assert (sr_fd);

    if (sr_fd->srf_flags & SFF_BUSY)
        ip_panic(("close on busy channel"));
}
```

```

    assert (!(sr_fd->srf_flags & SFF_MINOR));
    (*sr_fd->srf_close)(sr_fd->srf_fd);
    sr_fd->srf_flags= SFF_FREE;
}

PRIVATE int sr_rwio(m)
mq_t *m;
{
    sr_fd_t *sr_fd;
    mq_t **q_head_ptr, **q_tail_ptr;
    int ip_flag, susp_flag, first_flag;
    int r;
    ioreq_t request;
    size_t size;

    sr_fd= sr_getchannel(m->mq_mess.NDEV_MINOR);
    assert (sr_fd);

    switch(m->mq_mess.m_type)
    {
    case DEV_READ:
        q_head_ptr= &sr_fd->srf_read_q;
        q_tail_ptr= &sr_fd->srf_read_q_tail;
        ip_flag= SFF_READ_IP;
        susp_flag= SFF_READ_SUSP;
        first_flag= SFF_READ_FIRST;
        break;
    case DEV_WRITE:
        q_head_ptr= &sr_fd->srf_write_q;
        q_tail_ptr= &sr_fd->srf_write_q_tail;
        ip_flag= SFF_WRITE_IP;
        susp_flag= SFF_WRITE_SUSP;
        first_flag= SFF_WRITE_FIRST;
        break;
    case DEV_IOCTL3:
        q_head_ptr= &sr_fd->srf_ioctl_q;
        q_tail_ptr= &sr_fd->srf_ioctl_q_tail;
        ip_flag= SFF_IOCTL_IP;
        susp_flag= SFF_IOCTL_SUSP;
        first_flag= SFF_IOCTL_FIRST;
        break;
    default:
        ip_panic(("illegal case entry"));
    }

    if (sr_fd->srf_flags & ip_flag)
    {
        assert(sr_fd->srf_flags & susp_flag);
        assert(*q_head_ptr);

        (*q_tail_ptr)->mq_next= m;
        *q_tail_ptr= m;
        return SUSPEND;
    }
    assert(!*q_head_ptr);

    *q_tail_ptr= *q_head_ptr= m;
    sr_fd->srf_flags |= ip_flag;
    assert(!(sr_fd->srf_flags & first_flag));
    sr_fd->srf_flags |= first_flag;

    switch(m->mq_mess.m_type)
    {
    case DEV_READ:
        r= (*sr_fd->srf_read)(sr_fd->srf_fd,
                             m->mq_mess.NDEV_COUNT);
        break;
    case DEV_WRITE:
        r= (*sr_fd->srf_write)(sr_fd->srf_fd,
                              m->mq_mess.NDEV_COUNT);
        break;
    case DEV_IOCTL3:
        request= m->mq_mess.NDEV_IOCTL;

```

```

    /* There should be a better way to do this... */
    if (request == NWIOQUERYPARAM)
    {
        r= qp_query(m->mq_mess.NDEV_PROC,
                    (vir_bytes)m->mq_mess.NDEV_BUFFER);
        r= sr_put_userdata(sr_fd->sr_fd_table, r, NULL, 1);
        assert(r == OK);
        break;
    }

    /* And now, we continue with our regular program. */
    size= (request >> 16) & _IOCPARM_MASK;
    if (size>MAX_IOCTL_S)
    {
        DBLOCK(1, printf("replying EINVAL\n"));
        r= sr_put_userdata(sr_fd->sr_fd_table, EINVAL,
                          NULL, 1);
        assert(r == OK);
        assert(sr_fd->srf_flags & first_flag);
        sr_fd->srf_flags &= ~first_flag;
        return OK;
    }
    r= (*sr_fd->srf_ioctl)(sr_fd->srf_fd, request);
    break;
default:
    ip_panic(("illegal case entry"));
}

assert(sr_fd->srf_flags & first_flag);
sr_fd->srf_flags &= ~first_flag;

assert(r == OK || r == SUSPEND ||
       (printf("r=%d\n", r), 0));
if (r == SUSPEND)
    sr_fd->srf_flags |= susp_flag;
else
    mq_free(m);
return r;
}

PRIVATE int sr_restart_read(sr_fd)
sr_fd_t *sr_fd;
{
    mq_t *mp;
    int r;

    mp= sr_fd->srf_read_q;
    assert(mp);

    if (sr_fd->srf_flags & SFF_READ_IP)
    {
        assert(sr_fd->srf_flags & SFF_READ_SUSP);
        return SUSPEND;
    }
    sr_fd->srf_flags |= SFF_READ_IP;

    r= (*sr_fd->srf_read)(sr_fd->srf_fd,
                        mp->mq_mess.NDEV_COUNT);

    assert(r == OK || r == SUSPEND ||
           (printf("r=%d\n", r), 0));
    if (r == SUSPEND)
        sr_fd->srf_flags |= SFF_READ_SUSP;
    return r;
}

PRIVATE int sr_restart_write(sr_fd)
sr_fd_t *sr_fd;
{
    mq_t *mp;
    int r;

    mp= sr_fd->srf_write_q;
    assert(mp);

```

```

    if (sr_fd->srf_flags & SFF_WRITE_IP)
    {
        assert(sr_fd->srf_flags & SFF_WRITE_SUSP);
        return SUSPEND;
    }
    sr_fd->srf_flags |= SFF_WRITE_IP;

    r= (*sr_fd->srf_write)(sr_fd->srf_fd,
        mp->mq_mess.NDEV_COUNT);

    assert(r == OK || r == SUSPEND ||
        (printf("r= %d\n", r), 0));
    if (r == SUSPEND)
        sr_fd->srf_flags |= SFF_WRITE_SUSP;
    return r;
}

PRIVATE int sr_restart_ioctl(sr_fd)
sr_fd_t *sr_fd;
{
    mq_t *mp;
    int r;

    mp= sr_fd->srf_ioctl_q;
    assert(mp);

    if (sr_fd->srf_flags & SFF_IOCTL_IP)
    {
        assert(sr_fd->srf_flags & SFF_IOCTL_SUSP);
        return SUSPEND;
    }
    sr_fd->srf_flags |= SFF_IOCTL_IP;

    r= (*sr_fd->srf_ioctl)(sr_fd->srf_fd,
        mp->mq_mess.NDEV_COUNT);

    assert(r == OK || r == SUSPEND ||
        (printf("r= %d\n", r), 0));
    if (r == SUSPEND)
        sr_fd->srf_flags |= SFF_IOCTL_SUSP;
    return r;
}

PRIVATE int sr_cancel(m)
message *m;
{
    sr_fd_t *sr_fd;
    int result;
    int proc_nr, ref, operation;

    result=EINTR;
    proc_nr= m->NDEV_PROC;
#ifdef __minix_vmd
    ref= m->NDEV_REF;
    operation= m->NDEV_OPERATION;
#else /* Minix 3 */
    ref= 0;
    operation= 0;
#endif
    sr_fd= sr_getchannel(m->NDEV_MINOR);
    assert (sr_fd);

#ifdef __minix_vmd
    if (operation == CANCEL_ANY || operation == DEV_IOCTL3)
#endif
    {
        result= walk_queue(sr_fd, &sr_fd->srf_ioctl_q,
            &sr_fd->srf_ioctl_q_tail, SR_CANCEL_IOCTL,
            proc_nr, ref, SFF_IOCTL_FIRST);
        if (result != EAGAIN)
            return result;
    }
#ifdef __minix_vmd
}
#endif

```

```

    if (operation == CANCEL_ANY || operation == DEV_READ)
#endif
    {
        result= walk_queue(sr_fd, &sr_fd->srf_read_q,
                           &sr_fd->srf_read_q_tail, SR_CANCEL_READ,
                           proc_nr, ref, SFF_READ_FIRST);
        if (result != EAGAIN)
            return result;
    }
#ifdef __minix_vmd
    if (operation == CANCEL_ANY || operation == DEV_WRITE)
#endif
    {
        result= walk_queue(sr_fd, &sr_fd->srf_write_q,
                           &sr_fd->srf_write_q_tail, SR_CANCEL_WRITE,
                           proc_nr, ref, SFF_WRITE_FIRST);
        if (result != EAGAIN)
            return result;
    }
#ifdef __minix_vmd
    ip_panic(("request not found: from %d, type %d, MINOR= %d, PROC= %d, REF= %d OPERATION= %ld",
              m->m_source, m->m_type, m->NDEV_MINOR,
              m->NDEV_PROC, m->NDEV_REF, m->NDEV_OPERATION));
#else /* Minix 3 */
    ip_panic(("request not found: from %d, type %d, MINOR= %d, PROC= %d",
              m->m_source, m->m_type, m->NDEV_MINOR,
              m->NDEV_PROC));
#endif
}

#ifdef __minix_vmd /* Minix 3 */
PRIVATE int sr_select(m)
message *m;
{
    sr_fd_t *sr_fd;
    mq_t **q_head_ptr, **q_tail_ptr;
    int ip_flag, susp_flag;
    int r, ops;
    unsigned m_ops, i_ops;
    ioreq_t request;
    size_t size;

    sr_fd= sr_getchannel(m->NDEV_MINOR);
    assert (sr_fd);

    sr_fd->srf_select_proc= m->m_source;

    m_ops= m->IO_ENDPT;
    i_ops= 0;
    if (m_ops & SEL_RD) i_ops |= SR_SELECT_READ;
    if (m_ops & SEL_WR) i_ops |= SR_SELECT_WRITE;
    if (m_ops & SEL_ERR) i_ops |= SR_SELECT_EXCEPTION;
    if (!(m_ops & SEL_NOTIFY)) i_ops |= SR_SELECT_POLL;

    r= (*sr_fd->srf_select)(sr_fd->srf_fd, i_ops);
    if (r < 0)
        return r;
    m_ops= 0;
    if (r & SR_SELECT_READ) m_ops |= SEL_RD;
    if (r & SR_SELECT_WRITE) m_ops |= SEL_WR;
    if (r & SR_SELECT_EXCEPTION) m_ops |= SEL_ERR;

    return m_ops;
}

PRIVATE void sr_status(m)
message *m;
{
    int fd, result;
    unsigned m_ops;
    sr_fd_t *sr_fd;
    mq_t *mq;

```

```

mq= repl_queue;
if (mq != NULL)
{
    repl_queue= mq->mq_next;

    mq->mq_mess.m_type= DEV_REVIVE;
    result= send(mq->mq_mess.m_source, &mq->mq_mess);
    if (result != OK)
        ip_panic(( "unable to send" ));
    mq_free(mq);

    return;
}

for (fd=0, sr_fd= sr_fd_table; fd<FD_NR; fd++, sr_fd++)
{
    if ((sr_fd->srf_flags &
        (SFF_SELECT_R|SFF_SELECT_W|SFF_SELECT_X)) == 0)
    {
        /* Nothing to report */
        continue;
    }
    if (sr_fd->srf_select_proc != m->m_source)
    {
        /* Wrong process */
        continue;
    }

    m_ops= 0;
    if (sr_fd->srf_flags & SFF_SELECT_R) m_ops |= SEL_RD;
    if (sr_fd->srf_flags & SFF_SELECT_W) m_ops |= SEL_WR;
    if (sr_fd->srf_flags & SFF_SELECT_X) m_ops |= SEL_ERR;

    sr_fd->srf_flags &= ~(SFF_SELECT_R|SFF_SELECT_W|SFF_SELECT_X);

    m->m_type= DEV_IO_READY;
    m->DEV_MINOR= fd;
    m->DEV_SEL_OPS= m_ops;

    result= send(m->m_source, m);
    if (result != OK)
        ip_panic(( "unable to send" ));
    return;
}

m->m_type= DEV_NO_STATUS;
result= send(m->m_source, m);
if (result != OK)
    ip_panic(( "unable to send" ));
}
#endif

PRIVATE int walk_queue(sr_fd, q_head_ptr, q_tail_ptr, type, proc_nr, ref,
    first_flag)
sr_fd_t *sr_fd;
mq_t **q_head_ptr;
mq_t **q_tail_ptr;
int type;
int proc_nr;
int ref;
int first_flag;
{
    mq_t *q_ptr_prv, *q_ptr;
    int result;

    for(q_ptr_prv= NULL, q_ptr= *q_head_ptr; q_ptr;
        q_ptr_prv= q_ptr, q_ptr= q_ptr->mq_next)
    {
        if (q_ptr->mq_mess.NDEV_PROC != proc_nr)
            continue;
#ifdef __minix_vmd
        if (q_ptr->mq_mess.NDEV_REF != ref)
            continue;

```

```

#endif
        if (!q_ptr_prv)
        {
            assert(!(sr_fd->srf_flags & first_flag));
            sr_fd->srf_flags |= first_flag;

            result= (*sr_fd->srf_cancel)(sr_fd->srf_fd, type);
            assert(result == OK);

            *q_head_ptr= q_ptr->mq_next;
            mq_free(q_ptr);

            assert(sr_fd->srf_flags & first_flag);
            sr_fd->srf_flags &= ~first_flag;

            return OK;
        }
        q_ptr_prv->mq_next= q_ptr->mq_next;
        mq_free(q_ptr);
        if (!q_ptr_prv->mq_next)
            *q_tail_ptr= q_ptr_prv;
        return EINTR;
    }
    return EAGAIN;
}

PRIVATE sr_fd_t *sr_getchannel(minor)
int minor;
{
    sr_fd_t *loc_fd;

    compare(minor, >=, 0);
    compare(minor, <, FD_NR);

    loc_fd= &sr_fd_table[minor];

    assert (!(loc_fd->srf_flags & SFF_MINOR) &&
            (loc_fd->srf_flags & SFF_INUSE));

    return loc_fd;
}

PRIVATE void sr_reply_(mq, status, is_revive)
mq_t *mq;
int status;
int is_revive;
{
    int result, proc, ref, operation;
    message reply, *mp;

    proc= mq->mq_mess.NDEV_PROC;
#ifdef __minix_vmd
    ref= mq->mq_mess.NDEV_REF;
#else /* Minix 3 */
    ref= 0;
#endif
    operation= mq->mq_mess.m_type;
#ifdef __minix_vmd
    assert(operation != DEV_CANCEL);
#endif

    if (is_revive)
        mp= &mq->mq_mess;
    else
        mp= &reply;

    mp->m_type= DEVICE_REPLY;
    mp->REP_ENDPT= proc;
    mp->REP_STATUS= status;
#ifdef __minix_vmd
    mp->REP_REF= ref;
    mp->REP_OPERATION= operation;
#endif
    if (is_revive)

```



```

    {
        notify(mq->mq_mess.m_source);
        result= ELOCKED;
    }
    else
        result= send(mq->mq_mess.m_source, mp);

    if (result == ELOCKED && is_revive)
    {
        mq->mq_next= NULL;
        if (repl_queue)
            repl_queue_tail->mq_next= mq;
        else
            repl_queue= mq;
        repl_queue_tail= mq;
        return;
    }
    if (result != OK)
        ip_panic(("unable to send"));
    if (is_revive)
        mq_free(mq);
}

PRIVATE acc_t *sr_get_userdata (fd, offset, count, for_ioctl)
int fd;
vir_bytes offset;
vir_bytes count;
int for_ioctl;
{
    sr_fd_t *loc_fd;
    mq_t **head_ptr, *m, *mq;
    int ip_flag, susp_flag, first_flag;
    int result, suspended, is_revive;
    char *src;
    acc_t *acc;
    event_t *evp;
    ev_arg_t arg;

    loc_fd= &sr_fd_table[fd];

    if (for_ioctl)
    {
        head_ptr= &loc_fd->srf_ioctl_q;
        evp= &loc_fd->srf_ioctl_ev;
        ip_flag= SFF_IOCTL_IP;
        susp_flag= SFF_IOCTL_SUSP;
        first_flag= SFF_IOCTL_FIRST;
    }
    else
    {
        head_ptr= &loc_fd->srf_write_q;
        evp= &loc_fd->srf_write_ev;
        ip_flag= SFF_WRITE_IP;
        susp_flag= SFF_WRITE_SUSP;
        first_flag= SFF_WRITE_FIRST;
    }

    assert (loc_fd->srf_flags & ip_flag);

    if (!count)
    {
        m= *head_ptr;
        mq= m->mq_next;
        *head_ptr= mq;
        result= (int)offset;
        is_revive= !(loc_fd->srf_flags & first_flag);
        sr_reply(m, result, is_revive);
        suspended= (loc_fd->srf_flags & susp_flag);
        loc_fd->srf_flags &= ~(ip_flag|susp_flag);
        if (suspended)
        {
            if (mq)
            {
                arg.ev_ptr= loc_fd;

```

```

        ev_enqueue(evp, sr_event, arg);
    }
    }
    return NULL;
}

src= (*head_ptr)->mq_mess.NDEV_BUFFER + offset;
result= cp_u2b ((*head_ptr)->mq_mess.NDEV_PROC, src, &acc, count);

return result<0 ? NULL : acc;
}

```

```

PRIVATE int sr_put_userdata (fd, offset, data, for_ioctl)
int fd;
vir_bytes offset;
acc_t *data;
int for_ioctl;
{
    sr_fd_t *loc_fd;
    mq_t **head_ptr, *m, *mq;
    int ip_flag, susp_flag, first_flag;
    int result, suspended, is_revive;
    char *dst;
    event_t *evp;
    ev_arg_t arg;

    loc_fd= &sr_fd_table[fd];

    if (for_ioctl)
    {
        head_ptr= &loc_fd->srf_ioctl_q;
        evp= &loc_fd->srf_ioctl_ev;
        ip_flag= SFF_IOCTL_IP;
        susp_flag= SFF_IOCTL_SUSP;
        first_flag= SFF_IOCTL_FIRST;
    }
    else
    {
        head_ptr= &loc_fd->srf_read_q;
        evp= &loc_fd->srf_read_ev;
        ip_flag= SFF_READ_IP;
        susp_flag= SFF_READ_SUSP;
        first_flag= SFF_READ_FIRST;
    }

    assert (loc_fd->srf_flags & ip_flag);

    if (!data)
    {
        m= *head_ptr;
        mq= m->mq_next;
        *head_ptr= mq;
        result= (int)offset;
        is_revive= !(loc_fd->srf_flags & first_flag);
        sr_reply(m, result, is_revive);
        suspended= (loc_fd->srf_flags & susp_flag);
        loc_fd->srf_flags &= ~(ip_flag|susp_flag);
        if (suspended)
        {
            if (mq)
            {
                arg.ev_ptr= loc_fd;
                ev_enqueue(evp, sr_event, arg);
            }
        }
        return OK;
    }

    dst= (*head_ptr)->mq_mess.NDEV_BUFFER + offset;
    return cp_b2u (data, (*head_ptr)->mq_mess.NDEV_PROC, dst);
}

```

```

#ifdef __minix_vmd /* Minix 3 */
PRIVATE void sr_select_res(fd, ops)

```

```

int fd;
unsigned ops;
{
    sr_fd_t *sr_fd;

    sr_fd= &sr_fd_table[fd];

    if (ops & SR_SELECT_READ) sr_fd->srf_flags |= SFF_SELECT_R;
    if (ops & SR_SELECT_WRITE) sr_fd->srf_flags |= SFF_SELECT_W;
    if (ops & SR_SELECT_EXCEPTION) sr_fd->srf_flags |= SFF_SELECT_X;

    notify(sr_fd->srf_select_proc);
}
#endif

PRIVATE void process_req_q(mq, tail, tail_ptr)
mq_t *mq, *tail, **tail_ptr;
{
    mq_t *m;
    int result;

    for(;mq;)
    {
        m= mq;
        mq= mq->mq_next;

        result= sr_rwio(m);
        if (result == SUSPEND)
        {
            if (mq)
            {
                (*tail_ptr)->mq_next= mq;
                *tail_ptr= tail;
            }
            return;
        }
    }
    return;
}

PRIVATE void sr_event(ev, arg)
event_t *ev;
ev_arg_t arg;
{
    sr_fd_t *sr_fd;
    int r;

    sr_fd= arg.ev_ptr;
    if (ev == &sr_fd->srf_write_ev)
    {
        while(sr_fd->srf_write_q)
        {
            r= sr_restart_write(sr_fd);
            if (r == SUSPEND)
                return;
        }
        return;
    }
    if (ev == &sr_fd->srf_read_ev)
    {
        while(sr_fd->srf_read_q)
        {
            r= sr_restart_read(sr_fd);
            if (r == SUSPEND)
                return;
        }
        return;
    }
    if (ev == &sr_fd->srf_ioctl_ev)
    {
        while(sr_fd->srf_ioctl_q)
        {
            r= sr_restart_ioctl(sr_fd);
            if (r == SUSPEND)

```

```

        }
        return;
    }
    ip_panic(("sr_event: unkown event\n"));
}

PRIVATE int cp_u2b (proc, src, var_acc_ptr, size)
int proc;
char *src;
acc_t **var_acc_ptr;
int size;
{
    static message mess;
    acc_t *acc;
    int i;

    acc= bf_memreq(size);

    *var_acc_ptr= acc;
    i=0;

    while (acc)
    {
        size= (vir_bytes)acc->acc_length;
#ifdef __minix_vmd
        cpvec[i].cpv_src= (vir_bytes)src;
        cpvec[i].cpv_dst= (vir_bytes)ptr2acc_data(acc);
        cpvec[i].cpv_size= size;
#else /* Minix 3 */
        vir_cp_req[i].count= size;
        vir_cp_req[i].src.proc_nr_e = proc;
        vir_cp_req[i].src.segment = D;
        vir_cp_req[i].src.offset = (vir_bytes) src;
        vir_cp_req[i].dst.proc_nr_e = this_proc;
        vir_cp_req[i].dst.segment = D;
        vir_cp_req[i].dst.offset = (vir_bytes) ptr2acc_data(acc);
#endif

        src += size;
        acc= acc->acc_next;
        i++;

        if (i == CPVEC_NR || acc == NULL)
        {
#ifdef __minix_vmd
            mess.m_type= SYS_VCOPY;
            mess.ml_i1= proc;
            mess.ml_i2= this_proc;
            mess.ml_i3= i;
            mess.ml_p1= (char *)cpvec;
#else /* Minix 3 */
            mess.m_type= SYS_VIRVCOPY;
            mess.VCP_VEC_SIZE= i;
            mess.VCP_VEC_ADDR= (char *)vir_cp_req;
#endif

            if (sendrec(SYSTASK, &mess) <0)
                ip_panic(("unable to sendrec"));
            if (mess.m_type <0)
            {
                bf_afree(*var_acc_ptr);
                *var_acc_ptr= 0;
                return mess.m_type;
            }
            i= 0;
        }
    }
    return OK;
}

PRIVATE int cp_b2u (acc_ptr, proc, dest)
acc_t *acc_ptr;
int proc;

```

```

char *dest;
{
    static message mess;
    acc_t *acc;
    int i, size;

    acc= acc_ptr;
    i=0;

    while (acc)
    {
        size= (vir_bytes)acc->acc_length;

        if (size)
        {
#ifdef __minix_vmd
            cpvec[i].cpv_src= (vir_bytes)ptr2acc_data(acc);
            cpvec[i].cpv_dst= (vir_bytes)dest;
            cpvec[i].cpv_size= size;
#else /* Minix 3 */
            vir_cp_req[i].src.proc_nr_e = this_proc;
            vir_cp_req[i].src.segment = D;
            vir_cp_req[i].src.offset= (vir_bytes)ptr2acc_data(acc);
            vir_cp_req[i].dst.proc_nr_e = proc;
            vir_cp_req[i].dst.segment = D;
            vir_cp_req[i].dst.offset= (vir_bytes)dest;
            vir_cp_req[i].count= size;
#endif

            i++;

            dest += size;
            acc= acc->acc_next;

            if (i == CPVEC_NR || acc == NULL)
            {
#ifdef __minix_vmd
                mess.m_type= SYS_VCOPY;
                mess.ml_i1= this_proc;
                mess.ml_i2= proc;
                mess.ml_i3= i;
                mess.ml_p1= (char *)cpvec;
#else /* Minix 3 */
                mess.m_type= SYS_VIRVCOPY;
                mess.VCP_VEC_SIZE= i;
                mess.VCP_VEC_ADDR= (char *) vir_cp_req;
#endif

                if (sendrec(SYSTASK, &mess) <0)
                    ip_panic(("unable to sendrec"));
                if (mess.m_type <0)
                {
                    bf_afree(acc_ptr);
                    return mess.m_type;
                }
                i= 0;
            }
            bf_afree(acc_ptr);
            return OK;
        }
    }

PRIVATE int sr_repl_queue(proc, ref, operation)
int proc;
int ref;
int operation;
{
    mq_t *m, *m_cancel, *m_tmp;
    int result;

    m_cancel= NULL;

    for (m= repl_queue; m;)
    {
#ifdef __minix_vmd

```

```
        if (m->mq_mess.REP_ENDPT == proc &&
            m->mq_mess.REP_REF ==ref &&
            (m->mq_mess.REP_OPERATION == operation ||
             operation == CANCEL_ANY))
#else /* Minix 3 */
        if (m->mq_mess.REP_ENDPT == proc)
#endif
    {
assert(!m_cancel);

        m_cancel= m;
        m= m->mq_next;
        continue;
    }
    result= send(m->mq_mess.m_source, &m->mq_mess);
    if (result != OK)
        ip_panic(("unable to send: %d", result));
    m_tmp= m;
    m= m->mq_next;
    mq_free(m_tmp);
}
repl_queue= NULL;
if (m_cancel)
{
    result= send(m_cancel->mq_mess.m_source, &m_cancel->mq_mess);
    if (result != OK)
        ip_panic(("unable to send: %d", result));
    mq_free(m_cancel);
    return 1;
}
return 0;
}

/*
 * $PchId: sr.c,v 1.17 2005/06/28 14:26:16 philip Exp $
 */
```

```

/*
inet/sr_int.h

SR internals

Created:      Aug 2004 by Philip Homburg <philip@f-mnx.phicoh.com>
*/

#define FD_NR                (16*IP_PORT_MAX)

typedef struct sr_fd
{
    int srf_flags;
    int srf_fd;
    int srf_port;
    int srf_select_proc;
    sr_open_t srf_open;
    sr_close_t srf_close;
    sr_write_t srf_write;
    sr_read_t srf_read;
    sr_ioctl_t srf_ioctl;
    sr_cancel_t srf_cancel;
    sr_select_t srf_select;
    mq_t *srf_ioctl_q, *srf_ioctl_q_tail;
    mq_t *srf_read_q, *srf_read_q_tail;
    mq_t *srf_write_q, *srf_write_q_tail;
    event_t srf_ioctl_ev;
    event_t srf_read_ev;
    event_t srf_write_ev;
} sr_fd_t;

#    define SFF_FREE          0x00
#    define SFF_MINOR        0x01
#    define SFF_INUSE        0x02
#define SFF_BUSY             0x1C
#    define SFF_IOCTL_IP     0x04
#    define SFF_READ_IP      0x08
#    define SFF_WRITE_IP     0x10
#define SFF_SUSPENDED        0x1C0
#    define SFF_IOCTL_SUSP   0x40
#    define SFF_READ_SUSP    0x80
#    define SFF_WRITE_SUSP   0x100
#define SFF_IOCTL_FIRST      0x200
#define SFF_READ_FIRST       0x400
#define SFF_WRITE_FIRST      0x800
#define SFF_SELECT_R         0x1000
#define SFF_SELECT_W         0x2000
#define SFF_SELECT_X         0x4000

EXTERN sr_fd_t sr_fd_table[FD_NR];

/*
 * $PchId: sr_int.h,v 1.2 2005/06/28 14:28:17 philip Exp $
 */

```

```
/*
stacktrace.c

Created:      Jan 19, 1993 by Philip Homburg

Copyright 1995 Philip Homburg
*/

#include "inet.h"

PUBLIC void stacktrace()
{
    typedef unsigned int reg_t;
    reg_t bp, pc, hbp;
    extern reg_t get_bp ARGS(( void ));

    bp= get_bp();
    while(bp)
    {
        pc= ((reg_t *)bp)[1];
        hbp= ((reg_t *)bp)[0];
        printf("0x%x ", (unsigned long)pc);
        if (hbp != 0 && hbp <= bp)
        {
            printf("??");
            break;
        }
        bp= hbp;
    }
    printf("\n");
}

/*
 * $PchId: stacktrace.c,v 1.6 1996/05/07 21:11:34 philip Exp $
 */
```



```
/*  
version.c  
*/  
  
#include "inet.h"  
  
char version[] = "inet 0.79, last compiled on " __DATE__ " " __TIME__;  
  
/*  
 * $PchId: version.c,v 1.54 2005/06/28 14:35:01 philip Exp $  
 */
```

```
/*
arp.c

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "type.h"

#include "arp.h"
#include "assert.h"
#include "buf.h"
#include "clock.h"
#include "event.h"
#include "eth.h"
#include "io.h"
#include "sr.h"

THIS_FILE

#define ARP_CACHE_NR      256
#define AP_REQ_NR         32

#define ARP_HASH_NR       256
#define ARP_HASH_MASK     0xff
#define ARP_HASH_WIDTH    4

#define MAX_ARP_RETRIES    5
#define ARP_TIMEOUT        (HZ/2+1)      /* .5 seconds */
#ifndef ARP_EXP_TIME
#define ARP_EXP_TIME        (20L*60L*HZ)  /* 20 minutes */
#endif
#define ARP_NOTRCH_EXP_TIME (30*HZ)       /* 30 seconds */
#define ARP_INUSE_OFFSET   (60*HZ) /* an entry in the cache can be deleted
                                     if its not used for 1 minute */

typedef struct arp46
{
    ether_addr_t a46_dstaddr;
    ether_addr_t a46_srcaddr;
    ether_type_t a46_ethtype;
    union
    {
        struct
        {
            u16_t a_hdr, a_pro;
            u8_t a_hln, a_pln;
            u16_t a_op;
            ether_addr_t a_sha;
            u8_t a_spa[4];
            ether_addr_t a_tha;
            u8_t a_tpa[4];
        } a46_data;
        char a46_dummy[ETH_MIN_PACK_SIZE-ETH_HDR_SIZE];
    } a46_data;
} arp46_t;

#define a46_hdr a46_data.a46_data.a_hdr
#define a46_pro a46_data.a46_data.a_pro
#define a46_hln a46_data.a46_data.a_hln
#define a46_pln a46_data.a46_data.a_pln
#define a46_op a46_data.a46_data.a_op
#define a46_sha a46_data.a46_data.a_sha
#define a46_spa a46_data.a46_data.a_spa
#define a46_tha a46_data.a46_data.a_tha
#define a46_tpa a46_data.a46_data.a_tpa

typedef struct arp_port
{
    int ap_flags;
    int ap_state;
    int ap_eth_port;
    int ap_ip_port;
    int ap_eth_fd;
```

```

ether_addr_t ap_ethaddr;      /* Ethernet address of this port */
ipaddr_t ap_ipaddr;          /* IP address of this port */

struct arp_req
{
    timer_t ar_timer;
    int ar_entry;
    int ar_req_count;
} ap_req[AP_REQ_NR];

arp_func_t ap_arp_func;

acc_t *ap_sendpkt;
acc_t *ap_sendlist;
acc_t *ap_reclist;
event_t ap_event;
} arp_port_t;

#define APF_EMPTY      0x00
#define APF_ARP_RD_IP  0x01
#define APF_ARP_RD_SP  0x02
#define APF_ARP_WR_IP  0x04
#define APF_ARP_WR_SP  0x08
#define APF_INADDR_SET 0x10
#define APF_SUSPEND    0x20

#define APS_INITIAL     1
#define APS_GETADDR     2
#define APS_ARPSTART    3
#define APS_ARPPROTO    4
#define APS_ARPMAIN     5
#define APS_ERROR       6

typedef struct arp_cache
{
    int ac_flags;
    int ac_state;
    ether_addr_t ac_ethaddr;
    ipaddr_t ac_ipaddr;
    arp_port_t *ac_port;
    time_t ac_expire;
    time_t ac_lastuse;
} arp_cache_t;

#define ACF_EMPTY      0
#define ACF_PERM       1
#define ACF_PUB        2

#define ACS_UNUSED     0
#define ACS_INCOMPLETE 1
#define ACS_VALID      2
#define ACS_UNREACHABLE 3

PRIVATE struct arp_hash_ent
{
    arp_cache_t *ahe_row[ARP_HASH_WIDTH];
} arp_hash[ARP_HASH_NR];

PRIVATE arp_port_t *arp_port_table;
PRIVATE arp_cache_t *arp_cache;
PRIVATE int arp_cache_nr;

FORWARD acc_t *arp_getdata ARGS(( int fd, size_t offset,
    size_t count, int for_ioctl ));
FORWARD int arp_putdata ARGS(( int fd, size_t offset,
    acc_t *data, int for_ioctl ));
FORWARD void arp_main ARGS(( arp_port_t *arp_port ));
FORWARD void arp_timeout ARGS(( int ref, timer_t *timer ));
FORWARD void setup_write ARGS(( arp_port_t *arp_port ));
FORWARD void setup_read ARGS(( arp_port_t *arp_port ));
FORWARD void do_reclist ARGS(( event_t *ev, ev_arg_t ev_arg ));
FORWARD void process_arp_pkt ARGS(( arp_port_t *arp_port, acc_t *data ));
FORWARD void client_reply ARGS(( arp_port_t *arp_port,

```

```

        ipaddr_t ipaddr, ether_addr_t *ethaddr ));
FORWARD arp_cache_t *find_cache_ent ARGS(( arp_port_t *arp_port,
        ipaddr_t ipaddr ));
FORWARD arp_cache_t *alloc_cache_ent ARGS(( int flags ));
FORWARD void arp_buffree ARGS(( int priority ));
#ifdef BUF_CONSISTENCY_CHECK
FORWARD void arp_bufcheck ARGS(( void ));
#endif

PUBLIC void arp_prep()
{
    arp_port_table= alloc(eth_conf_nr * sizeof(arp_port_table[0]));

    arp_cache_nr= ARP_CACHE_NR;
    if (arp_cache_nr < (eth_conf_nr+1)*AP_REQ_NR)
    {
        arp_cache_nr= (eth_conf_nr+1)*AP_REQ_NR;
        printf("arp: using %d cache entries instead of %d\n",
                arp_cache_nr, ARP_CACHE_NR);
    }
    arp_cache= alloc(arp_cache_nr * sizeof(arp_cache[0]));
}

PUBLIC void arp_init()
{
    arp_port_t *arp_port;
    arp_cache_t *cache;
    int i;

    assert (BUF_S >= sizeof(struct nwio_ethstat));
    assert (BUF_S >= sizeof(struct nwio_ethopt));
    assert (BUF_S >= sizeof(arp46_t));

    for (i=0, arp_port= arp_port_table; i<eth_conf_nr; i++, arp_port++)
    {
        arp_port->ap_state= APS_ERROR; /* Mark all ports as
                                         * unavailable */
    }

    cache= arp_cache;
    for (i=0; i<arp_cache_nr; i++, cache++)
    {
        cache->ac_state= ACS_UNUSED;
        cache->ac_flags= ACF_EMPTY;
        cache->ac_expire= 0;
        cache->ac_lastuse= 0;
    }

#ifdef BUF_CONSISTENCY_CHECK
    bf_logon(arp_buffree);
#else
    bf_logon(arp_buffree, arp_bufcheck);
#endif
}

PRIVATE void arp_main(arp_port)
arp_port_t *arp_port;
{
    int result;

    switch (arp_port->ap_state)
    {
    case APS_INITIAL:
        arp_port->ap_eth_fd= eth_open(arp_port->ap_eth_port,
            arp_port->ap_eth_port, arp_getdata, arp_putdata,
            0 /* no put_pkt */, 0 /* no select_res */);

        if (arp_port->ap_eth_fd<0)
        {
            DBLOCK(1, printf("arp[%d]: unable to open eth[%d]\n",
                arp_port-arp_port_table,
                arp_port->ap_eth_port));
            return;
        }
    }
}

```

```

        arp_port->ap_state= APS_GETADDR;

        result= eth_ioctl (arp_port->ap_eth_fd, NWIOGETHSTAT);

        if ( result == NW_SUSPEND)
        {
            arp_port->ap_flags |= APF_SUSPEND;
            return;
        }
        assert(result == NW_OK);

        /* fall through */
    case APS_GETADDR:
        /* Wait for IP address */
        if (!(arp_port->ap_flags & APF_INADDR_SET))
            return;

        /* fall through */
    case APS_ARPSTART:
        arp_port->ap_state= APS_ARPPROTO;

        result= eth_ioctl (arp_port->ap_eth_fd, NWIOSETHOPT);

        if (result==NW_SUSPEND)
        {
            arp_port->ap_flags |= APF_SUSPEND;
            return;
        }
        assert(result == NW_OK);

        /* fall through */
    case APS_ARPPROTO:
        arp_port->ap_state= APS_ARPMAIN;
        setup_write(arp_port);
        setup_read(arp_port);
        return;

    default:
        ip_panic((
            "arp_main(&arp_port_table[%d]) called but ap_state=0x%x\n",
            arp_port->ap_eth_port, arp_port->ap_state ));
    }
}

PRIVATE acc_t *arp_getdata (fd, offset, count, for_ioctl)
int fd;
size_t offset;
size_t count;
int for_ioctl;
{
    arp_port_t *arp_port;
    acc_t *data;
    int result;

    arp_port= &arp_port_table[fd];

    switch (arp_port->ap_state)
    {
    case APS_ARPPROTO:
        if (!count)
        {
            result= (int)offset;
            if (result<0)
            {
                arp_port->ap_state= APS_ERROR;
                break;
            }
            if (arp_port->ap_flags & APF_SUSPEND)
            {
                arp_port->ap_flags &= ~APF_SUSPEND;
                arp_main(arp_port);
            }
            return NW_OK;
        }
    }
}

```

```

    }
    assert ((!offset) && (count == sizeof(struct nwio_ethopt)));
    {
        struct nwio_ethopt *ethopt;
        acc_t *acc;

        acc= bf_memreq(sizeof(*ethopt));
        ethopt= (struct nwio_ethopt *)ptr2acc_data(acc);
        ethopt->nweo_flags= NWEO_COPY|NWEO_EN_BROAD|
            NWEO_TYPESPEC;
        ethopt->nweo_type= HTONS(ETH_ARP_PROTO);
        return acc;
    }
case APS_ARPMAIN:
    assert (arp_port->ap_flags & APF_ARP_WR_IP);
    if (!count)
    {
        data= arp_port->ap_sendpkt;
        arp_port->ap_sendpkt= NULL;
        assert(data);
        bf_afree(data); data= NULL;

        result= (int)offset;
        if (result<0)
        {
            DIFBLOCK(1, (result != NW_SUSPEND),
                printf(
                    "arp[%d]: write error on port %d: error %d\n",
                    fd, arp_port->ap_eth_fd, result));

            arp_port->ap_state= APS_ERROR;
            break;
        }
        arp_port->ap_flags &= ~APF_ARP_WR_IP;
        if (arp_port->ap_flags & APF_ARP_WR_SP)
            setup_write(arp_port);
        return NW_OK;
    }
    assert (offset+count <= sizeof(arp46_t));
    data= arp_port->ap_sendpkt;
    assert(data);
    data= bf_cut(data, offset, count);

    return data;
default:
    printf("arp_getdata(%d, 0x%d, 0x%d) called but ap_state=0x%x\n",
        fd, offset, count, arp_port->ap_state);
    break;
}
return 0;
}

```

```

PRIVATE int arp_putdata (fd, offset, data, for_ioctl)
int fd;
size_t offset;
acc_t *data;
int for_ioctl;
{
    arp_port_t *arp_port;
    int result;
    struct nwio_ethstat *ethstat;
    ev_arg_t ev_arg;
    acc_t *tmpacc;

    arp_port= &arp_port_table[fd];

    if (arp_port->ap_flags & APF_ARP_RD_IP)
    {
        if (!data)
        {
            result= (int)offset;
            if (result<0)
            {
                DIFBLOCK(1, (result != NW_SUSPEND), printf(

```

```

        "arp[%d]: read error on port %d: error %d\n",
        fd, arp_port->ap_eth_fd, result));

    return NW_OK;
}
if (arp_port->ap_flags & APF_ARP_RD_SP)
{
    arp_port->ap_flags &= ~(APF_ARP_RD_IP |
        APF_ARP_RD_SP);
    setup_read(arp_port);
}
else
    arp_port->ap_flags &= ~(APF_ARP_RD_IP |
        APF_ARP_RD_SP);
return NW_OK;
}
assert (!offset);
/* Warning: the above assertion is illegal; puts and gets of
   data can be brokenup in any piece the server likes. However
   we assume that the server is eth.c and it transfers only
   whole packets.
*/
data= bf_packIfLess(data, sizeof(arp46_t));
if (data->acc_length >= sizeof(arp46_t))
{
    if (!arp_port->ap_reclist)
    {
        ev_arg.ev_ptr= arp_port;
        ev_enqueue(&arp_port->ap_event, do_reclist,
            ev_arg);
    }
    if (data->acc_linkC != 1)
    {
        tmpacc= bf_dupacc(data);
        bf_afree(data);
        data= tmpacc;
        tmpacc= NULL;
    }
    data->acc_ext_link= arp_port->ap_reclist;
    arp_port->ap_reclist= data;
}
else
    bf_afree(data);
return NW_OK;
}
switch (arp_port->ap_state)
{
case APS_GETADDR:
    if (!data)
    {
        result= (int)offset;
        if (result<0)
        {
            arp_port->ap_state= APS_ERROR;
            break;
        }
        if (arp_port->ap_flags & APF_SUSPEND)
        {
            arp_port->ap_flags &= ~APF_SUSPEND;
            arp_main(arp_port);
        }
        return NW_OK;
    }
    compare (bf_bufsize(data), ==, sizeof(*ethstat));
    data= bf_packIfLess(data, sizeof(*ethstat));
    compare (data->acc_length, ==, sizeof(*ethstat));
    ethstat= (struct nwio_ethstat *)ptr2acc_data(data);
    arp_port->ap_ethaddr= ethstat->nwes_addr;
    bf_afree(data);
    return NW_OK;
default:
    printf("arp_putdata(%d, 0x%d, 0x%lx) called but ap_state=0x%x\n",
        fd, offset, (unsigned long)data, arp_port->ap_state);
    break;
}

```

```

    }
    return EGENERIC;
}

PRIVATE void setup_read(arp_port)
arp_port_t *arp_port;
{
    int result;

    while (!(arp_port->ap_flags & APF_ARP_RD_IP))
    {
        arp_port->ap_flags |= APF_ARP_RD_IP;
        result= eth_read (arp_port->ap_eth_fd, ETH_MAX_PACK_SIZE);
        if (result == NW_SUSPEND)
        {
            arp_port->ap_flags |= APF_ARP_RD_SP;
            return;
        }
        DIFBLOCK(1, (result != NW_OK),
            printf("arp[%d]: eth_read(...,%d)=%d\n",
                arp_port-arp_port_table, ETH_MAX_PACK_SIZE, result));
    }
}

PRIVATE void setup_write(arp_port)
arp_port_t *arp_port;
{
    int result;
    acc_t *data;

    for(;;)
    {
        data= arp_port->ap_sendlist;
        if (!data)
            break;
        arp_port->ap_sendlist= data->acc_ext_link;

        if (arp_port->ap_ipaddr == HTONL(0x00000000))
        {
            /* Interface is down */
            printf(
                "arp[%d]: not sending ARP packet, interface is down\n",
                arp_port-arp_port_table);
            bf_afree(data); data= NULL;
            continue;
        }

        assert(!arp_port->ap_sendpkt);
        arp_port->ap_sendpkt= data; data= NULL;

        arp_port->ap_flags= (arp_port->ap_flags & ~APF_ARP_WR_SP) |
            APF_ARP_WR_IP;
        result= eth_write(arp_port->ap_eth_fd, sizeof(arp46_t));
        if (result == NW_SUSPEND)
        {
            arp_port->ap_flags |= APF_ARP_WR_SP;
            break;
        }
        if (result<0)
        {
            DIFBLOCK(1, (result != NW_SUSPEND),
                printf("arp[%d]: eth_write(...,%d)=%d\n",
                    arp_port-arp_port_table, sizeof(arp46_t),
                    result));
            return;
        }
    }
}

PRIVATE void do_reclist(ev, ev_arg)
event_t *ev;
ev_arg_t ev_arg;
{
    arp_port_t *arp_port;

```



```

    acc_t *data;

    arp_port= ev_arg.ev_ptr;
    assert(ev == &arp_port->ap_event);

    while (data= arp_port->ap_reclist, data != NULL)
    {
        arp_port->ap_reclist= data->acc_ext_link;
        process_arp_pkt(arp_port, data);
        bf_afree(data);
    }
}

PRIVATE void process_arp_pkt (arp_port, data)
arp_port_t *arp_port;
acc_t *data;
{
    int i, entry, do_reply;
    arp46_t *arp;
    ul6_t *p;
    arp_cache_t *ce, *cache;
    struct arp_req *reqp;
    time_t curr_time;
    ipaddr_t spa, tpa;

    curr_time= get_time();

    arp= (arp46_t *)ptr2acc_data(data);
    memcpy(&spa, arp->a46_spa, sizeof(ipaddr_t));
    memcpy(&tpa, arp->a46_tpa, sizeof(ipaddr_t));

    if (arp->a46_hdr != HTONS(ARP_ETHERNET) ||
        arp->a46_hln != 6 ||
        arp->a46_pro != HTONS(ETH_IP_PROTO) ||
        arp->a46_pln != 4)
        return;
    if (arp_port->ap_ipaddr == HTONL(0x00000000))
    {
        /* Interface is down */
#ifdef DEBUG
        printf("arp[%d]: dropping ARP packet, interface is down\n",
            arp_port-arp_port_table);
#endif
        return;
    }

    ce= find_cache_ent(arp_port, spa);
    cache= NULL; /* lint */

    do_reply= 0;
    if (arp->a46_op != HTONS(ARP_REQUEST))
        ; /* No need to reply */
    else if (tpa == arp_port->ap_ipaddr)
        do_reply= 1;
    else
    {
        /* Look for a published entry */
        cache= find_cache_ent(arp_port, tpa);
        if (cache)
        {
            if (cache->ac_flags & ACF_PUB)
            {
                /* Published entry */
                do_reply= 1;
            }
            else
            {
                /* Nothing to do */
                cache= NULL;
            }
        }
    }

    if (ce == NULL)

```

```

{
    if (!do_reply)
        return;

    DBLOCK(0x10, printf("arp[%d]: allocating entry for ",
        arp_port-arp_port_table);
        writeIpAddr(spa); printf("\n"));

    ce= alloc_cache_ent(ACF_EMPTY);
    ce->ac_flags= ACF_EMPTY;
    ce->ac_state= ACS_VALID;
    ce->ac_ethaddr= arp->a46_sha;
    ce->ac_ipaddr= spa;
    ce->ac_port= arp_port;
    ce->ac_expire= curr_time+ARP_EXP_TIME;
    ce->ac_lastuse= curr_time-ARP_INUSE_OFFSET; /* never used */
}

if (ce->ac_state == ACS_INCOMPLETE || ce->ac_state == ACS_UNREACHABLE)
{
    ce->ac_ethaddr= arp->a46_sha;
    if (ce->ac_state == ACS_INCOMPLETE)
    {
        /* Find request entry */
        entry= ce->arp_cache;
        for (i= 0, reqp= arp_port->ap_req; i<AP_REQ_NR;
            i++, reqp++)
        {
            if (reqp->ar_entry == entry)
                break;
        }
        assert(i < AP_REQ_NR);
        clk_untimer(&reqp->ar_timer);
        reqp->ar_entry= -1;

        ce->ac_state= ACS_VALID;
        client_reply(arp_port, spa, &arp->a46_sha);
    }
    else
        ce->ac_state= ACS_VALID;
}

/* Update fields in the arp cache. */
if (memcmp(&ce->ac_ethaddr, &arp->a46_sha,
    sizeof(ce->ac_ethaddr)) != 0)
{
    printf("arp[%d]: ethernet address for IP address ",
        arp_port-arp_port_table);
    writeIpAddr(spa);
    printf(" changed from ");
    writeEtherAddr(&ce->ac_ethaddr);
    printf(" to ");
    writeEtherAddr(&arp->a46_sha);
    printf("\n");
    ce->ac_ethaddr= arp->a46_sha;
}
ce->ac_expire= curr_time+ARP_EXP_TIME;

if (do_reply)
{
    data= bf_memreq(sizeof(arp46_t));
    arp= (arp46_t *)ptr2acc_data(data);

    /* Clear padding */
    assert(sizeof(arp->a46_data.a46_dummy) % sizeof(*p) == 0);
    for (i= 0, p= (ul6_t *)arp->a46_data.a46_dummy;
        i < sizeof(arp->a46_data.a46_dummy)/sizeof(*p);
        i++, p++)
    {
        *p= 0xdead;
    }

    arp->a46_dstaddr= ce->ac_ethaddr;
    arp->a46_hdr= HTONS(ARP_ETHERNET);
}

```

```

    arp->a46_pro= HTONS(ETH_IP_PROTO);
    arp->a46_hln= 6;
    arp->a46_pln= 4;

    arp->a46_op= htons(ARP_REPLY);
    if (tpa == arp_port->ap_ipaddr)
    {
        arp->a46_sha= arp_port->ap_ethaddr;
    }
    else
    {
        assert(cache);
        arp->a46_sha= cache->ac_ethaddr;
    }
    memcpy (arp->a46_spa, &tpa, sizeof(ipaddr_t));
    arp->a46_tha= ce->ac_ethaddr;
    memcpy (arp->a46_tpa, &ce->ac_ipaddr, sizeof(ipaddr_t));

    assert(data->acc_linkC == 1);
    data->acc_ext_link= arp_port->ap_sendlist;
    arp_port->ap_sendlist= data; data= NULL;

    if (!(arp_port->ap_flags & APF_ARP_WR_IP))
        setup_write(arp_port);
}

}

PRIVATE void client_reply (arp_port, ipaddr, ethaddr)
arp_port_t *arp_port;
ipaddr_t ipaddr;
ether_addr_t *ethaddr;
{
    (*arp_port->ap_arp_func)(arp_port->ap_ip_port, ipaddr, ethaddr);
}

PRIVATE arp_cache_t *find_cache_ent (arp_port, ipaddr)
arp_port_t *arp_port;
ipaddr_t ipaddr;
{
    arp_cache_t *ce;
    int i;
    unsigned hash;

    hash= (ipaddr >> 24) ^ (ipaddr >> 16) ^ (ipaddr >> 8) ^ ipaddr;
    hash &= ARP_HASH_MASK;

    ce= arp_hash[hash].ahe_row[0];
    if (ce && ce->ac_ipaddr == ipaddr && ce->ac_port == arp_port &&
        ce->ac_state != ACS_UNUSED)
    {
        return ce;
    }
    for (i= 1; i<ARP_HASH_WIDTH; i++)
    {
        ce= arp_hash[hash].ahe_row[i];
        if (!ce || ce->ac_ipaddr != ipaddr || ce->ac_port != arp_port ||
            ce->ac_state == ACS_UNUSED)
        {
            continue;
        }
        arp_hash[hash].ahe_row[i]= arp_hash[hash].ahe_row[0];
        arp_hash[hash].ahe_row[0]= ce;
        return ce;
    }

    for (i=0, ce= arp_cache; i<arp_cache_nr; i++, ce++)
    {
        if (ce->ac_state != ACS_UNUSED &&
            ce->ac_port == arp_port &&
            ce->ac_ipaddr == ipaddr)
        {
            for (i= ARP_HASH_WIDTH-1; i>0; i--)
            {
                arp_hash[hash].ahe_row[i]=

```

```

        arp_hash[hash].ahe_row[i-1];
    }
    assert(i == 0);
    arp_hash[hash].ahe_row[0] = ce;
    return ce;
}
}
return NULL;
}
}

PRIVATE arp_cache_t *alloc_cache_ent(flags)
int flags;
{
    arp_cache_t *cache, *old;
    int i;

    old = NULL;
    for (i=0, cache= arp_cache; i<arp_cache_nr; i++, cache++)
    {
        if (cache->ac_state == ACS_UNUSED)
        {
            old = cache;
            break;
        }
        if (cache->ac_state == ACS_INCOMPLETE)
            continue;
        if (cache->ac_flags & ACF_PERM)
            continue;
        if (!old || cache->ac_lastuse < old->ac_lastuse)
            old = cache;
    }
    assert(old);

    if (!flags)
        return old;

    /* Get next permanent entry */
    for (i=0, cache= arp_cache; i<arp_cache_nr; i++, cache++)
    {
        if (cache->ac_state == ACS_UNUSED)
            break;
        if (cache->ac_flags & ACF_PERM)
            continue;
        break;
    }
    if (i >= arp_cache_nr/2)
        return NULL; /* Too many entries */
    if (cache != old)
    {
        assert(old > cache);
        *old = *cache;
        old = cache;
    }

    if (!(flags & ACF_PUB))
        return old;

    /* Get first nonpublished entry */
    for (i=0, cache= arp_cache; i<arp_cache_nr; i++, cache++)
    {
        if (cache->ac_state == ACS_UNUSED)
            break;
        if (cache->ac_flags & ACF_PUB)
            continue;
        break;
    }
    if (cache != old)
    {
        assert(old > cache);
        *old = *cache;
        old = cache;
    }
    return old;
}

```

```
PUBLIC void arp_set_ipaddr (eth_port, ipaddr)
int eth_port;
ipaddr_t ipaddr;
{
    arp_port_t *arp_port;

    if (eth_port < 0 || eth_port >= eth_conf_nr)
        return;
    arp_port = &arp_port_table[eth_port];

    arp_port->ap_ipaddr = ipaddr;
    arp_port->ap_flags |= APF_INADDR_SET;
    arp_port->ap_flags &= ~APF_SUSPEND;
    if (arp_port->ap_state == APS_GETADDR)
        arp_main(arp_port);
}

PUBLIC int arp_set_cb(eth_port, ip_port, arp_func)
int eth_port;
int ip_port;
arp_func_t arp_func;
{
    int i;
    arp_port_t *arp_port;

    assert(eth_port >= 0);
    if (eth_port >= eth_conf_nr)
        return ENXIO;

    arp_port = &arp_port_table[eth_port];
    arp_port->ap_eth_port = eth_port;
    arp_port->ap_ip_port = ip_port;
    arp_port->ap_state = APS_INITIAL;
    arp_port->ap_flags = APF_EMPTY;
    arp_port->ap_arp_func = arp_func;
    arp_port->ap_sendpkt = NULL;
    arp_port->ap_sendlist = NULL;
    arp_port->ap_reclist = NULL;
    for (i = 0; i < AP_REQ_NR; i++)
        arp_port->ap_req[i].ar_entry = -1;
    ev_init(&arp_port->ap_event);

    arp_main(arp_port);

    return NW_OK;
}

PUBLIC int arp_ip_eth (eth_port, ipaddr, ethaddr)
int eth_port;
ipaddr_t ipaddr;
ether_addr_t *ethaddr;
{
    int i, ref;
    arp_port_t *arp_port;
    struct arp_req *reqp;
    arp_cache_t *ce;
    time_t curr_time;

    assert(eth_port >= 0 && eth_port < eth_conf_nr);
    arp_port = &arp_port_table[eth_port];
    assert(arp_port->ap_state == APS_ARPMAIN ||
        (printf("arp[%d]: ap_state= %d\n", arp_port-arp_port_table,
            arp_port->ap_state), 0));

    curr_time = get_time();

    ce = find_cache_ent (arp_port, ipaddr);
    if (ce && ce->ac_expire < curr_time)
    {
        assert(ce->ac_state != ACS_INCOMPLETE);

        /* Check whether there is enough space for an ARP
         * request or not.

```

```

        */
        for (i= 0, reqp= arp_port->ap_req; i<AP_REQ_NR; i++, reqp++)
        {
            if (reqp->ar_entry < 0)
                break;
        }
        if (i < AP_REQ_NR)
        {
            /* Okay, expire this entry. */
            ce->ac_state= ACS_UNUSED;
            ce= NULL;
        }
        else
        {
            /* Continue using this entry for a while */
            printf("arp[%d]: Overloaded! Keeping entry for ",
                arp_port->arp_port_table);
            writeIpAddr(ipaddr);
            printf("\n");
            ce->ac_expire= curr_time+ARP_NOTRCH_EXP_TIME;
        }
    }
    if (ce)
    {
        /* Found an entry. This entry should be valid, unreachable
        * or incomplete.
        */
        ce->ac_lastuse= curr_time;
        if (ce->ac_state == ACS_VALID)
        {
            *ethaddr= ce->ac_ethaddr;
            return NW_OK;
        }
        if (ce->ac_state == ACS_UNREACHABLE)
            return EDSTNOTRCH;
        assert(ce->ac_state == ACS_INCOMPLETE);

        return NW_SUSPEND;
    }

    /* Find an empty slot for an ARP request */
    for (i= 0, reqp= arp_port->ap_req; i<AP_REQ_NR; i++, reqp++)
    {
        if (reqp->ar_entry < 0)
            break;
    }
    if (i >= AP_REQ_NR)
    {
        /* We should be able to report that this ARP request
        * cannot be accepted. At the moment we just return SUSPEND.
        */
        return NW_SUSPEND;
    }
    ref= (eth_port*AP_REQ_NR + i);

    ce= alloc_cache_ent(ACF_EMPTY);
    ce->ac_flags= 0;
    ce->ac_state= ACS_INCOMPLETE;
    ce->ac_ipaddr= ipaddr;
    ce->ac_port= arp_port;
    ce->ac_expire= curr_time+ARP_EXP_TIME;
    ce->ac_lastuse= curr_time;

    reqp->ar_entry= ce->arp_cache;
    reqp->ar_req_count= -1;

    /* Send the first packet by expiring the timer */
    clk_timer(&reqp->ar_timer, 1, arp_timeout, ref);

    return NW_SUSPEND;
}

```

```

PUBLIC int arp_ioctl (eth_port, fd, req, get_userdata, put_userdata)
int eth_port;

```

```

int fd;
ioreq_t req;
get_userdata_t get_userdata;
put_userdata_t put_userdata;
{
    arp_port_t *arp_port;
    arp_cache_t *ce, *cache;
    acc_t *data;
    nwio_arp_t *arp_iop;
    int entno, result, ac_flags;
    u32_t flags;
    ipaddr_t ipaddr;
    time_t curr_time;

    assert(eth_port >= 0 && eth_port < eth_conf_nr);
    arp_port = &arp_port_table[eth_port];
    assert(arp_port->ap_state == APS_ARPMAIN ||
        (printf("arp[%d]: ap_state= %d\n", arp_port->ap_state), 0));

    switch(req)
    {
    case NWIOARPGIP:
        data = (*get_userdata)(fd, 0, sizeof(*arp_iop), TRUE);
        if (data == NULL)
            return EFAULT;
        data = bf_packIfLess(data, sizeof(*arp_iop));
        arp_iop = (nwio_arp_t *)ptr2acc_data(data);
        ipaddr = arp_iop->nwa_ipaddr;
        ce = NULL; /* lint */
        for (entno = 0; entno < arp_cache_nr; entno++)
        {
            ce = &arp_cache[entno];
            if (ce->ac_state == ACS_UNUSED ||
                ce->ac_port != arp_port)
            {
                continue;
            }
            if (ce->ac_ipaddr == ipaddr)
                break;
        }
        if (entno == arp_cache_nr)
        {
            /* Also report the address of this interface */
            if (ipaddr != arp_port->ap_ipaddr)
            {
                bf_afree(data);
                return ENOENT;
            }
            arp_iop->nwa_entno = arp_cache_nr;
            arp_iop->nwa_ipaddr = ipaddr;
            arp_iop->nwa_ethaddr = arp_port->ap_ethaddr;
            arp_iop->nwa_flags = NWAFF_PERM | NWAFF_PUB;
        }
        else
        {
            arp_iop->nwa_entno = entno+1;
            arp_iop->nwa_ipaddr = ce->ac_ipaddr;
            arp_iop->nwa_ethaddr = ce->ac_ethaddr;
            arp_iop->nwa_flags = 0;
            if (ce->ac_state == ACS_INCOMPLETE)
                arp_iop->nwa_flags |= NWAFF_INCOMPLETE;
            if (ce->ac_state == ACS_UNREACHABLE)
                arp_iop->nwa_flags |= NWAFF_DEAD;
            if (ce->ac_flags & ACF_PERM)
                arp_iop->nwa_flags |= NWAFF_PERM;
            if (ce->ac_flags & ACF_PUB)
                arp_iop->nwa_flags |= NWAFF_PUB;
        }

        result = (*put_userdata)(fd, 0, data, TRUE);
        return result;

    case NWIOARPGNEXT:

```

```

data= (*get_userdata)(fd, 0, sizeof(*arp_iop), TRUE);
if (data == NULL)
    return EFAULT;
data= bf_packIfLess(data, sizeof(*arp_iop));
arp_iop= (nwio_arp_t *)ptr2acc_data(data);
entno= arp_iop->nwa_entno;
if (entno < 0)
    entno= 0;
ce= NULL; /* lint */
for (; entno < arp_cache_nr; entno++)
{
    ce= &arp_cache[entno];
    if (ce->ac_state == ACS_UNUSED ||
        ce->ac_port != arp_port)
    {
        continue;
    }
    break;
}
if (entno == arp_cache_nr)
{
    bf_afree(data);
    return ENOENT;
}
arp_iop->nwa_entno= entno+1;
arp_iop->nwa_ipaddr= ce->ac_ipaddr;
arp_iop->nwa_ethaddr= ce->ac_ethaddr;
arp_iop->nwa_flags= 0;
if (ce->ac_state == ACS_INCOMPLETE)
    arp_iop->nwa_flags |= NWAf_INCOMPLETE;
if (ce->ac_state == ACS_UNREACHABLE)
    arp_iop->nwa_flags |= NWAf_DEAD;
if (ce->ac_flags & ACF_PERM)
    arp_iop->nwa_flags |= NWAf_PERM;
if (ce->ac_flags & ACF_PUB)
    arp_iop->nwa_flags |= NWAf_PUB;

result= (*put_userdata)(fd, 0, data, TRUE);
return result;

case NWIOARPSIP:
data= (*get_userdata)(fd, 0, sizeof(*arp_iop), TRUE);
if (data == NULL)
    return EFAULT;
data= bf_packIfLess(data, sizeof(*arp_iop));
arp_iop= (nwio_arp_t *)ptr2acc_data(data);
ipaddr= arp_iop->nwa_ipaddr;
if (find_cache_ent(arp_port, ipaddr))
{
    bf_afree(data);
    return EEXIST;
}

flags= arp_iop->nwa_flags;
ac_flags= ACF_EMPTY;
if (flags & NWAf_PERM)
    ac_flags |= ACF_PERM;
if (flags & NWAf_PUB)
    ac_flags |= ACF_PUB|ACF_PERM;

/* Allocate a cache entry */
ce= alloc_cache_ent(ac_flags);
if (ce == NULL)
{
    bf_afree(data);
    return ENOMEM;
}

ce->ac_flags= ac_flags;
ce->ac_state= ACS_VALID;
ce->ac_ethaddr= arp_iop->nwa_ethaddr;
ce->ac_ipaddr= arp_iop->nwa_ipaddr;
ce->ac_port= arp_port;

```



```
curr_time= get_time();
ce->ac_expire= curr_time+ARP_EXP_TIME;
ce->ac_lastuse= curr_time;

bf_afree(data);
return 0;

case NWIOARPDIP:
    data= (*get_userdata)(fd, 0, sizeof(*arp_iop), TRUE);
    if (data == NULL)
        return EFAULT;
    data= bf_packIfLess(data, sizeof(*arp_iop));
    arp_iop= (nwio_arp_t *)ptr2acc_data(data);
    ipaddr= arp_iop->nwa_ipaddr;
    bf_afree(data); data= NULL;
    ce= find_cache_ent(arp_port, ipaddr);
    if (!ce)
        return ENOENT;
    if (ce->ac_state == ACS_INCOMPLETE)
        return EINVAL;

    ac_flags= ce->ac_flags;
    if (ac_flags & ACF_PUB)
    {
        /* Make sure entry is at the end of published
         * entries.
         */
        for (entno= 0, cache= arp_cache;
             entno<arp_cache_nr; entno++, cache++)
        {
            if (cache->ac_state == ACS_UNUSED)
                break;
            if (cache->ac_flags & ACF_PUB)
                continue;
            break;
        }
        assert(cache > arp_cache);
        cache--;
        if (cache != ce)
        {
            assert(cache > ce);
            *ce= *cache;
            ce= cache;
        }
    }
    if (ac_flags & ACF_PERM)
    {
        /* Make sure entry is at the end of permanent
         * entries.
         */
        for (entno= 0, cache= arp_cache;
             entno<arp_cache_nr; entno++, cache++)
        {
            if (cache->ac_state == ACS_UNUSED)
                break;
            if (cache->ac_flags & ACF_PERM)
                continue;
            break;
        }
        assert(cache > arp_cache);
        cache--;
        if (cache != ce)
        {
            assert(cache > ce);
            *ce= *cache;
            ce= cache;
        }
    }

    /* Clear entry */
    ce->ac_state= ACS_UNUSED;

    return 0;
```

```

    default:
        ip_panic(("arp_ioctl: unknown request 0x%lx",
                (unsigned long)req));
    }
    return 0;
}

PRIVATE void arp_timeout (ref, timer)
int ref;
timer_t *timer;
{
    int i, port, reqind, acind;
    arp_port_t *arp_port;
    arp_cache_t *ce;
    struct arp_req *reqp;
    time_t curr_time;
    acc_t *data;
    arp46_t *arp;
    ul6_t *p;

    port= ref / AP_REQ_NR;
    reqind= ref % AP_REQ_NR;

    assert(port >= 0 && port < eth_conf_nr);
    arp_port= &arp_port_table[port];

    reqp= &arp_port->ap_req[reqind];
    assert (timer == &reqp->ar_timer);

    acind= reqp->ar_entry;

    assert(acind >= 0 && acind < arp_cache_nr);
    ce= &arp_cache[acind];

    assert(ce->ac_port == arp_port);
    assert(ce->ac_state == ACS_INCOMPLETE);

    if (++reqp->ar_req_count >= MAX_ARP_RETRIES)
    {
        curr_time= get_time();
        ce->ac_state= ACS_UNREACHABLE;
        ce->ac_expire= curr_time+ ARP_NOTRCH_EXP_TIME;
        ce->ac_lastuse= curr_time;

        clk_untimer(&reqp->ar_timer);
        reqp->ar_entry= -1;
        client_reply(arp_port, ce->ac_ipaddr, NULL);
        return;
    }

    data= bf_memreq(sizeof(arp46_t));
    arp= (arp46_t *)ptr2acc_data(data);

    /* Clear padding */
    assert(sizeof(arp->a46_data.a46_dummy) % sizeof(*p) == 0);
    for (i= 0, p= (ul6_t *)arp->a46_data.a46_dummy;
        i < sizeof(arp->a46_data.a46_dummy)/sizeof(*p);
        i++, p++)
    {
        *p= 0xdead;
    }

    arp->a46_dstaddr.ea_addr[0]= 0xff;
    arp->a46_dstaddr.ea_addr[1]= 0xff;
    arp->a46_dstaddr.ea_addr[2]= 0xff;
    arp->a46_dstaddr.ea_addr[3]= 0xff;
    arp->a46_dstaddr.ea_addr[4]= 0xff;
    arp->a46_dstaddr.ea_addr[5]= 0xff;
    arp->a46_hdr= HTONS(ARP_ETHERNET);
    arp->a46_pro= HTONS(ETH_IP_PROTO);
    arp->a46_hln= 6;
    arp->a46_pln= 4;
    arp->a46_op= HTONS(ARP_REQUEST);
    arp->a46_sha= arp_port->ap_ethaddr;

```

```

memcpy (arp->a46_spa, &arp_port->ap_ipaddr, sizeof(ipaddr_t));
memset(&arp->a46_tha, '\0', sizeof(ether_addr_t));
memcpy (arp->a46_tpa, &ce->ac_ipaddr, sizeof(ipaddr_t));

assert(data->acc_linkC == 1);
data->acc_ext_link= arp_port->ap_sendlist;
arp_port->ap_sendlist= data; data= NULL;

if (!(arp_port->ap_flags & APF_ARP_WR_IP))
    setup_write(arp_port);

clk_timer(&reqp->ar_timer, get_time() + ARP_TIMEOUT,
          arp_timeout, ref);
}

PRIVATE void arp_buffree(priority)
int priority;
{
    int i;
    acc_t *pack, *next_pack;
    arp_port_t *arp_port;

    for (i= 0, arp_port= arp_port_table; i<eth_conf_nr; i++, arp_port++)
    {
        if (priority == ARP_PRI_REC)
        {
            next_pack= arp_port->ap_reclist;
            while(next_pack && next_pack->acc_ext_link)
            {
                pack= next_pack;
                next_pack= pack->acc_ext_link;
                bf_afree(pack);
            }
            if (next_pack)
            {
                if (ev_in_queue(&arp_port->ap_event))
                {
                    DBLOCK(1, printf(
                        "not freeing ap_reclist, ap_event enqueued\n"));
                }
                else
                {
                    bf_afree(next_pack);
                    next_pack= NULL;
                }
            }
            arp_port->ap_reclist= next_pack;
        }
        if (priority == ARP_PRI_SEND)
        {
            next_pack= arp_port->ap_sendlist;
            while(next_pack && next_pack->acc_ext_link)
            {
                pack= next_pack;
                next_pack= pack->acc_ext_link;
                bf_afree(pack);
            }
            if (next_pack)
            {
                if (ev_in_queue(&arp_port->ap_event))
                {
                    DBLOCK(1, printf(
                        "not freeing ap_sendlist, ap_event enqueued\n"));
                }
                else
                {
                    bf_afree(next_pack);
                    next_pack= NULL;
                }
            }
            arp_port->ap_sendlist= next_pack;
        }
    }
}

```

```
#ifndef BUF_CONSISTENCY_CHECK
PRIVATE void arp_bufcheck()
{
    int i;
    arp_port_t *arp_port;
    acc_t *pack;

    for (i= 0, arp_port= arp_port_table; i<eth_conf_nr; i++, arp_port++)
    {
        for (pack= arp_port->ap_reclist; pack;
             pack= pack->acc_ext_link)
        {
            bf_check_acc(pack);
        }
        for (pack= arp_port->ap_sendlist; pack;
             pack= pack->acc_ext_link)
        {
            bf_check_acc(pack);
        }
    }
}
#endif /* BUF_CONSISTENCY_CHECK */

/*
 * $PchId: arp.c,v 1.22 2005/06/28 14:15:06 philip Exp $
 */
```

```
/*
arp.h

Copyright 1995 Philip Homburg
*/

#ifndef ARP_H
#define ARP_H

#define ARP_ETHERNET      1

#define ARP_REQUEST       1
#define ARP_REPLY         2

/* Prototypes */
typedef void (*arp_func_t) ARGS(( int fd, ipaddr_t ipaddr,
                                ether_addr_t *ethaddr ));

void arp_prep ARGS(( void ));
void arp_init ARGS(( void ));
void arp_set_ipaddr ARGS(( int eth_port, ipaddr_t ipaddr ));
int arp_set_cb ARGS(( int eth_port, int ip_port, arp_func_t arp_func ));
int arp_ip_eth ARGS(( int eth_port, ipaddr_t ipaddr, ether_addr_t *ethaddr ));

int arp_ioctl ARGS(( int eth_port, int fd, ioreq_t req,
                    get_userdata_t get_userdata, put_userdata_t put_userdata ));

#endif /* ARP_H */

/*
 * $PchId: arp.h,v 1.7 2001/04/19 18:58:17 philip Exp $
 */
```

```
/*
assert.h

Copyright 1995 Philip Homburg
*/
#ifndef INET_ASSERT_H
#define INET_ASSERT_H

#if !NDEBUG

void bad_assertion(char *file, int line, char *what) _NORETURN;
void bad_compare(char *file, int line, int lhs, char *what, int rhs) _NORETURN;

#define assert(x) ((void)(!(x) ? bad_assertion(this_file, __LINE__, \
#x), 0 : 0))
#define compare(a,t,b) (!((a) t (b)) ? bad_compare(this_file, __LINE__, \
(a), #a " " #t " " #b, (b)) : (void) 0)

#else /* NDEBUG */

#define assert(x) 0
#define compare(a,t,b) 0

#endif /* NDEBUG */

#endif /* INET_ASSERT_H */

/*
 * $PchId: assert.h,v 1.8 2002/03/18 21:50:32 philip Exp $
 */
```

```
/*
buf.h

Copyright 1995 Philip Homburg
*/

#ifndef BUF_H
#define BUF_H

/* Note: BUF_S should be defined in const.h */

#define MAX_BUFREQ_PRI 10

#define ARP_PRI_REC 3
#define ARP_PRI_SEND 3

#define ETH_PRI_PORTBUFS 3
#define ETH_PRI_FDBUFS_EXTRA 5
#define ETH_PRI_FDBUFS 6

#define IP_PRI_PORTBUFS 3
#define IP_PRI_ASSBUFS 4
#define IP_PRI_FDBUFS_EXTRA 5
#define IP_PRI_FDBUFS 6

#define ICMP_PRI_QUEUE 1

#define TCP_PRI_FRAG2SEND 4
#define TCP_PRI_CONN_EXTRA 5
#define TCP_PRI_CONNWouser 7
#define TCP_PRI_CONN_INUSE 9

#define UDP_PRI_FDBUFS_EXTRA 5
#define UDP_PRI_FDBUFS 6

#define PSIP_PRI_EXP_PROMISC 2

struct acc;
typedef void (*buffree_t) ARGS(( struct acc *acc ));
typedef void (*bf_freereq_t) ARGS(( int priority ));

#ifdef BUF_CONSISTENCY_CHECK
typedef void (*bf_checkreq_t) ARGS(( void ));
#endif

typedef struct buf
{
    int buf_linkC;
    buffree_t buf_free;
    size_t buf_size;
    char *buf_data_p;

#ifdef BUF_TRACK_ALLOC_FREE
    char *buf_alloc_file;
    int buf_alloc_line;
    char *buf_free_file;
    int buf_free_line;
#endif
#ifdef BUF_CONSISTENCY_CHECK
    unsigned buf_generation;
    int buf_check_linkC;
#endif
} buf_t;

typedef struct acc
{
    int acc_linkC;
    int acc_offset, acc_length;
    buf_t *acc_buffer;
    struct acc *acc_next, *acc_ext_link;

#ifdef BUF_TRACK_ALLOC_FREE
    char *acc_alloc_file;
    int acc_alloc_line;
#endif
}
```

```

        char *acc_free_file;
        int acc_free_line;
#endif
#ifdef BUF_CONSISTENCY_CHECK
        unsigned acc_generation;
        int acc_check_linkC;
#endif
} acc_t;

extern acc_t *bf_temporary_acc;
extern acc_t *bf_linkcheck_acc;

/* For debugging... */

#ifdef BUF_TRACK_ALLOC_FREE

#ifndef BUF_IMPLEMENTATION

#define bf_memreq(a) _bf_memreq(this_file, __LINE__, a)
#define bf_cut(a,b,c) _bf_cut(this_file, __LINE__, a, b, c)
#define bf_delhead(a,b) _bf_delhead(this_file, __LINE__, a, b)
#define bf_packIffLess(a,b) _bf_packIffLess(this_file, __LINE__, \
a, b)

#define bf_afree(a) _bf_afree(this_file, __LINE__, a)
#define bf_pack(a) _bf_pack(this_file, __LINE__, a)
#define bf_append(a,b) _bf_append(this_file, __LINE__, a, b)
#define bf_dupacc(a) _bf_dupacc(this_file, __LINE__, a)
#define bf_mark_lacc(a) _bf_mark_lacc(this_file, __LINE__, a)
#define bf_mark_acc(a) _bf_mark_acc(this_file, __LINE__, a)
#endif
#define bf_align(a,s,al) _bf_align(this_file, __LINE__, a, s, al)

#else /* BUF_IMPLEMENTATION */

#define bf_afree(a) _bf_afree(clnt_file, clnt_line, a)
#define bf_pack(a) _bf_pack(clnt_file, clnt_line, a)
#define bf_memreq(a) _bf_memreq(clnt_file, clnt_line, a)
#define bf_dupacc(a) _bf_dupacc(clnt_file, clnt_line, a)
#define bf_cut(a,b,c) _bf_cut(clnt_file, clnt_line, a, b, c)
#define bf_delhead(a,b) _bf_delhead(clnt_file, clnt_line, a, b)
#define bf_align(a,s,al) _bf_align(clnt_file, clnt_line, a, s, al)

#endif /* !BUF_IMPLEMENTATION */

#else

#define bf_mark_lacc(acc) ((void)0)
#define bf_mark_acc(acc) ((void)0)

#endif /* BUF_TRACK_ALLOC_FREE */

/* Prototypes */

void bf_init ARGS(( void ));
#ifndef BUF_CONSISTENCY_CHECK
void bf_logon ARGS(( bf_freereq_t func ));
#else
void bf_logon ARGS(( bf_freereq_t func, bf_checkreq_t checkfunc ));
#endif

#ifndef BUF_TRACK_ALLOC_FREE
acc_t *bf_memreq ARGS(( unsigned size));
#else
acc_t *_bf_memreq ARGS(( char *clnt_file, int clnt_line,
unsigned size));
#endif

/* the result is an acc with linkC == 1 */

#ifndef BUF_TRACK_ALLOC_FREE
acc_t *bf_dupacc ARGS(( acc_t *acc ));
#else
acc_t *_bf_dupacc ARGS(( char *clnt_file, int clnt_line,
acc_t *acc ));

```



```
#endif
/* the result is an acc with linkC == 1 identical to the given one */

#ifdef BUF_TRACK_ALLOC_FREE
void bf_afree ARGS(( acc_t *acc));
#else
void _bf_afree ARGS(( char *clnt_file, int clnt_line,
                    acc_t *acc));
#endif
/* this reduces the linkC off the given acc with one */

#ifdef BUF_TRACK_ALLOC_FREE
acc_t *bf_pack ARGS(( acc_t *pack));
#else
acc_t *_bf_pack ARGS(( char *clnt_file, int clnt_line,
                    acc_t *pack));
#endif
/* this gives a packed copy of the given acc, the linkC of the given acc is
   reduced by one, the linkC of the result == 1 */

#ifdef BUF_TRACK_ALLOC_FREE
acc_t *bf_packIfLess ARGS(( acc_t *pack, int min_len ));
#else
acc_t *_bf_packIfLess ARGS(( char *clnt_file, int clnt_line,
                    acc_t *pack, int min_len ));
#endif
/* this performs a bf_pack iff pack->acc_length<min_len */

size_t bf_bufsize ARGS(( acc_t *pack));
/* this gives the length of the buffer specified by the given acc. The linkC
   of the given acc remains the same */

#ifdef BUF_TRACK_ALLOC_FREE
acc_t *bf_cut ARGS(( acc_t *data, unsigned offset, unsigned length ));
#else
acc_t *_bf_cut ARGS(( char *clnt_file, int clnt_line,
                    acc_t *data, unsigned offset, unsigned length ));
#endif
/* the result is a cut of the buffer from offset with length length.
   The linkC of the result == 1, the linkC of the given acc remains the
   same. */

#ifdef BUF_TRACK_ALLOC_FREE
acc_t *bf_delhead ARGS(( acc_t *data, unsigned offset ));
#else
acc_t *_bf_delhead ARGS(( char *clnt_file, int clnt_line,
                    acc_t *data, unsigned offset ));
#endif
/* the result is a cut of the buffer from offset until the end.
   The linkC of the result == 1, the linkC of the given acc is
   decremented. */

#ifdef BUF_TRACK_ALLOC_FREE
acc_t *bf_append ARGS(( acc_t *data_first, acc_t *data_second ));
#else
acc_t *_bf_append ARGS(( char *clnt_file, int clnt_line,
                    acc_t *data_first, acc_t *data_second ));
#endif
/* data_second is appended after data_first, a link is returned to the
   result and the linkCs of data_first and data_second are reduced.
   further more, if the contents of the last part of data_first and
   the first part of data_second fit in a buffer, both parts are
   copied into a (possibly fresh) buffer
*/

#ifdef BUF_TRACK_ALLOC_FREE
acc_t *bf_align ARGS(( acc_t *acc, size_t size, size_t alignment ));
#else
acc_t *_bf_align ARGS(( char *clnt_file, int clnt_line,
                    acc_t *acc, size_t size, size_t alignment ));
#endif
/* size bytes of acc (or all bytes of acc if the size buffer is smaller
   than size) are aligned on an address that is multiple of alignment.
   Size must be less than or equal to BUF_S.
```

```
*/

int bf_linkcheck ARGS(( acc_t *acc ));
/* check if all link count are positive, and offsets and sizes are within
 * the underlying buffer.
 */

#define ptr2acc_data(/* acc_t * */ a) (bf_temporary_acc=(a), \
    (&bf_temporary_acc->acc_buffer->buf_data_p[bf_temporary_acc-> \
    acc_offset]))

#define bf_chkbuf(buf) ((buf)? (compare((buf)->acc_linkC,>,0), \
    compare((buf)->acc_buffer, !=, 0), \
    compare((buf)->acc_buffer->buf_linkC,>,0)) : (void)0)

#ifdef BUF_CONSISTENCY_CHECK
int bf_consistency_check ARGS(( void ));
void bf_check_acc ARGS(( acc_t *acc ));
void _bf_mark_lacc ARGS(( char *clnt_file, int clnt_line, acc_t *acc ));
void _bf_mark_acc ARGS(( char *clnt_file, int clnt_line, acc_t *acc ));
#endif

#endif /* BUF_H */

/*
 * $PchId: buf.h,v 1.13 2003/09/10 08:52:09 philip Exp $
 */
```

```
/*
clock.h

Copyright 1995 Philip Homburg
*/

#ifndef CLOCK_H
#define CLOCK_H

struct timer;

typedef void (*timer_func_t) ARGS(( int fd, struct timer *timer ));

typedef struct timer
{
    struct timer *tim_next;
    timer_func_t tim_func;
    int tim_ref;
    time_t tim_time;
    int tim_active;
} timer_t;

extern int clk_call_expire;    /* Call clk_expire_timer from the mainloop */

void clk_init ARGS(( void ));
void set_time ARGS(( time_t time ));
time_t get_time ARGS(( void ));
void reset_time ARGS(( void ));
/* set a timer to go off at the time specified by timeout */
void clk_timer ARGS(( struct timer *timer, time_t timeout, timer_func_t func,
                      int fd ));

void clk_untimer ARGS(( struct timer *timer ));
void clk_expire_timers ARGS(( void ));

#endif /* CLOCK_H */

/*
 * $PchId: clock.h,v 1.5 1995/11/21 06:45:27 philip Exp $
 */
```

```

/*
eth.c

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "buf.h"
#include "clock.h"
#include "event.h"
#include "osdep_eth.h"
#include "type.h"

#include "assert.h"
#include "buf.h"
#include "eth.h"
#include "eth_int.h"
#include "io.h"
#include "sr.h"

THIS_FILE

#define ETH_FD_NR          (4*IP_PORT_MAX)
#define EXPIRE_TIME       60*HZ    /* seconds */

typedef struct eth_fd
{
    int ef_flags;
    nwio_ethopt_t ef_ethopt;
    eth_port_t *ef_port;
    struct eth_fd *ef_type_next;
    struct eth_fd *ef_send_next;
    int ef_srfd;
    acc_t *ef_rdbuf_head;
    acc_t *ef_rdbuf_tail;
    get_userdata_t ef_get_userdata;
    put_userdata_t ef_put_userdata;
    put_pkt_t ef_put_pkt;
    time_t ef_exp_time;
    size_t ef_write_count;
    ioreq_t ef_ioctl_req;
} eth_fd_t;

#define EFF_FLAGS          0xf
#   define EFF_EMPTY          0x0
#   define EFF_INUSE          0x1
#   define EFF_BUSY          0xE
#       define EFF_READ_IP      0x2
#       define EFF_WRITE_IP     0x4
#       define EFF_IOCTL_IP     0x8
#   define EFF_OPTSET         0x10

/* Note that the vh_type field is normally considered part of the ethernet
 * header.
 */
typedef struct
{
    ul6_t vh_type;
    ul6_t vh_vlan;
} vlan_hdr_t;

FORWARD int eth_checkopt ARGS(( eth_fd_t *eth_fd ));
FORWARD void hash_fd ARGS(( eth_fd_t *eth_fd ));
FORWARD void unhash_fd ARGS(( eth_fd_t *eth_fd ));
FORWARD void eth_buffree ARGS(( int priority ));
#ifdef BUF_CONSISTENCY_CHECK
FORWARD void eth_bufcheck ARGS(( void ));
#endif
FORWARD void packet2user ARGS(( eth_fd_t *fd, acc_t *pack, time_t exp_time ));
FORWARD void reply_thr_get ARGS(( eth_fd_t *eth_fd,
    size_t result, int for_ioctl ));
FORWARD void reply_thr_put ARGS(( eth_fd_t *eth_fd,
    size_t result, int for_ioctl ));
FORWARD void do_rec_conf ARGS(( eth_port_t *eth_port ));

```

```

FORWARD u32_t compute_rec_conf ARGS(( eth_port_t *eth_port ));
FORWARD acc_t *insert_vlan_hdr ARGS(( eth_port_t *eth_port, acc_t *pack ));

PUBLIC eth_port_t *eth_port_table;
PUBLIC int no_ethWritePort= 0;

PRIVATE eth_fd_t eth_fd_table[ETH_FD_NR];
PRIVATE ether_addr_t broadcast= { { 255, 255, 255, 255, 255, 255 } };

PUBLIC void eth_prep()
{
    eth_port_table= alloc(eth_conf_nr * sizeof(eth_port_table[0]));
}

PUBLIC void eth_init()
{
    int i, j;

    assert (BUF_S >= sizeof(nwio_ethopt_t));
    assert (BUF_S >= ETH_HDR_SIZE); /* these are in fact static assertions,
                                     thus a good compiler doesn't
                                     generate any code for this */

    for (i=0; i<ETH_FD_NR; i++)
        eth_fd_table[i].ef_flags= EFF_EMPTY;
    for (i=0; i<eth_conf_nr; i++)
    {
        eth_port_table[i].etp_flags= EFF_EMPTY;
        eth_port_table[i].etp_sendq_head= NULL;
        eth_port_table[i].etp_sendq_tail= NULL;
        eth_port_table[i].etp_type_any= NULL;
        ev_init(&eth_port_table[i].etp_sendev);
        for (j= 0; j<ETH_TYPE_HASH_NR; j++)
            eth_port_table[i].etp_type[j]= NULL;
        for (j= 0; j<ETH_VLAN_HASH_NR; j++)
            eth_port_table[i].etp_vlan_tab[j]= NULL;
    }

#ifdef BUF_CONSISTENCY_CHECK
    bf_logon(eth_buffree);
#else
    bf_logon(eth_buffree, eth_bufcheck);
#endif

    osdep_eth_init();
}

PUBLIC int eth_open(port, srfd, get_userdata, put_userdata, put_pkt,
    select_res)
int port, srfd;
get_userdata_t get_userdata;
put_userdata_t put_userdata;
put_pkt_t put_pkt;
select_res_t select_res;
{
    int i;
    eth_port_t *eth_port;
    eth_fd_t *eth_fd;

    DBLOCK(0x20, printf("eth_open(%d,%d,%lx,%lx)\n", port, srfd,
        (unsigned long)get_userdata, (unsigned long)put_userdata));
    eth_port= &eth_port_table[port];
    if (!(eth_port->etp_flags & EPF_ENABLED))
        return EGENERIC;

    for (i=0; i<ETH_FD_NR && (eth_fd_table[i].ef_flags & EFF_INUSE);
        i++);

    if (i>=ETH_FD_NR)
    {
        DBLOCK(1, printf("out of fds\n"));
        return EAGAIN;
    }
}

```

```

    eth_fd= &eth_fd_table[i];

    eth_fd->ef_flags= EFF_INUSE;
    eth_fd->ef_ethopt.nweo_flags=NWEO_DEFAULT;
    eth_fd->ef_port= eth_port;
    eth_fd->ef_srfd= srfd;
    assert(eth_fd->ef_rdbuf_head == NULL);
    eth_fd->ef_get_userdata= get_userdata;
    eth_fd->ef_put_userdata= put_userdata;
    eth_fd->ef_put_pkt= put_pkt;

    return i;
}

PUBLIC int eth_ioctl(fd, req)
int fd;
ioreq_t req;
{
    acc_t *data;
    eth_fd_t *eth_fd;
    eth_port_t *eth_port;

    DBLOCK(0x20, printf("eth_ioctl(%d, 0x%x)\n", fd, (unsigned long)req));
    eth_fd= &eth_fd_table[fd];
    eth_port= eth_fd->ef_port;

    assert(eth_fd->ef_flags & EFF_INUSE);

    switch (req)
    {
    case NWIOSETHOPT:
        {
            nwio_ethopt_t *ethopt;
            nwio_ethopt_t oldopt, newopt;
            int result;
            u32_t new_en_flags, new_di_flags,
                old_en_flags, old_di_flags;

            data= (*eth_fd->ef_get_userdata)(eth_fd->
                ef_srfd, 0, sizeof(nwio_ethopt_t), TRUE);

            ethopt= (nwio_ethopt_t *)ptr2acc_data(data);
            oldopt= eth_fd->ef_ethopt;
            newopt= *ethopt;

            old_en_flags= oldopt.nweo_flags & 0xffff;
            old_di_flags= (oldopt.nweo_flags >> 16) & 0xffff;
            new_en_flags= newopt.nweo_flags & 0xffff;
            new_di_flags= (newopt.nweo_flags >> 16) & 0xffff;
            if (new_en_flags & new_di_flags)
            {
                bf_afree(data);
                reply_thr_get (eth_fd, EBADMODE, TRUE);
                return NW_OK;
            }

            /* NWEO_ACC_MASK */
            if (new_di_flags & NWEO_ACC_MASK)
            {
                bf_afree(data);
                reply_thr_get (eth_fd, EBADMODE, TRUE);
                return NW_OK;
            }

            /* you can't disable access modes */

            if (!(new_en_flags & NWEO_ACC_MASK))
                new_en_flags |= (old_en_flags & NWEO_ACC_MASK);

            /* NWEO_LOC_MASK */
            if (((new_en_flags | new_di_flags) & NWEO_LOC_MASK))
            {
                new_en_flags |= (old_en_flags & NWEO_LOC_MASK);
            }
        }
    }
}

```

```
        new_di_flags |= (old_di_flags & NWEO_LOC_MASK);
    }

    /* NWEO_BROAD_MASK */
    if (!((new_en_flags | new_di_flags) & NWEO_BROAD_MASK))
    {
        new_en_flags |= (old_en_flags & NWEO_BROAD_MASK);
        new_di_flags |= (old_di_flags & NWEO_BROAD_MASK);
    }

    /* NWEO_MULTI_MASK */
    if (!((new_en_flags | new_di_flags) & NWEO_MULTI_MASK))
    {
        new_en_flags |= (old_en_flags & NWEO_MULTI_MASK);
        new_di_flags |= (old_di_flags & NWEO_MULTI_MASK);
        newopt.nweo_multi= oldopt.nweo_multi;
    }

    /* NWEO_PROMISC_MASK */
    if (!((new_en_flags | new_di_flags) & NWEO_PROMISC_MASK))
    {
        new_en_flags |= (old_en_flags & NWEO_PROMISC_MASK);
        new_di_flags |= (old_di_flags & NWEO_PROMISC_MASK);
    }

    /* NWEO_REM_MASK */
    if (!((new_en_flags | new_di_flags) & NWEO_REM_MASK))
    {
        new_en_flags |= (old_en_flags & NWEO_REM_MASK);
        new_di_flags |= (old_di_flags & NWEO_REM_MASK);
        newopt.nweo_rem= oldopt.nweo_rem;
    }

    /* NWEO_TYPE_MASK */
    if (!((new_en_flags | new_di_flags) & NWEO_TYPE_MASK))
    {
        new_en_flags |= (old_en_flags & NWEO_TYPE_MASK);
        new_di_flags |= (old_di_flags & NWEO_TYPE_MASK);
        newopt.nweo_type= oldopt.nweo_type;
    }

    /* NWEO_RW_MASK */
    if (!((new_en_flags | new_di_flags) & NWEO_RW_MASK))
    {
        new_en_flags |= (old_en_flags & NWEO_RW_MASK);
        new_di_flags |= (old_di_flags & NWEO_RW_MASK);
    }

    if (eth_fd->ef_flags & EFF_OPTSET)
        unhash_fd(eth_fd);

    newopt.nweo_flags= ((unsigned long)new_di_flags << 16) |
        new_en_flags;
    eth_fd->ef_ethopt= newopt;

    result= eth_checkopt(eth_fd);

    if (result<0)
        eth_fd->ef_ethopt= oldopt;
    else
    {
        unsigned long opt_flags;
        unsigned changes;
        opt_flags= oldopt.nweo_flags ^
            eth_fd->ef_ethopt.nweo_flags;
        changes= ((opt_flags >> 16) | opt_flags) &
            0xffff;
        if (changes & (NWEO_BROAD_MASK |
            NWEO_MULTI_MASK | NWEO_PROMISC_MASK))
        {
            do_rec_conf(eth_port);
        }
    }
}
```

```

        if (eth_fd->ef_flags & EFF_OPTSET)
            hash_fd(eth_fd);

        bf_afree(data);
        reply_thr_get (eth_fd, result, TRUE);
        return NW_OK;
    }

    case NWIOGETHOPT:
    {
        nwio_ethopt_t *ethopt;
        acc_t *acc;
        int result;

        acc= bf_memreq(sizeof(nwio_ethopt_t));

        ethopt= (nwio_ethopt_t *)ptr2acc_data(acc);

        *ethopt= eth_fd->ef_ethopt;

        result= (*eth_fd->ef_put_userdata)(eth_fd->
            ef_srfd, 0, acc, TRUE);
        if (result >= 0)
            reply_thr_put(eth_fd, NW_OK, TRUE);
        return result;
    }

    case NWIOGETHSTAT:
    {
        nwio_ethstat_t *ethstat;
        acc_t *acc;
        int result;

        assert (sizeof(nwio_ethstat_t) <= BUF_S);

        eth_port= eth_fd->ef_port;
        if (!(eth_port->etp_flags & EPF_ENABLED))
        {
            reply_thr_put(eth_fd, EGENERIC, TRUE);
            return NW_OK;
        }

        if (!(eth_port->etp_flags & EPF_GOT_ADDR))
        {
            printf(
                "eth_ioctl: suspending NWIOGETHSTAT ioctl\n" );

            eth_fd->ef_ioctl_req= req;
            assert(!(eth_fd->ef_flags & EFF_IOCTL_IP));
            eth_fd->ef_flags |= EFF_IOCTL_IP;
            return NW_SUSPEND;
        }

        acc= bf_memreq(sizeof(nwio_ethstat_t));
        compare (bf_bufsize(acc), ==, sizeof(*ethstat));

        ethstat= (nwio_ethstat_t *)ptr2acc_data(acc);
        ethstat->nwes_addr= eth_port->etp_ethaddr;

        if (!eth_port->etp_vlan)
        {
            result= eth_get_stat(eth_port,
                &ethstat->nwes_stat);
            if (result != NW_OK)
            {
                bf_afree(acc);
                reply_thr_put(eth_fd, result, TRUE);
                return NW_OK;
            }
        }
        else
        {
            /* No statistics */
            memset(&ethstat->nwes_stat, '\0',
                sizeof(ethstat->nwes_stat));

```



```

    }

    result= (*eth_fd->ef_put_userdata)(eth_fd->
        ef_srfd, 0, acc, TRUE);
    if (result >= 0)
        reply_thr_put(eth_fd, NW_OK, TRUE);
    return result;
}
default:
    break;
}
reply_thr_put(eth_fd, EBADIOCTL, TRUE);
return NW_OK;
}

PUBLIC int eth_write(fd, count)
int fd;
size_t count;
{
    eth_fd_t *eth_fd;
    eth_port_t *eth_port, *rep;
    acc_t *user_data;
    int r;

    eth_fd= &eth_fd_table[fd];
    eth_port= eth_fd->ef_port;

    if (!(eth_fd->ef_flags & EFF_OPTSET))
    {
        reply_thr_get (eth_fd, EBADMODE, FALSE);
        return NW_OK;
    }

    assert (!(eth_fd->ef_flags & EFF_WRITE_IP));

    eth_fd->ef_write_count= count;
    if (eth_fd->ef_ethopt.nweo_flags & NWEO_RWDATONLY)
        count += ETH_HDR_SIZE;

    if (count<ETH_MIN_PACK_SIZE || count>ETH_MAX_PACK_SIZE)
    {
        DBLOCK(1, printf("illegal packet size (%d)\n", count));
        reply_thr_get (eth_fd, EPACKSIZE, FALSE);
        return NW_OK;
    }
    eth_fd->ef_flags |= EFF_WRITE_IP;

    /* Enqueue at the real ethernet port */
    rep= eth_port->etp_vlan_port;
    if (!rep)
        rep= eth_port;
    if (rep->etp_wr_pack)
    {
        eth_fd->ef_send_next= NULL;
        if (rep->etp_sendq_head)
            rep->etp_sendq_tail->ef_send_next= eth_fd;
        else
            rep->etp_sendq_head= eth_fd;
        rep->etp_sendq_tail= eth_fd;
        return NW_SUSPEND;
    }

    user_data= (*eth_fd->ef_get_userdata)(eth_fd->ef_srfd, 0,
        eth_fd->ef_write_count, FALSE);
    if (!user_data)
    {
        eth_fd->ef_flags &= ~EFF_WRITE_IP;
        reply_thr_get (eth_fd, EFAULT, FALSE);
        return NW_OK;
    }
    r= eth_send(fd, user_data, eth_fd->ef_write_count);
    assert(r == NW_OK);

    eth_fd->ef_flags &= ~EFF_WRITE_IP;

```

```
    reply_thr_get(eth_fd, eth_fd->ef_write_count, FALSE);
    return NW_OK;
}

PUBLIC int eth_send(fd, data, data_len)
int fd;
acc_t *data;
size_t data_len;
{
    eth_fd_t *eth_fd;
    eth_port_t *eth_port, *rep;
    eth_hdr_t *eth_hdr;
    acc_t *eth_pack;
    unsigned long nweo_flags;
    size_t count;
    ev_arg_t ev_arg;

    eth_fd= &eth_fd_table[fd];
    eth_port= eth_fd->ef_port;

    if (!(eth_fd->ef_flags & EFF_OPTSET))
        return EBADMODE;

    count= data_len;
    if (eth_fd->ef_ethopt.nweo_flags & NWEO_RWDATONLY)
        count += ETH_HDR_SIZE;

    if (count<ETH_MIN_PACK_SIZE || count>ETH_MAX_PACK_SIZE)
    {
        DBLOCK(1, printf("illegal packet size (%d)\n", count));
        return EPACKSIZE;
    }
    rep= eth_port->etp_vlan_port;
    if (!rep)
        rep= eth_port;

    if (rep->etp_wr_pack)
        return NW_WOULDBLOCK;

    nweo_flags= eth_fd->ef_ethopt.nweo_flags;
    if (nweo_flags & NWEO_RWDATONLY)
    {
        eth_pack= bf_memreq(ETH_HDR_SIZE);
        eth_pack->acc_next= data;
    }
    else
        eth_pack= bf_packIffLess(data, ETH_HDR_SIZE);

    eth_hdr= (eth_hdr_t *)ptr2acc_data(eth_pack);

    if (nweo_flags & NWEO_REMSPEC)
        eth_hdr->eh_dst= eth_fd->ef_ethopt.nweo_rem;

    if (!(eth_port->etp_flags & EPF_GOT_ADDR))
    {
        /* No device, discard packet */
        bf_afree(eth_pack);
        return NW_OK;
    }

    if (!(nweo_flags & NWEO_EN_PROMISC))
        eth_hdr->eh_src= eth_port->etp_ethaddr;

    if (nweo_flags & NWEO_TYPESPEC)
        eth_hdr->eh_proto= eth_fd->ef_ethopt.nweo_type;

    if (eth_addrcmp(eth_hdr->eh_dst, eth_port->etp_ethaddr) == 0)
    {
        /* Local loopback. */
        eth_port->etp_wr_pack= eth_pack;
        ev_arg.ev_ptr= eth_port;
        ev_enqueue(&eth_port->etp_sendev, eth_loop_ev, ev_arg);
        return NW_OK;
    }
}
```

```

    if (rep != eth_port)
    {
        eth_pack= insert_vlan_hdr(eth_port, eth_pack);
        if (!eth_pack)
        {
            /* Packet is silently discarded */
            return NW_OK;
        }
    }

    eth_write_port(rep, eth_pack);
    return NW_OK;
}

PUBLIC int eth_read (fd, count)
int fd;
size_t count;
{
    eth_fd_t *eth_fd;
    acc_t *pack;

    eth_fd= &eth_fd_table[fd];
    if (!(eth_fd->ef_flags & EFF_OPTSET))
    {
        reply_thr_put(eth_fd, EBADMODE, FALSE);
        return NW_OK;
    }
    if (count < ETH_MAX_PACK_SIZE)
    {
        reply_thr_put(eth_fd, EPACKSIZE, FALSE);
        return NW_OK;
    }

    assert(!(eth_fd->ef_flags & EFF_READ_IP));
    eth_fd->ef_flags |= EFF_READ_IP;

    while (eth_fd->ef_rdbuf_head)
    {
        pack= eth_fd->ef_rdbuf_head;
        eth_fd->ef_rdbuf_head= pack->acc_ext_link;
        if (get_time() <= eth_fd->ef_exp_time)
        {
            packet2user(eth_fd, pack, eth_fd->ef_exp_time);
            if (!(eth_fd->ef_flags & EFF_READ_IP))
                return NW_OK;
        }
        else
            bf_afree(pack);
    }
    return NW_SUSPEND;
}

PUBLIC int eth_cancel(fd, which_operation)
int fd;
int which_operation;
{
    eth_fd_t *eth_fd, *prev, *loc_fd;
    eth_port_t *eth_port;

    DBLOCK(2, printf("eth_cancel(%d)\n", fd));
    eth_fd= &eth_fd_table[fd];

    switch (which_operation)
    {
    case SR_CANCEL_READ:
        assert (eth_fd->ef_flags & EFF_READ_IP);
        eth_fd->ef_flags &= ~EFF_READ_IP;
        reply_thr_put(eth_fd, EINTR, FALSE);
        break;
    case SR_CANCEL_WRITE:
        assert (eth_fd->ef_flags & EFF_WRITE_IP);
        eth_fd->ef_flags &= ~EFF_WRITE_IP;

```

```

    /* Remove fd from send queue */
    eth_port= eth_fd->ef_port;
    if (eth_port->etp_vlan_port)
        eth_port= eth_port->etp_vlan_port;
    for (prev= 0, loc_fd= eth_port->etp_sendq_head; loc_fd != NULL;
        prev= loc_fd, loc_fd= loc_fd->ef_send_next)
    {
        if (loc_fd == eth_fd)
            break;
    }
    assert(loc_fd == eth_fd);
    if (prev == NULL)
        eth_port->etp_sendq_head= loc_fd->ef_send_next;
    else
        prev->ef_send_next= loc_fd->ef_send_next;
    if (loc_fd->ef_send_next == NULL)
        eth_port->etp_sendq_tail= prev;

    reply_thr_get(eth_fd, EINTR, FALSE);
    break;
case SR_CANCEL_IOCTL:
    assert (eth_fd->ef_flags & EFF_IOCTL_IP);
    eth_fd->ef_flags &= ~EFF_IOCTL_IP;
    reply_thr_get(eth_fd, EINTR, TRUE);
    break;
default:
    ip_panic(( "got unknown cancel request" ));
}
return NW_OK;
}

PUBLIC int eth_select(fd, operations)
int fd;
unsigned operations;
{
    printf("eth_select: not implemented\n");
    return 0;
}

PUBLIC void eth_close(fd)
int fd;
{
    eth_fd_t *eth_fd;
    eth_port_t *eth_port;
    acc_t *pack;

    eth_fd= &eth_fd_table[fd];

    assert ((eth_fd->ef_flags & EFF_INUSE) &&
        !(eth_fd->ef_flags & EFF_BUSY));

    if (eth_fd->ef_flags & EFF_OPTSET)
        unhash_fd(eth_fd);
    while (eth_fd->ef_rdbuf_head != NULL)
    {
        pack= eth_fd->ef_rdbuf_head;
        eth_fd->ef_rdbuf_head= pack->acc_ext_link;
        bf_afree(pack);
    }
    eth_fd->ef_flags= EFF_EMPTY;

    eth_port= eth_fd->ef_port;
    do_rec_conf(eth_port);
}

PUBLIC void eth_loop_ev(ev, ev_arg)
event_t *ev;
ev_arg_t ev_arg;
{
    acc_t *pack;
    eth_port_t *eth_port;

    eth_port= ev_arg.ev_ptr;
    assert(ev == &eth_port->etp_sendev);

```

```
    pack= eth_port->etp_wr_pack;

    assert(!no_ethWritePort);
    no_ethWritePort= 1;
    eth_arrive(eth_port, pack, bf_bufsize(pack));
    assert(no_ethWritePort);
    no_ethWritePort= 0;

    eth_port->etp_wr_pack= NULL;
    eth_restart_write(eth_port);
}

PRIVATE int eth_checkopt (eth_fd)
eth_fd_t *eth_fd;
{
    /* bug: we don't check access modes yet */

    unsigned long flags;
    unsigned int en_di_flags;
    eth_port_t *eth_port;
    acc_t *pack;

    eth_port= eth_fd->ef_port;
    flags= eth_fd->ef_ethopt.nweo_flags;
    en_di_flags= (flags >>16) | (flags & 0xffff);

    if ((en_di_flags & NWEO_ACC_MASK) &&
        (en_di_flags & NWEO_LOC_MASK) &&
        (en_di_flags & NWEO_BROAD_MASK) &&
        (en_di_flags & NWEO_MULTI_MASK) &&
        (en_di_flags & NWEO_PROMISC_MASK) &&
        (en_di_flags & NWEO_REM_MASK) &&
        (en_di_flags & NWEO_TYPE_MASK) &&
        (en_di_flags & NWEO_RW_MASK))
    {
        eth_fd->ef_flags |= EFF_OPTSET;
    }
    else
        eth_fd->ef_flags &= ~EFF_OPTSET;

    while (eth_fd->ef_rdbuf_head != NULL)
    {
        pack= eth_fd->ef_rdbuf_head;
        eth_fd->ef_rdbuf_head= pack->acc_ext_link;
        bf_afree(pack);
    }

    return NW_OK;
}

PRIVATE void hash_fd(eth_fd)
eth_fd_t *eth_fd;
{
    eth_port_t *eth_port;
    int hash;

    eth_port= eth_fd->ef_port;
    if (eth_fd->ef_ethopt.nweo_flags & NWEO_TYPEANY)
    {
        eth_fd->ef_type_next= eth_port->etp_type_any;
        eth_port->etp_type_any= eth_fd;
    }
    else
    {
        hash= eth_fd->ef_ethopt.nweo_type;
        hash ^= (hash >> 8);
        hash &= (ETH_TYPE_HASH_NR-1);

        eth_fd->ef_type_next= eth_port->etp_type[hash];
        eth_port->etp_type[hash]= eth_fd;
    }
}
```

```

PRIVATE void unhash_fd(eth_fd)
eth_fd_t *eth_fd;
{
    eth_port_t *eth_port;
    eth_fd_t *prev, *curr, **eth_fd_p;
    int hash;

    eth_port= eth_fd->ef_port;
    if (eth_fd->ef_ethopt.nweo_flags & NWEO_TYPEANY)
    {
        eth_fd_p= &eth_port->etp_type_any;
    }
    else
    {
        hash= eth_fd->ef_ethopt.nweo_type;
        hash ^= (hash >> 8);
        hash &= (ETH_TYPE_HASH_NR-1);

        eth_fd_p= &eth_port->etp_type[hash];
    }
    for (prev= NULL, curr= *eth_fd_p; curr;
        prev= curr, curr= curr->ef_type_next)
    {
        if (curr == eth_fd)
            break;
    }
    assert(curr);
    if (prev)
        prev->ef_type_next= curr->ef_type_next;
    else
        *eth_fd_p= curr->ef_type_next;
}

PUBLIC void eth_restart_write(eth_port)
eth_port_t *eth_port;
{
    eth_fd_t *eth_fd;
    int r;

    assert(eth_port->etp_wr_pack == NULL);
    while (eth_fd= eth_port->etp_sendq_head, eth_fd != NULL)
    {
        if (eth_port->etp_wr_pack)
            return;
        eth_port->etp_sendq_head= eth_fd->ef_send_next;

        assert(eth_fd->ef_flags & EFF_WRITE_IP);
        eth_fd->ef_flags &= ~EFF_WRITE_IP;
        r= eth_write(eth_fd->eth_fd_table, eth_fd->ef_write_count);
        assert(r == NW_OK);
    }
}

PUBLIC void eth_arrive (eth_port, pack, pack_size)
eth_port_t *eth_port;
acc_t *pack;
size_t pack_size;
{
    eth_hdr_t *eth_hdr;
    ether_addr_t *dst_addr;
    int pack_stat;
    ether_type_t type;
    eth_fd_t *eth_fd, *first_fd, *share_fd;
    int hash, i;
    ul6_t vlan, temp;
    time_t exp_time;
    acc_t *vlan_pack, *hdr_acc, *tmp_acc;
    eth_port_t *vp;
    vlan_hdr_t vh;
    u32_t *p;

    exp_time= get_time() + EXPIRE_TIME;

```

```

pack= bf_packIfLess(pack, ETH_HDR_SIZE);

eth_hdr= (eth_hdr_t*)ptr2acc_data(pack);
dst_addr= &eth_hdr->eh_dst;

DIFBLOCK(0x20, dst_addr->ea_addr[0] != 0xFF &&
        (dst_addr->ea_addr[0] & 0x1),
        printf("got multicast packet\n"));

if (dst_addr->ea_addr[0] & 0x1)
{
    /* multi cast or broadcast */
    if (eth_addrncmp(*dst_addr, broadcast) == 0)
        pack_stat= NWE0_EN_BROAD;
    else
        pack_stat= NWE0_EN_MULTI;
}
else
{
    assert(eth_port->etp_flags & EPF_GOT_ADDR);
    if (eth_addrncmp(*dst_addr, eth_port->etp_ethaddr) == 0)
        pack_stat= NWE0_EN_LOC;
    else
        pack_stat= NWE0_EN_PROMISC;
}
type= eth_hdr->eh_proto;
hash= type;
hash ^= (hash >> 8);
hash &= (ETH_TYPE_HASH_NR-1);

if (type == HTONS(ETH_VLAN_PROTO))
{
    /* VLAN packet. Extract original ethernet packet */

    vlan_pack= pack;
    vlan_pack->acc_linkC++;
    hdr_acc= bf_cut(vlan_pack, 0, 2*sizeof(ether_addr_t));
    vlan_pack= bf_delhead(vlan_pack, 2*sizeof(ether_addr_t));
    vlan_pack= bf_packIfLess(vlan_pack, sizeof(vh));
    vh= *(vlan_hdr_t *)ptr2acc_data(vlan_pack);
    vlan_pack= bf_delhead(vlan_pack, sizeof(vh));
    hdr_acc= bf_append(hdr_acc, vlan_pack);
    vlan_pack= hdr_acc; hdr_acc= NULL;
    if (bf_bufsize(vlan_pack) < ETH_MIN_PACK_SIZE)
    {
        tmp_acc= bf_memreq(sizeof(vh));

        /* Clear padding */
        assert(sizeof(vh) <= sizeof(*p));
        p= (u32_t *)ptr2acc_data(tmp_acc);
        *p= 0xdeadbeef;

        vlan_pack= bf_append(vlan_pack, tmp_acc);
        tmp_acc= NULL;
    }
    vlan= ntohs(vh.vh_vlan);
    if (vlan & ETH_TCI_CFI)
    {
        /* No support for extended address formats */
        bf_afree(vlan_pack); vlan_pack= NULL;
    }
    vlan &= ETH_TCI_VLAN_MASK;
}
else
{
    /* No VLAN processing */
    vlan_pack= NULL;
    vlan= 0; /* lint */
}

first_fd= NULL;
for (i= 0; i<2; i++)
{
    share_fd= NULL;

```

```

eth_fd= (i == 0) ? eth_port->etp_type_any :
        eth_port->etp_type[hash];
for (; eth_fd; eth_fd= eth_fd->ef_type_next)
{
    if (i && eth_fd->ef_ethopt.nweo_type != type)
        continue;
    if (!(eth_fd->ef_ethopt.nweo_flags & pack_stat))
        continue;
    if (eth_fd->ef_ethopt.nweo_flags & NWEO_REMSPEC &&
        eth_addrncmp(eth_hdr->eh_src,
            eth_fd->ef_ethopt.nweo_rem) != 0)
    {
        continue;
    }
    if ((eth_fd->ef_ethopt.nweo_flags & NWEO_ACC_MASK) ==
        NWEO_SHARED)
    {
        if (!share_fd)
        {
            share_fd= eth_fd;
            continue;
        }
        if (!eth_fd->ef_rdbuf_head)
            share_fd= eth_fd;
        continue;
    }
    if (!first_fd)
    {
        first_fd= eth_fd;
        continue;
    }
    pack->acc_linkC++;
    packet2user(eth_fd, pack, exp_time);
}
if (share_fd)
{
    pack->acc_linkC++;
    packet2user(share_fd, pack, exp_time);
}
}
if (first_fd)
{
    if (first_fd->ef_put_pkt &&
        (first_fd->ef_flags & EFF_READ_IP) &&
        !(first_fd->ef_ethopt.nweo_flags & NWEO_RWDATONLY))
    {
        (*first_fd->ef_put_pkt)(first_fd->ef_srfd, pack,
            pack_size);
    }
    else
        packet2user(first_fd, pack, exp_time);
}
else
{
    if (pack_stat == NWEO_EN_LOC)
    {
        DBLOCK(0x01,
            printf("eth_arrive: dropping packet for proto 0x%x\n",
                ntohs(type)));
    }
    else
    {
        DBLOCK(0x20, printf("dropping packet for proto 0x%x\n",
            ntohs(type)));
    }
    bf_afree(pack);
}
if (vlan_pack)
{
    hash= ETH_HASH_VLAN(vlan, temp);
    for (vp= eth_port->etp_vlan_tab[hash]; vp;
        vp= vp->etp_vlan_next)
    {

```



```

        if (vp->etp_vlan == vlan)
            break;
    }
    if (vp)
    {
        eth_arrive(vp, vlan_pack, pack_size-sizeof(vh));
        vlan_pack= NULL;
    }
    else
    {
        /* No device for VLAN */
        bf_afree(vlan_pack);
        vlan_pack= NULL;
    }
}

PUBLIC void eth_reg_vlan(eth_port, vlan_port)
eth_port_t *eth_port;
eth_port_t *vlan_port;
{
    ul6_t t, vlan;
    int h;

    vlan= vlan_port->etp_vlan;
    h= ETH_HASH_VLAN(vlan, t);
    vlan_port->etp_vlan_next= eth_port->etp_vlan_tab[h];
    eth_port->etp_vlan_tab[h]= vlan_port;
}

PUBLIC void eth_restart_ioctl(eth_port)
eth_port_t *eth_port;
{
    int i;
    eth_fd_t *eth_fd;

    for (i= 0, eth_fd= eth_fd_table; i<ETH_FD_NR; i++, eth_fd++)
    {
        if (!(eth_fd->ef_flags & EFF_INUSE))
            continue;
        if (eth_fd->ef_port != eth_port)
            continue;
        if (!(eth_fd->ef_flags & EFF_IOCTL_IP))
            continue;
        if (eth_fd->ef_ioctl_req != NWIOGETHSTAT)
            continue;

        eth_fd->ef_flags &= ~EFF_IOCTL_IP;
        eth_ioctl(i, eth_fd->ef_ioctl_req);
    }
}

PRIVATE void packet2user (eth_fd, pack, exp_time)
eth_fd_t *eth_fd;
acc_t *pack;
time_t exp_time;
{
    int result;
    acc_t *tmp_pack;
    size_t size;

    assert (eth_fd->ef_flags & EFF_INUSE);
    if (!(eth_fd->ef_flags & EFF_READ_IP))
    {
        if (pack->acc_linkC != 1)
        {
            tmp_pack= bf_dupacc(pack);
            bf_afree(pack);
            pack= tmp_pack;
            tmp_pack= NULL;
        }
        pack->acc_ext_link= NULL;
        if (eth_fd->ef_rdbuf_head == NULL)
        {

```

```

        eth_fd->ef_rdbuf_head= pack;
        eth_fd->ef_exp_time= exp_time;
    }
    else
        eth_fd->ef_rdbuf_tail->acc_ext_link= pack;
    eth_fd->ef_rdbuf_tail= pack;
    return;
}

if (eth_fd->ef_ethopt.nweo_flags & NWEO_RWDATONLY)
    pack= bf_delhead(pack, ETH_HDR_SIZE);

size= bf_bufsize(pack);

if (eth_fd->ef_put_pkt)
{
    (*eth_fd->ef_put_pkt)(eth_fd->ef_srfd, pack, size);
    return;
}

eth_fd->ef_flags &= ~EFF_READ_IP;
result= (*eth_fd->ef_put_userdata)(eth_fd->ef_srfd, (size_t)0, pack,
    FALSE);
if (result >=0)
    reply_thr_put(eth_fd, size, FALSE);
else
    reply_thr_put(eth_fd, result, FALSE);
}

PRIVATE void eth_buffree (priority)
int priority;
{
    int i;
    eth_fd_t *eth_fd;
    acc_t *pack;

    if (priority == ETH_PRI_FDBUFS_EXTRA)
    {
        for (i= 0, eth_fd= eth_fd_table; i<ETH_FD_NR; i++, eth_fd++)
        {
            while (eth_fd->ef_rdbuf_head &&
                eth_fd->ef_rdbuf_head->acc_ext_link)
            {
                pack= eth_fd->ef_rdbuf_head;
                eth_fd->ef_rdbuf_head= pack->acc_ext_link;
                bf_afree(pack);
            }
        }
    }
    if (priority == ETH_PRI_FDBUFS)
    {
        for (i= 0, eth_fd= eth_fd_table; i<ETH_FD_NR; i++, eth_fd++)
        {
            while (eth_fd->ef_rdbuf_head)
            {
                pack= eth_fd->ef_rdbuf_head;
                eth_fd->ef_rdbuf_head= pack->acc_ext_link;
                bf_afree(pack);
            }
        }
    }
}

#ifdef BUF_CONSISTENCY_CHECK
PRIVATE void eth_bufcheck()
{
    int i;
    eth_fd_t *eth_fd;
    acc_t *pack;

    for (i= 0; i<eth_conf_nr; i++)
    {
        bf_check_acc(eth_port_table[i].etp_rd_pack);
        bf_check_acc(eth_port_table[i].etp_wr_pack);
    }
}

```

```

    }
    for (i= 0, eth_fd= eth_fd_table; i<ETH_FD_NR; i++, eth_fd++)
    {
        for (pack= eth_fd->ef_rdbuf_head; pack;
             pack= pack->acc_ext_link)
        {
            bf_check_acc(pack);
        }
    }
}
#endif

PRIVATE void do_rec_conf(eth_port)
eth_port_t *eth_port;
{
    int i;
    u32_t flags;
    eth_port_t *vp;

    if (eth_port->etp_vlan)
    {
        /* Configure underlying device */
        eth_port= eth_port->etp_vlan_port;
    }
    flags= compute_rec_conf(eth_port);
    for (i= 0; i<ETH_VLAN_HASH_NR; i++)
    {
        for (vp= eth_port->etp_vlan_tab[i]; vp; vp= vp->etp_vlan_next)
            flags |= compute_rec_conf(vp);
    }
    eth_set_rec_conf(eth_port, flags);
}

PRIVATE u32_t compute_rec_conf(eth_port)
eth_port_t *eth_port;
{
    eth_fd_t *eth_fd;
    u32_t flags;
    int i;

    flags= NWEQ_NOFLAGS;
    for (i=0, eth_fd= eth_fd_table; i<ETH_FD_NR; i++, eth_fd++)
    {
        if ((eth_fd->ef_flags & (EFF_INUSE|EFF_OPTSET)) !=
            (EFF_INUSE|EFF_OPTSET))
        {
            continue;
        }
        if (eth_fd->ef_port != eth_port)
            continue;
        flags |= eth_fd->ef_ethopt.nweo_flags;
    }
    return flags;
}

PRIVATE void reply_thr_get (eth_fd, result, for_ioctl)
eth_fd_t *eth_fd;
size_t result;
int for_ioctl;
{
    acc_t *data;

    data= (*eth_fd->ef_get_userdata)(eth_fd->ef_srfd, result, 0, for_ioctl);
    assert (!data);
}

PRIVATE void reply_thr_put (eth_fd, result, for_ioctl)
eth_fd_t *eth_fd;
size_t result;
int for_ioctl;
{
    int error;

    error= (*eth_fd->ef_put_userdata)(eth_fd->ef_srfd, result, (acc_t *)0,

```

```
        for_ioctl);
        assert(error == NW_OK);
}

PRIVATE acc_t *insert_vlan_hdr(eth_port, pack)
eth_port_t *eth_port;
acc_t *pack;
{
    acc_t *head_acc, *vh_acc;
    ul6_t type, vlan;
    vlan_hdr_t *vp;

    head_acc= bf_cut(pack, 0, 2*sizeof(ether_addr_t));
    pack= bf_delhead(pack, 2*sizeof(ether_addr_t));
    pack= bf_packIfLess(pack, sizeof(type));
    type= *(ul6_t *)ptr2acc_data(pack);
    if (type == HTONS(ETH_VLAN_PROTO))
    {
        /* Packeted is already tagged. Should update vlan number.
        * For now, just discard packet.
        */
        printf("insert_vlan_hdr: discarding vlan packet\n");
        bf_afree(head_acc); head_acc= NULL;
        bf_afree(pack); pack= NULL;
        return NULL;
    }
    vlan= eth_port->etp_vlan;        /* priority and CFI are zero */

    vh_acc= bf_memreq(sizeof(vlan_hdr_t));
    vp= (vlan_hdr_t *)ptr2acc_data(vh_acc);
    vp->vh_type= HTONS(ETH_VLAN_PROTO);
    vp->vh_vlan= htons(vlan);

    head_acc= bf_append(head_acc, vh_acc); vh_acc= NULL;
    head_acc= bf_append(head_acc, pack); pack= NULL;
    pack= head_acc; head_acc= NULL;
    return pack;
}

/*
 * $PchId: eth.c,v 1.23 2005/06/28 14:15:58 philip Exp $
 */
```

```
/*
eth.h

Copyright 1995 Philip Homburg
*/

#ifndef ETH_H
#define ETH_H

#define NWE0_DEFAULT      (NWE0_EN_LOC | NWE0_DI_BROAD | NWE0_DI_MULTI | \
    NWE0_DI_PROMISC | NWE0_REMANY | NWE0_RWDATALL)

#define eth_addrncmp(a,b) (memcmp((_VOIDSTAR)&a, (_VOIDSTAR)&b, \
    sizeof(a)))

/* Forward declatations */

struct acc;

/* prototypes */

void eth_prep ARGS(( void ));
void eth_init ARGS(( void ));
int eth_open ARGS(( int port, int srfd,
    get_userdata_t get_userdata, put_userdata_t put_userdata,
    put_pkt_t put_pkt, select_res_t sel_res ));
int eth_ioctl ARGS(( int fd, ioreq_t req));
int eth_read ARGS(( int port, size_t count ));
int eth_write ARGS(( int port, size_t count ));
int eth_cancel ARGS(( int fd, int which_operation ));
int eth_select ARGS(( int fd, unsigned operations ));
void eth_close ARGS(( int fd ));
int eth_send ARGS(( int port, struct acc *data, size_t data_len ));

#endif /* ETH_H */

/*
 * $PchId: eth.h,v 1.8 2005/06/28 14:16:10 philip Exp $
 */
```

```

/*
eth_int.h

Copyright 1995 Philip Homburg
*/

#ifndef ETH_INT_H
#define ETH_INT_H

#define ETH_TYPE_HASH_NR      16
#define ETH_VLAN_HASH_NR     16

/* Assume that the arguments are a local variable */
#define ETH_HASH_VLAN(v,t) \
    ((t)= (((v) >> 8) ^ (v)), \
    (t)= (((t) >> 4) ^ (t)), \
    (t) & (ETH_VLAN_HASH_NR-1))

typedef struct eth_port
{
    int etp_flags;
    ether_addr_t etp_ethaddr;
    acc_t *etp_wr_pack, *etp_rd_pack;
    struct eth_fd *etp_sendq_head;
    struct eth_fd *etp_sendq_tail;
    struct eth_fd *etp_type_any;
    struct eth_fd *etp_type[ETH_TYPE_HASH_NR];
    event_t etp_sendev;

    /* VLAN support */
    ul6_t etp_vlan;
    struct eth_port *etp_vlan_port;
    struct eth_port *etp_vlan_tab[ETH_VLAN_HASH_NR];
    struct eth_port *etp_vlan_next;

    osdep_eth_port_t etp_osdep;
} eth_port_t;

#define EPF_EMPTY      0x0
#define EPF_ENABLED    0x1
#define EPF_GOT_ADDR   0x2    /* Got ethernet address from device */
#define EPF_READ_IP    0x20
#define EPF_READ_SP    0x40

extern eth_port_t *eth_port_table;

extern int no_ethWritePort;    /* debug, consistency check */

void osdep_eth_init ARGS(( void ));
int eth_get_stat ARGS(( eth_port_t *eth_port, eth_stat_t *eth_stat ));
void eth_write_port ARGS(( eth_port_t *eth_port, acc_t *pack ));
void eth_arrive ARGS(( eth_port_t *port, acc_t *pack, size_t pack_size ));
void eth_set_rec_conf ARGS(( eth_port_t *eth_port, u32_t flags ));
void eth_restart_write ARGS(( eth_port_t *eth_port ));
void eth_loop_ev ARGS(( event_t *ev, ev_arg_t ev_arg ));
void eth_reg_vlan ARGS(( eth_port_t *eth_port, eth_port_t *vlan_port ));
void eth_restart_ioctl ARGS(( eth_port_t *eth_port ));

#endif /* ETH_INT_H */

/*
 * $PchId: eth_int.h,v 1.9 2001/04/23 08:04:06 philip Exp $
 */

```

```
/*
inet/generic/event.c

Created:      April 1995 by Philip Homburg <philip@f-mnx.phicoh.com>

Implementation of an event queue.

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "assert.h"
#include "event.h"

THIS_FILE

event_t *ev_head;
static event_t *ev_tail;

void ev_init(ev)
event_t *ev;
{
    ev->ev_func= 0;
    ev->ev_next= NULL;
}

void ev_enqueue(ev, func, ev_arg)
event_t *ev;
ev_func_t func;
ev_arg_t ev_arg;
{
    assert(ev->ev_func == 0);
    ev->ev_func= func;
    ev->ev_arg= ev_arg;
    ev->ev_next= NULL;
    if (ev_head == NULL)
        ev_head= ev;
    else
        ev_tail->ev_next= ev;
    ev_tail= ev;
}

void ev_process()
{
    ev_func_t func;
    event_t *curr;

    while (ev_head)
    {
        curr= ev_head;
        ev_head= curr->ev_next;
        func= curr->ev_func;
        curr->ev_func= 0;

        assert(func != 0);
        func(curr, curr->ev_arg);
    }
}

int ev_in_queue(ev)
event_t *ev;
{
    return ev->ev_func != 0;
}

/*
 * $PchId: event.c,v 1.6 2004/08/03 16:23:32 philip Exp $
 */
```

```
/*
inet/generic/event.h

Created:      April 1995 by Philip Homburg <philip@f-mnx.phicoh.com>

Header file for an event mechanism.

Copyright 1995 Philip Homburg
*/

#ifndef INET__GENERIC__EVENT_H
#define INET__GENERIC__EVENT_H

struct event;

typedef union ev_arg
{
    int ev_int;
    void *ev_ptr;
} ev_arg_t;

typedef void (*ev_func_t) ARGS(( struct event *ev, union ev_arg eva ));

typedef struct event
{
    ev_func_t ev_func;
    ev_arg_t ev_arg;
    struct event *ev_next;
} event_t;

extern event_t *ev_head;

void ev_init ARGS(( event_t *ev ));
void ev_enqueue ARGS(( event_t *ev, ev_func_t func, ev_arg_t ev_arg ));
void ev_process ARGS(( void ));
int ev_in_queue ARGS(( event_t *ev ));

#endif /* INET__GENERIC__EVENT_H */

/*
 * $PchId: event.h,v 1.5 2004/08/03 16:23:49 philip Exp $
 */
```



```
/*
icmp.c

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "buf.h"
#include "event.h"
#include "type.h"

#include "assert.h"
#include "clock.h"
#include "icmp.h"
#include "icmp_lib.h"
#include "io.h"
#include "ip.h"
#include "ip_int.h"
#include "ipr.h"

THIS_FILE

typedef struct icmp_port
{
    int icp_flags;
    int icp_state;
    int icp_ipport;
    int icp_ipfd;
    unsigned icp_rate_count;
    unsigned icp_rate_report;
    time_t icp_rate_lasttime;
    acc_t *icp_head_queue;
    acc_t *icp_tail_queue;
    acc_t *icp_write_pack;
    event_t icp_event;
} icmp_port_t;

#define ICPF_EMPTY      0x0
#define ICPF_SUSPEND    0x1
#define ICPF_READ_IP    0x2
#define ICPF_READ_SP    0x4
#define ICPF_WRITE_IP   0x8
#define ICPF_WRITE_SP   0x10

#define ICPS_BEGIN      0
#define ICPS_IPOPT      1
#define ICPS_MAIN        2
#define ICPS_ERROR      3

PRIVATE icmp_port_t *icmp_port_table;

FORWARD void icmp_main ARGS(( icmp_port_t *icmp_port ));
FORWARD acc_t *icmp_getdata ARGS(( int port, size_t offset,
size_t count, int for_ioctl ));
FORWARD int icmp_putdata ARGS(( int port, size_t offset,
acc_t *data, int for_ioctl ));
FORWARD void icmp_read ARGS(( icmp_port_t *icmp_port ));
FORWARD void process_data ARGS(( icmp_port_t *icmp_port,
acc_t *data ));
FORWARD ul6_t icmp_pack_oneCsum ARGS(( acc_t *ip_pack ));
FORWARD void icmp_echo_request ARGS(( icmp_port_t *icmp_port,
acc_t *ip_pack, int ip_hdr_len, ip_hdr_t *ip_hdr,
acc_t *icmp_pack, int icmp_len, icmp_hdr_t *icmp_hdr ));
FORWARD void icmp_dst_unreach ARGS(( icmp_port_t *icmp_port,
acc_t *ip_pack, int ip_hdr_len, ip_hdr_t *ip_hdr,
acc_t *icmp_pack, int icmp_len, icmp_hdr_t *icmp_hdr ));
FORWARD void icmp_time_exceeded ARGS(( icmp_port_t *icmp_port,
acc_t *ip_pack, int ip_hdr_len, ip_hdr_t *ip_hdr,
acc_t *icmp_pack, int icmp_len, icmp_hdr_t *icmp_hdr ));
FORWARD void icmp_router_advertisement ARGS(( icmp_port_t *icmp_port,
acc_t *icmp_pack, int icmp_len, icmp_hdr_t *icmp_hdr ));
FORWARD void icmp_redirect ARGS(( icmp_port_t *icmp_port,
ip_hdr_t *ip_hdr, acc_t *icmp_pack, int icmp_len,
icmp_hdr_t *icmp_hdr ));
```

```

FORWARD acc_t *make_repl_ip ARGS(( ip_hdr_t *ip_hdr,
    int ip_len ));
FORWARD void enqueue_pack ARGS(( icmp_port_t *icmp_port,
    acc_t *reply_ip_hdr ));
FORWARD int icmp_rate_limit ARGS(( icmp_port_t *icmp_port,
    acc_t *reply_ip_hdr ));
FORWARD void icmp_write ARGS(( event_t *ev, ev_arg_t ev_arg ));
FORWARD void icmp_buffree ARGS(( int priority ));
FORWARD acc_t *icmp_err_pack ARGS(( acc_t *pack, icmp_hdr_t **icmp_hdr_pp ));
#ifdef BUF_CONSISTENCY_CHECK
FORWARD void icmp_bufcheck ARGS(( void ));
#endif

PUBLIC void icmp_prep()
{
    icmp_port_table= alloc(ip_conf_nr * sizeof(icmp_port_table[0]));
}

PUBLIC void icmp_init()
{
    int i;
    icmp_port_t *icmp_port;

    assert (BUF_S >= sizeof (nwio_ipopt_t));

    for (i= 0, icmp_port= icmp_port_table; i<ip_conf_nr; i++, icmp_port++)
    {
        icmp_port->icp_flags= ICPF_EMPTY;
        icmp_port->icp_state= ICPS_BEGIN;
        icmp_port->icp_ipport= i;
        icmp_port->icp_rate_count= 0;
        icmp_port->icp_rate_report= ICMP_MAX_RATE;
        icmp_port->icp_rate_lasttime= 0;
        ev_init(&icmp_port->icp_event);
    }

#ifdef BUF_CONSISTENCY_CHECK
    bf_logon(icmp_buffree);
#else
    bf_logon(icmp_buffree, icmp_bufcheck);
#endif

    for (i= 0, icmp_port= icmp_port_table; i<ip_conf_nr; i++, icmp_port++)
    {
        icmp_main (icmp_port);
    }
}

PRIVATE void icmp_main(icmp_port)
icmp_port_t *icmp_port;
{
    int result;
    switch (icmp_port->icp_state)
    {
    case ICPS_BEGIN:
        icmp_port->icp_head_queue= 0;
        icmp_port->icp_ipfd= ip_open(icmp_port->icp_ipport,
            icmp_port->icp_ipport, icmp_getdata, icmp_putdata,
            0 /* no put_pkt */, 0 /* no select_res */);
        if (icmp_port->icp_ipfd<0)
        {
            DBLOCK(1, printf("unable to open ip_port %d\n",
                icmp_port->icp_ipport));
            break;
        }
        icmp_port->icp_state= ICPS_IPOPT;
        icmp_port->icp_flags &= ~ICPF_SUSPEND;
        result= ip_ioctl (icmp_port->icp_ipfd, NWIOSIPOPT);
        if (result == NW_SUSPEND)
        {
            icmp_port->icp_flags |= ICPF_SUSPEND;
            break;
        }
        assert(result == NW_OK);
    }
}

```

```

        /* falls through */
    case ICPS_IPOPT:
        icmp_port->icp_state= ICPS_MAIN;
        icmp_port->icp_flags &= ~ICPF_SUSPEND;
        icmp_read(icmp_port);
        break;
    default:
        DBLOCK(1, printf("unknown state %d\n",
            icmp_port->icp_state));
        break;
}
}

PRIVATE acc_t *icmp_getdata(port, offset, count, for_ioctl)
int port;
size_t offset, count;
int for_ioctl;
{
    icmp_port_t *icmp_port;
    nwio_ipopt_t *ipopt;
    acc_t *data;
    int result;
    ev_arg_t ev_arg;

    icmp_port= &icmp_port_table[port];

    if (icmp_port->icp_flags & ICPF_WRITE_IP)
    {
        if (!count)
        {
            bf_afree(icmp_port->icp_write_pack);
            icmp_port->icp_write_pack= 0;

            result= (int)offset;
            if (result<0)
            {
                DBLOCK(1, printf("got write error %d\n",
                    result));
            }
            if (icmp_port->icp_flags & ICPF_WRITE_SP)
            {
                icmp_port->icp_flags &= ~ICPF_WRITE_SP;
                ev_arg.ev_ptr= icmp_port;
                ev_enqueue(&icmp_port->icp_event, icmp_write,
                    ev_arg);
            }
            return NW_OK;
        }
        return bf_cut(icmp_port->icp_write_pack, offset, count);
    }
    switch (icmp_port->icp_state)
    {
    case ICPS_IPOPT:
        if (!count)
        {
            result= (int)offset;
            assert(result == NW_OK);
            if (result < 0)
            {
                icmp_port->icp_state= ICPS_ERROR;
                break;
            }
            if (icmp_port->icp_flags & ICPF_SUSPEND)
                icmp_main(icmp_port);
            return NW_OK;
        }

        data= bf_memreq (sizeof (*ipopt));
        ipopt= (nwio_ipopt_t *)ptr2acc_data(data);
        ipopt->nwio_flags= NWIO_COPY | NWIO_EN_LOC |
            NWIO_EN_BROAD |
            NWIO_REMANY | NWIO_PROTOSPEC |
            NWIO_HDR_O_ANY | NWIO_RWDATALL;

```

```
        ipopt->nwio_proto= IPPROTO_ICMP;
        return data;
    default:
        break;
    }
    DBLOCK(1, printf("unknown state %d\n", icmp_port->icp_state));
    return NULL;
}

PRIVATE int icmp_putdata(port, offset, data, for_ioctl)
int port;
size_t offset;
acc_t *data;
int for_ioctl;
{
    icmp_port_t *icmp_port;
    int result;

    icmp_port= &icmp_port_table[port];

    if (icmp_port->icp_flags & ICPF_READ_IP)
    {
        if (!data)
        {
            result= (int)offset;
            if (result<0)
            {
                DBLOCK(1, printf("got read error %d\n",
                                result));
            }
            if (icmp_port->icp_flags & ICPF_READ_SP)
            {
                icmp_port->icp_flags &=
                    ~(ICPF_READ_IP|ICPF_READ_SP);
                icmp_read (icmp_port);
            }
            return NW_OK;
        }
        process_data(icmp_port, data);
        return NW_OK;
    }
    switch (icmp_port->icp_state)
    {
    default:
        DBLOCK(1, printf("unknown state %d\n",
                        icmp_port->icp_state));
        return 0;
    }
}

PRIVATE void icmp_read(icmp_port)
icmp_port_t *icmp_port;
{
    int result;

    for (;;)
    {
        icmp_port->icp_flags |= ICPF_READ_IP;
        icmp_port->icp_flags &= ~ICPF_READ_SP;

        result= ip_read(icmp_port->icp_ipfd, ICMP_MAX_DATAGRAM);
        if (result == NW_SUSPEND)
        {
            icmp_port->icp_flags |= ICPF_READ_SP;
            return;
        }
    }
}

PUBLIC void icmp_snd_time_exceeded(port_nr, pack, code)
int port_nr;
acc_t *pack;
int code;
{
```

```
    icmp_hdr_t *icmp_hdr;
    icmp_port_t *icmp_port;

    if (port_nr >= 0 && port_nr < ip_conf_nr)
        icmp_port = &icmp_port_table[port_nr];
    else
    {
        printf("icmp_snd_time_exceeded: strange port %d\n", port_nr);
        bf_afree(pack);
        return;
    }
    pack = icmp_err_pack(pack, &icmp_hdr);
    if (pack == NULL)
        return;
    icmp_hdr->ih_type = ICMP_TYPE_TIME_EXCEEDED;
    icmp_hdr->ih_code = code;
    icmp_hdr->ih_chksum = ~oneC_sum(~icmp_hdr->ih_chksum,
        (ul6_t *)&icmp_hdr->ih_type, 2);
    enqueue_pack(icmp_port, pack);
}
```

```
PUBLIC void icmp_snd_redirect(port_nr, pack, code, gw)
int port_nr;
acc_t *pack;
int code;
ipaddr_t gw;
{
    icmp_hdr_t *icmp_hdr;
    icmp_port_t *icmp_port;

    if (port_nr >= 0 && port_nr < ip_conf_nr)
        icmp_port = &icmp_port_table[port_nr];
    else
    {
        printf("icmp_snd_redirect: strange port %d\n", port_nr);
        bf_afree(pack);
        return;
    }
    pack = icmp_err_pack(pack, &icmp_hdr);
    if (pack == NULL)
        return;
    icmp_hdr->ih_type = ICMP_TYPE_REDIRECT;
    icmp_hdr->ih_code = code;
    icmp_hdr->ih_hun.ihh_gateway = gw;
    icmp_hdr->ih_chksum = ~oneC_sum(~icmp_hdr->ih_chksum,
        (ul6_t *)&icmp_hdr->ih_type, 2);
    icmp_hdr->ih_chksum = ~oneC_sum(~icmp_hdr->ih_chksum,
        (ul6_t *)&icmp_hdr->ih_hun.ihh_gateway, 4);
    enqueue_pack(icmp_port, pack);
}
```

```
PUBLIC void icmp_snd_unreachable(port_nr, pack, code)
int port_nr;
acc_t *pack;
int code;
{
    icmp_hdr_t *icmp_hdr;
    icmp_port_t *icmp_port;

    if (port_nr >= 0 && port_nr < ip_conf_nr)
        icmp_port = &icmp_port_table[port_nr];
    else
    {
        printf("icmp_snd_unreachable: strange port %d\n", port_nr);
        bf_afree(pack);
        return;
    }
    pack = icmp_err_pack(pack, &icmp_hdr);
    if (pack == NULL)
        return;
    icmp_hdr->ih_type = ICMP_TYPE_DST_UNRCH;
    icmp_hdr->ih_code = code;
    icmp_hdr->ih_chksum = ~oneC_sum(~icmp_hdr->ih_chksum,
        (ul6_t *)&icmp_hdr->ih_type, 2);
}
```

```

        enqueue_pack(icmp_port, pack);
    }

PUBLIC void icmp_snd_mtu(port_nr, pack, mtu)
int port_nr;
acc_t *pack;
ul6_t mtu;
{
    icmp_hdr_t *icmp_hdr;
    icmp_port_t *icmp_port;

    if (port_nr >= 0 && port_nr < ip_conf_nr)
        icmp_port= &icmp_port_table[port_nr];
    else
    {
        printf("icmp_snd_mtu: strange port %d\n", port_nr);
        bf_afree(pack);
        return;
    }

    pack= icmp_err_pack(pack, &icmp_hdr);
    if (pack == NULL)
        return;
    icmp_hdr->ih_type= ICMP_TYPE_DST_UNRCH;
    icmp_hdr->ih_code= ICMP_FRAGM_AND_DF;
    icmp_hdr->ih_hun.ihh_mtu.im_mtu= htons(mtu);
    icmp_hdr->ih_chksum= ~oneC_sum(~icmp_hdr->ih_chksum,
        (ul6_t *)&icmp_hdr->ih_type, 2);
    icmp_hdr->ih_chksum= ~oneC_sum(~icmp_hdr->ih_chksum,
        (ul6_t *)&icmp_hdr->ih_hun.ihh_mtu.im_mtu, 2);
    enqueue_pack(icmp_port, pack);
}

PRIVATE void process_data(icmp_port, data)
icmp_port_t *icmp_port;
acc_t *data;
{
    ip_hdr_t *ip_hdr;
    icmp_hdr_t *icmp_hdr;
    acc_t *icmp_data;
    int ip_hdr_len;
    size_t pack_len;

    /* Align entire packet */
    data= bf_align(data, BUF_S, 4);

    data= bf_packIfLess(data, IP_MIN_HDR_SIZE);
    ip_hdr= (ip_hdr_t *)ptr2acc_data(data);
    DIFBLOCK(0x10, (ip_hdr->ih_dst & HTONL(0xf0000000)) == HTONL(0xe0000000),
        printf("got multicast packet\n"));
    ip_hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;

    if (ip_hdr_len > IP_MIN_HDR_SIZE)
    {
        data= bf_packIfLess(data, ip_hdr_len);
        ip_hdr= (ip_hdr_t *)ptr2acc_data(data);
    }

    pack_len= bf_bufsize(data);
    pack_len -= ip_hdr_len;
    if (pack_len < ICMP_MIN_HDR_SIZE)
    {
        if (pack_len == 0 && ip_hdr->ih_proto == 0)
        {
            /* IP layer reports new ip address, which can be
             * ignored.
             */
        }
        else
            DBLOCK(1, printf("got an incomplete icmp packet\n"));
        bf_afree(data);
        return;
    }
}

```

```

icmp_data= bf_cut(data, ip_hdr_len, pack_len);

icmp_data= bf_packIfLess (icmp_data, ICMP_MIN_HDR_SIZE);
icmp_hdr= (icmp_hdr_t *)ptr2acc_data(icmp_data);

if ((ul6_t)~icmp_pack_oneCsum(icmp_data))
{
    DBLOCK(1, printf(
        "got packet with bad checksum (= 0x%x, 0x%x)\n",
        icmp_hdr->ih_chksum,
        (ul6_t)~icmp_pack_oneCsum(icmp_data)));
    bf_afree(data);
    bf_afree(icmp_data);
    return;
}

switch (icmp_hdr->ih_type)
{
case ICMP_TYPE_ECHO_REPL:
    break;
case ICMP_TYPE_DST_UNRCH:
    icmp_dst_unreach (icmp_port, data, ip_hdr_len, ip_hdr,
        icmp_data, pack_len, icmp_hdr);
    break;
case ICMP_TYPE_SRC_QUENCH:
    /* Ignore src quench ICMPs */
    DBLOCK(2, printf("ignoring SRC QUENCH ICMP.\n"));
    break;
case ICMP_TYPE_REDIRECT:
    icmp_redirect (icmp_port, ip_hdr, icmp_data, pack_len,
        icmp_hdr);
    break;
case ICMP_TYPE_ECHO_REQ:
    icmp_echo_request(icmp_port, data, ip_hdr_len, ip_hdr,
        icmp_data, pack_len, icmp_hdr);
    return;
case ICMP_TYPE_ROUTER_ADVER:
    icmp_router_advertisement(icmp_port, icmp_data, pack_len,
        icmp_hdr);
    break;
case ICMP_TYPE_ROUTE_SOL:
    break; /* Should be handled by a routing deamon. */
case ICMP_TYPE_TIME_EXCEEDED:
    icmp_time_exceeded (icmp_port, data, ip_hdr_len, ip_hdr,
        icmp_data, pack_len, icmp_hdr);
    break;
default:
    DBLOCK(1, printf("got an unknown icmp (%d) from ",
        icmp_hdr->ih_type);
        writeIpAddr(ip_hdr->ih_src); printf("\n"));
    break;
}
bf_afree(data);
bf_afree(icmp_data);
}

PRIVATE void icmp_echo_request(icmp_port, ip_data, ip_len, ip_hdr,
    icmp_data, icmp_len, icmp_hdr)
icmp_port_t *icmp_port;
acc_t *ip_data, *icmp_data;
int ip_len, icmp_len;
ip_hdr_t *ip_hdr;
icmp_hdr_t *icmp_hdr;
{
    acc_t *repl_ip_hdr, *repl_icmp;
    ipaddr_t tmpaddr, locaddr, netmask;
    icmp_hdr_t *repl_icmp_hdr;
    i32_t tmp_chksum;
    ip_port_t *ip_port;

    if (icmp_hdr->ih_code != 0)
    {
        DBLOCK(1,
            printf("got an icmp echo request with unknown code (%d)\n",

```

```

        icmp_hdr->ih_code));
    bf_afree(ip_data);
    bf_afree(icmp_data);
    return;
}
if (icmp_len < ICMP_MIN_HDR_SIZE + sizeof(icmp_id_seq_t))
{
    DBLOCK(1, printf("got an incomplete icmp echo request\n"));
    bf_afree(ip_data);
    bf_afree(icmp_data);
    return;
}
tmpaddr= ntohl(ip_hdr->ih_dst);
if ((tmpaddr & 0xe0000000) == 0xe0000000 &&
    tmpaddr != 0xffffffff)
{
    /* Respond only to the all hosts multicast address until
     * a decent listening service has been implemented
     */
    if (tmpaddr != 0xe0000001)
    {
        bf_afree(ip_data);
        bf_afree(icmp_data);
        return;
    }
}

/* Limit subnet broadcasts to the local net */
ip_port= &ip_port_table[icmp_port->icp_ipport];
locaddr= ip_port->ip_ipaddr;
netmask= ip_port->ip_subnetmask;
if (ip_hdr->ih_dst == (locaddr | ~netmask) &&
    (ip_port->ip_flags & IPF_SUBNET_BCAST) &&
    ((ip_hdr->ih_src ^ locaddr) & netmask) != 0)
{
    /* Directed broadcast */
    bf_afree(ip_data);
    bf_afree(icmp_data);
    return;
}

repl_ip_hdr= make_repl_ip(ip_hdr, ip_len);
repl_icmp= bf_memreq (ICMP_MIN_HDR_SIZE);
repl_icmp_hdr= (icmp_hdr_t *)ptr2acc_data(repl_icmp);
repl_icmp_hdr->ih_type= ICMP_TYPE_ECHO_REPL;
repl_icmp_hdr->ih_code= 0;

DBLOCK(2,
printf("ih_chksum= 0x%x, ih_type= 0x%x, repl->ih_type= 0x%x\n",
        icmp_hdr->ih_chksum, *(u16_t *)&icmp_hdr->ih_type,
        *(u16_t *)&repl_icmp_hdr->ih_type));
tmp_chksum= (~icmp_hdr->ih_chksum & 0xffff) -
    (i32_t)*(u16_t *)&icmp_hdr->ih_type+
    *(u16_t *)&repl_icmp_hdr->ih_type;
tmp_chksum= (tmp_chksum >> 16) + (tmp_chksum & 0xffff);
tmp_chksum= (tmp_chksum >> 16) + (tmp_chksum & 0xffff);
repl_icmp_hdr->ih_chksum= ~tmp_chksum;
DBLOCK(2, printf("sending chksum 0x%x\n", repl_icmp_hdr->ih_chksum));

repl_ip_hdr->acc_next= repl_icmp;
repl_icmp->acc_next= bf_cut (icmp_data, ICMP_MIN_HDR_SIZE,
    icmp_len - ICMP_MIN_HDR_SIZE);

bf_afree(ip_data);
bf_afree(icmp_data);

enqueue_pack(icmp_port, repl_ip_hdr);
}

PRIVATE u16_t icmp_pack_oneCsum(icmp_pack)
acc_t *icmp_pack;
{
    u16_t prev;
    int odd_byte;

```



```

    char *data_ptr;
    int length;
    char byte_buf[2];

    prev= 0;

    odd_byte= FALSE;
    for (; icmp_pack; icmp_pack= icmp_pack->acc_next)
    {
        data_ptr= ptr2acc_data(icmp_pack);
        length= icmp_pack->acc_length;

        if (!length)
            continue;
        if (odd_byte)
        {
            byte_buf[1]= *data_ptr;
            prev= oneC_sum(prev, (ul6_t *)byte_buf, 2);
            data_ptr++;
            length--;
            odd_byte= FALSE;
        }
        if (length & 1)
        {
            odd_byte= TRUE;
            length--;
            byte_buf[0]= data_ptr[length];
        }
        if (!length)
            continue;
        prev= oneC_sum (prev, (ul6_t *)data_ptr, length);
    }
    if (odd_byte)
        prev= oneC_sum (prev, (ul6_t *)byte_buf, 1);
    return prev;
}

PRIVATE acc_t *make_repl_ip(ip_hdr, ip_len)
ip_hdr_t *ip_hdr;
int ip_len;
{
    ip_hdr_t *repl_ip_hdr;
    acc_t *repl;
    int repl_hdr_len;

    if (ip_len>IP_MIN_HDR_SIZE)
    {
        DBLOCK(1, printf("ip_hdr options NOT supported (yet?)\n" ));
        ip_len= IP_MIN_HDR_SIZE;
    }

    repl_hdr_len= IP_MIN_HDR_SIZE;

    repl= bf_memreq(repl_hdr_len);

    repl_ip_hdr= (ip_hdr_t *)ptr2acc_data(repl);

    repl_ip_hdr->ih_vers_ihl= repl_hdr_len >> 2;
    repl_ip_hdr->ih_tos= ip_hdr->ih_tos;
    repl_ip_hdr->ih_ttl= ICMP_DEF_TTL;
    repl_ip_hdr->ih_proto= IPPROTO_ICMP;
    repl_ip_hdr->ih_dst= ip_hdr->ih_src;
    repl_ip_hdr->ih_flags_fragoff= 0;

    return repl;
}

PRIVATE void enqueue_pack(icmp_port, reply_ip_hdr)
icmp_port_t *icmp_port;
acc_t *reply_ip_hdr;
{
    int r;
    ev_arg_t ev_arg;

```

```

/* Check rate */
if (icmp_port->icp_rate_count >= ICMP_MAX_RATE)
{
    /* Something is going wrong; check policy */
    r= icmp_rate_limit(icmp_port, reply_ip_hdr);
    if (r == -1)
    {
        bf_afree(reply_ip_hdr);
        reply_ip_hdr= NULL;
        return;
    }

    /* OK, continue */
}
icmp_port->icp_rate_count++;

reply_ip_hdr->acc_ext_link= 0;

if (icmp_port->icp_head_queue)
{
    icmp_port->icp_tail_queue->acc_ext_link=
        reply_ip_hdr;
}
else
{
    icmp_port->icp_head_queue= reply_ip_hdr;
}
reply_ip_hdr->acc_ext_link= NULL;
icmp_port->icp_tail_queue= reply_ip_hdr;

if (!(icmp_port->icp_flags & ICPF_WRITE_IP))
{
    icmp_port->icp_flags |= ICPF_WRITE_IP;
    ev_arg.ev_ptr= icmp_port;
    ev_enqueue(&icmp_port->icp_event, icmp_write, ev_arg);
}
}

```

```

PRIVATE int icmp_rate_limit(icmp_port, reply_ip_hdr)
icmp_port_t *icmp_port;
acc_t *reply_ip_hdr;
{
    time_t t;
    acc_t *pack;
    ip_hdr_t *ip_hdr;
    icmp_hdr_t *icmp_hdr;
    int hdrlen, icmp_hdr_len, type;

    /* Check the time first */
    t= get_time();
    if (t >= icmp_port->icp_rate_lasttime + ICMP_RATE_INTERVAL)
    {
        icmp_port->icp_rate_lasttime= t;
        icmp_port->icp_rate_count= 0;
        return 0;
    }

    icmp_port->icp_rate_count++;

    /* Adjust report limit if necessary */
    if (icmp_port->icp_rate_count >
        icmp_port->icp_rate_report+ICMP_RATE_WARN)
    {
        icmp_port->icp_rate_report *= 2;
        return -1;
    }

    /* Do we need to report */
    if (icmp_port->icp_rate_count < icmp_port->icp_rate_report)
        return -1;

    pack= bf_dupacc(reply_ip_hdr);
    pack= bf_packIfLess(pack, IP_MIN_HDR_SIZE);
    ip_hdr= (ip_hdr_t *)ptr2acc_data(pack);

```

```

printf("icmp[%d]: dropping ICMP packet #%%d to ",
       icmp_port->icp_ipport, icmp_port->icp_rate_count);
writeIpAddr(ip_hdr->ih_dst);
hdrlen= (ip_hdr->ih_vers_ihl & IH_IHL_MASK)*4;
pack= bf_packIfLess(pack, hdrlen+ICMP_MIN_HDR_SIZE);
ip_hdr= (ip_hdr_t *)ptr2acc_data(pack);
icmp_hdr= (icmp_hdr_t *) (ptr2acc_data(pack)+hdrlen);
type= icmp_hdr->ih_type;
printf(" type %%d,code %%d\\n", type, icmp_hdr->ih_code);
switch(type)
{
case ICMP_TYPE_DST_UNRCH:
case ICMP_TYPE_SRC_QUENCH:
case ICMP_TYPE_REDIRECT:
case ICMP_TYPE_TIME_EXCEEDED:
case ICMP_TYPE_PARAM_PROBLEM:
    icmp_hdr_len= offsetof(struct icmp_hdr, ih_dun);
    pack= bf_packIfLess(pack,
        hdrlen+icmp_hdr_len+IP_MIN_HDR_SIZE);
    ip_hdr= (ip_hdr_t *) (ptr2acc_data(pack)+hdrlen+icmp_hdr_len);
    icmp_hdr= (icmp_hdr_t *) (ptr2acc_data(pack)+hdrlen);
    printf("\\tinfo %%08x, original dst ",
        ntohs(icmp_hdr->ih_hun.ihh_unused));
    writeIpAddr(ip_hdr->ih_dst);
    printf(", proto %%d, length %%u\\n",
        ip_hdr->ih_proto, ntohs(ip_hdr->ih_length));
    break;
default:
    break;
}
bf_afree(pack); pack= NULL;

return -1;
}

PRIVATE void icmp_write(ev, ev_arg)
event_t *ev;
ev_arg_t ev_arg;
{
    int result;
    icmp_port_t *icmp_port;
    acc_t *data;

    icmp_port= ev_arg.ev_ptr;
    assert(ev == &icmp_port->icp_event);

    assert(icmp_port->icp_flags & ICPF_WRITE_IP);
    assert(!(icmp_port->icp_flags & ICPF_WRITE_SP));

    while(icmp_port->icp_head_queue != NULL)
    {
        data= icmp_port->icp_head_queue;
        icmp_port->icp_head_queue= data->acc_ext_link;

        result= ip_send(icmp_port->icp_ipfd, data,
            bf_bufsize(data));
        if (result != NW_WOULDBLOCK)
        {
            if (result == NW_OK)
                continue;
            DBLOCK(1, printf("icmp_write: error %%d\\n", result));
            continue;
        }

        assert(icmp_port->icp_write_pack == NULL);
        icmp_port->icp_write_pack= data;

        result= ip_write(icmp_port->icp_ipfd,
            bf_bufsize(icmp_port->icp_write_pack));
        if (result == NW_SUSPEND)
        {
            icmp_port->icp_flags |= ICPF_WRITE_SP;
            return;
        }
    }
}

```

```

    }
    icmp_port->icp_flags &= ~ICPF_WRITE_IP;
}

PRIVATE void icmp_buffree(priority)
int priority;
{
    acc_t *tmp_acc;
    int i;
    icmp_port_t *icmp_port;

    if (priority == ICMP_PRI_QUEUE)
    {
        for (i=0, icmp_port= icmp_port_table; i<ip_conf_nr;
             i++, icmp_port++)
        {
            while(icmp_port->icp_head_queue)
            {
                tmp_acc= icmp_port->icp_head_queue;
                icmp_port->icp_head_queue=
                    tmp_acc->acc_ext_link;
                bf_afree(tmp_acc);
            }
        }
    }
}

#ifdef BUF_CONSISTENCY_CHECK
PRIVATE void icmp_bufcheck()
{
    int i;
    icmp_port_t *icmp_port;
    acc_t *pack;

    for (i= 0, icmp_port= icmp_port_table; i<ip_conf_nr; i++, icmp_port++)
    {
        for (pack= icmp_port->icp_head_queue; pack;
             pack= pack->acc_ext_link)
        {
            bf_check_acc(pack);
        }
        bf_check_acc(icmp_port->icp_write_pack);
    }
}
#endif

PRIVATE void icmp_dst_unreach(icmp_port, ip_pack, ip_hdr_len, ip_hdr, icmp_pack,
                             icmp_len, icmp_hdr)
icmp_port_t *icmp_port;
acc_t *ip_pack;
int ip_hdr_len;
ip_hdr_t *ip_hdr;
acc_t *icmp_pack;
int icmp_len;
icmp_hdr_t *icmp_hdr;
{
    acc_t *old_ip_pack;
    ip_hdr_t *old_ip_hdr;
    int ip_port_nr;
    ipaddr_t dst, mask;
    size_t old_pack_size;
    ul6_t new_mtu;

    if (icmp_len < 8 + IP_MIN_HDR_SIZE)
    {
        DBLOCK(1, printf("dest unrch with wrong size\n"));
        return;
    }
    old_ip_pack= bf_cut (icmp_pack, 8, icmp_len-8);
    old_ip_pack= bf_packIfLess(old_ip_pack, IP_MIN_HDR_SIZE);
    old_ip_hdr= (ip_hdr_t *)ptr2acc_data(old_ip_pack);

    if (old_ip_hdr->ih_src != ip_hdr->ih_dst)
    {

```

```

        DBLOCK(1, printf("dest unrch based on wrong packet\n"));
        bf_afree(old_ip_pack);
        return;
    }

    ip_port_nr= icmp_port->icp_ipport;

    switch(icmp_hdr->ih_code)
    {
    case ICMP_NET_UNRCH:
        dst= old_ip_hdr->ih_dst;
        mask= ip_get_netmask(dst);
        ipr_destunrch (ip_port_nr, dst & mask, mask,
            IPR_UNRCH_TIMEOUT);
        break;
    case ICMP_HOST_UNRCH:
        ipr_destunrch (ip_port_nr, old_ip_hdr->ih_dst, (ipaddr_t)-1,
            IPR_UNRCH_TIMEOUT);
        break;
    case ICMP_PORT_UNRCH:
        /* At the moment we don't do anything with this information.
         * It should be handed to the appropriate transport layer.
         */
        break;
    case ICMP_FRAGM_AND_DF:

        DBLOCK(1, printf("icmp_dst_unreach: got mtu icmp from "));
        writeIpAddr(ip_hdr->ih_src);
        printf("; original destination: ");
        writeIpAddr(old_ip_hdr->ih_dst);
        printf("; protocol: %d\n",
            old_ip_hdr->ih_proto);
        old_pack_size= ntohs(old_ip_hdr->ih_length);
        if (!old_pack_size)
            break;
        new_mtu= ntohs(icmp_hdr->ih_hun.ihh_mtu.im_mtu);
        if (!new_mtu || new_mtu > old_pack_size)
            new_mtu= old_pack_size-1;
        ipr_mtu(ip_port_nr, old_ip_hdr->ih_dst, new_mtu,
            IPR_MTU_TIMEOUT);
        break;

    default:
        DBLOCK(1, printf("icmp_dst_unreach: got strange code %d from ",
            icmp_hdr->ih_code));
        writeIpAddr(ip_hdr->ih_src);
        printf("; original destination: ");
        writeIpAddr(old_ip_hdr->ih_dst);
        printf("; protocol: %d\n",
            old_ip_hdr->ih_proto);
        break;
    }
    bf_afree(old_ip_pack);
}

PRIVATE void icmp_time_exceeded(icmp_port, ip_pack, ip_hdr_len, ip_hdr,
    icmp_pack, icmp_len, icmp_hdr)
icmp_port_t *icmp_port;
acc_t *ip_pack;
int ip_hdr_len;
ip_hdr_t *ip_hdr;
acc_t *icmp_pack;
int icmp_len;
icmp_hdr_t *icmp_hdr;
{
    acc_t *old_ip_pack;
    ip_hdr_t *old_ip_hdr;
    int ip_port_nr;

    if (icmp_len < 8 + IP_MIN_HDR_SIZE)
    {
        DBLOCK(1, printf("time exceeded with wrong size\n"));
        return;
    }
}

```

```
old_ip_pack= bf_cut (icmp_pack, 8, icmp_len-8);
old_ip_pack= bf_packIfLess(old_ip_pack, IP_MIN_HDR_SIZE);
old_ip_hdr= (ip_hdr_t *)ptr2acc_data(old_ip_pack);

if (old_ip_hdr->ih_src != ip_hdr->ih_dst)
{
    DBLOCK(1, printf("time exceeded based on wrong packet\n"));
    bf_afree(old_ip_pack);
    return;
}

ip_port_nr= icmp_port->icp_ipport;

switch(icmp_hdr->ih_code)
{
case ICMP_TTL_EXC:
    ipr_ttl_exc (ip_port_nr, old_ip_hdr->ih_dst, (ipaddr_t)-1,
        IPR_TTL_TIMEOUT);
    break;
case ICMP_FRAG_REASSEM:
    /* Ignore reassembly time-outs. */
    break;
default:
    DBLOCK(1, printf("got strange code: %d\n",
        icmp_hdr->ih_code));
    break;
}
bf_afree(old_ip_pack);
}

PRIVATE void icmp_router_advertisement(icmp_port, icmp_pack, icmp_len, icmp_hdr)
icmp_port_t *icmp_port;
acc_t *icmp_pack;
int icmp_len;
icmp_hdr_t *icmp_hdr;
{
    int entries;
    int entry_size;
    u32_t addr;
    i32_t pref;
    ul6_t lifetime;
    int i;
    char *bufp;
    ip_port_t *ip_port;

    if (icmp_len < 8)
    {
        DBLOCK(1,
            printf("router advertisement with wrong size (%d)\n",
                icmp_len));
        return;
    }
    if (icmp_hdr->ih_code != 0)
    {
        DBLOCK(1,
            printf("router advertisement with wrong code (%d)\n",
                icmp_hdr->ih_code));
        return;
    }
    entries= icmp_hdr->ih_hun.ihh_ram.iram_na;
    entry_size= icmp_hdr->ih_hun.ihh_ram.iram_aes * 4;
    if (entries < 1)
    {
        DBLOCK(1, printf(
            "router advertisement with wrong number of entries (%d)\n",
                entries));
        return;
    }
    if (entry_size < 8)
    {
        DBLOCK(1, printf(
            "router advertisement with wrong entry size (%d)\n",
                entry_size));
        return;
    }
}
```

```

    }
    if (icmp_len < 8 + entries * entry_size)
    {
        DBLOCK(1,
            printf("router advertisement with wrong size\n");
            printf(
                "\t(entries= %d, entry_size= %d, icmp_len= %d)\n",
                entries, entry_size, icmp_len));
        return;
    }
    lifetime= ntohs(icmp_hdr->ih_hun.ihh_ram.iram_lt);
    if (lifetime > 9000)
    {
        DBLOCK(1, printf(
            "router advertisement with wrong lifetime (%d)\n",
            lifetime));
        return;
    }
    ip_port= &ip_port_table[icmp_port->icp_ipport];
    for (i= 0, bufp= (char *)&icmp_hdr->ih_dun.uhd_data[0]; i< entries; i++,
        bufp += entry_size)
    {
        addr= *(ipaddr_t *)bufp;
        pref= ntohl(*(u32_t *) (bufp+4));
        ipr_add_oroute(icmp_port->icp_ipport, HTONL(0L), HTONL(0L),
            addr, lifetime ? lifetime * HZ : 1,
            1, 0, 0, pref, NULL);
    }
}

PRIVATE void icmp_redirect(icmp_port, ip_hdr, icmp_pack, icmp_len, icmp_hdr)
icmp_port_t *icmp_port;
ip_hdr_t *ip_hdr;
acc_t *icmp_pack;
int icmp_len;
icmp_hdr_t *icmp_hdr;
{
    acc_t *old_ip_pack;
    ip_hdr_t *old_ip_hdr;
    int ip_port_nr;
    ipaddr_t dst, mask;

    if (icmp_len < 8 + IP_MIN_HDR_SIZE)
    {
        DBLOCK(1, printf("redirect with wrong size\n"));
        return;
    }
    old_ip_pack= bf_cut(icmp_pack, 8, icmp_len-8);
    old_ip_pack= bf_packIfLess(old_ip_pack, IP_MIN_HDR_SIZE);
    old_ip_hdr= (ip_hdr_t *)ptr2acc_data(old_ip_pack);

    ip_port_nr= icmp_port->icp_ipport;

    switch(icmp_hdr->ih_code)
    {
    case ICMP_REDIRECT_NET:
        dst= old_ip_hdr->ih_dst;
        mask= ip_get_netmask(dst);
        ipr_redirect(ip_port_nr, dst & mask, mask,
            ip_hdr->ih_src, icmp_hdr->ih_hun.ihh_gateway,
            IPR_REDIRECT_TIMEOUT);
        break;
    case ICMP_REDIRECT_HOST:
        ipr_redirect(ip_port_nr, old_ip_hdr->ih_dst, (ipaddr_t)-1,
            ip_hdr->ih_src, icmp_hdr->ih_hun.ihh_gateway,
            IPR_REDIRECT_TIMEOUT);
        break;
    default:
        DBLOCK(1, printf("got strange code: %d\n",
            icmp_hdr->ih_code));
        break;
    }
    bf_free(old_ip_pack);
}

```

```

PRIVATE acc_t *icmp_err_pack(pack, icmp_hdr_pp)
acc_t *pack;
icmp_hdr_t **icmp_hdr_pp;
{
    ip_hdr_t *ip_hdr;
    icmp_hdr_t *icmp_hdr_p;
    acc_t *ip_pack, *icmp_pack, *tmp_pack;
    int ip_hdr_len, icmp_hdr_len, ih_type;
    size_t size, pack_len;
    ipaddr_t dest, netmask;
    nettype_t nettype;

    pack= bf_packIfLess(pack, IP_MIN_HDR_SIZE);
    ip_hdr= (ip_hdr_t *)ptr2acc_data(pack);
    ip_hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
    pack_len= bf_bufsize(pack);

    /* If the IP protocol is ICMP (except echo request/reply) or the
     * fragment offset is non-zero,
     * drop the packet. Also check if the source address is valid.
     */
    if ((ntohs(ip_hdr->ih_flags_fragoff) & IH_FRAGOFF_MASK) != 0)
    {
        bf_afree(pack);
        return NULL;
    }
    if (ip_hdr->ih_proto == IPPROTO_ICMP)
    {
        if (ip_hdr_len > IP_MIN_HDR_SIZE)
        {
            pack= bf_packIfLess(pack, ip_hdr_len);
            ip_hdr= (ip_hdr_t *)ptr2acc_data(pack);
        }

        if (pack_len < ip_hdr_len+ICMP_MIN_HDR_SIZE)
        {
            bf_afree(pack);
            return NULL;
        }

        icmp_pack= bf_cut(pack, ip_hdr_len, ICMP_MIN_HDR_SIZE);
        icmp_pack= bf_packIfLess (icmp_pack, ICMP_MIN_HDR_SIZE);
        icmp_hdr_p= (icmp_hdr_t *)ptr2acc_data(icmp_pack);
        ih_type= icmp_hdr_p->ih_type;
        bf_afree(icmp_pack); icmp_pack= NULL;

        if (ih_type != ICMP_TYPE_ECHO_REQ &&
            ih_type != ICMP_TYPE_ECHO_REPL)
        {
            bf_afree(pack);
            return NULL;
        }
    }
    dest= ip_hdr->ih_src;
    nettype= ip_nettype(dest);
    netmask= ip_netmask(nettype);
    if (nettype != IPNT_CLASS_A && nettype != IPNT_LOCAL &&
        nettype != IPNT_CLASS_B && nettype != IPNT_CLASS_C)
    {
        printf("icmp_err_pack: invalid source address: " );
        writeIpAddr(dest);
        printf("\n");
        bf_afree(pack);
        return NULL;
    }

    /* Take the IP header and the first 64 bits of user data. */
    size= ntohs(ip_hdr->ih_length);
    if (size < ip_hdr_len || pack_len < size)
    {
        printf("icmp_err_pack: wrong packet size:\n" );
        printf("\thdrLen= %d, ih_length= %d, bufsize= %d\n",
            ip_hdr_len, size, pack_len);
        bf_afree(pack);
    }
}

```



```
        return NULL;
    }
    if (ip_hdr_len + 8 < size)
        size= ip_hdr_len+8;
    tmp_pack= bf_cut(pack, 0, size);
    bf_afree(pack);
    pack= tmp_pack;
    tmp_pack= NULL;

    /* Create a minimal size ICMP hdr. */
    icmp_hdr_len= offsetof(icmp_hdr_t, ih_dun);
    icmp_pack= bf_memreq(icmp_hdr_len);
    pack= bf_append(icmp_pack, pack);
    size += icmp_hdr_len;
    pack= bf_packIfLess(pack, icmp_hdr_len);
    icmp_hdr_p= (icmp_hdr_t *)ptr2acc_data(pack);
    icmp_hdr_p->ih_type= 0;
    icmp_hdr_p->ih_code= 0;
    icmp_hdr_p->ih_chksum= 0;
    icmp_hdr_p->ih_hun.ihh_unused= 0;
    icmp_hdr_p->ih_chksum= ~icmp_pack_oneCsum(pack);
    *icmp_hdr_pp= icmp_hdr_p;

    /* Create an IP header */
    ip_hdr_len= IP_MIN_HDR_SIZE;

    ip_pack= bf_memreq(ip_hdr_len);
    ip_hdr= (ip_hdr_t *)ptr2acc_data(ip_pack);

    ip_hdr->ih_vers_ihl= ip_hdr_len >> 2;
    ip_hdr->ih_tos= 0;
    ip_hdr->ih_length= htons(ip_hdr_len + size);
    ip_hdr->ih_flags_fragoff= 0;
    ip_hdr->ih_ttl= ICMP_DEF_TTL;
    ip_hdr->ih_proto= IPPROTO_ICMP;
    ip_hdr->ih_dst= dest;

    assert(ip_pack->acc_next == NULL);
    ip_pack->acc_next= pack;
    return ip_pack;
}

/*
 * $PchId: icmp.c,v 1.23 2005/06/28 14:16:56 philip Exp $
 */
```

```
/*
icmp.h

Copyright 1995 Philip Homburg
*/

#ifndef ICMP_H
#define ICMP_H

#define ICMP_MAX_DATAGRAM      8196
#define ICMP_DEF_TTL           96

/* Rate limit. The implementation is a bit sloppy and may send twice the
 * number of packets.
 */
#define ICMP_MAX_RATE           100      /* This many per interval */
#define ICMP_RATE_INTERVAL     (1*HZ)   /* Interval in ticks */
#define ICMP_RATE_WARN          10       /* Report this many dropped packets */

/* Prototypes */

void icmp_prep ARGS(( void ));
void icmp_init ARGS(( void ));

#endif /* ICMP_H */

/*
 * $PchId: icmp.h,v 1.7 2001/04/19 19:06:18 philip Exp $
 */
```

```
/*
icmp_lib.h

Created Sept 30, 1991 by Philip Homburg

Copyright 1995 Philip Homburg
*/

#ifndef ICMP_LIB_H
#define ICMP_LIB_H

/* Prototypes */

void icmp_snd_parmproblem ARGS(( acc_t *pack ));
void icmp_snd_time_exceeded ARGS(( int port_nr, acc_t *pack, int code ));
void icmp_snd_unreachable ARGS(( int port_nr, acc_t *pack, int code ));
void icmp_snd_redirect ARGS(( int port_nr, acc_t *pack, int code,
                             ipaddr_t gw ));
void icmp_snd_mtu ARGS(( int port_nr, acc_t *pack, U16_t mtu ));

#endif /* ICMP_LIB_H */

/*
 * $PchId: icmp_lib.h,v 1.6 2002/06/08 21:32:44 philip Exp $
 */
```

```
/*
io.c

Copyright 1995 Philip Homburg
*/

#include <stdlib.h>

#include "inet.h"
#include "io.h"

PUBLIC void writeIpAddr(addr)
ipaddr_t addr;
{
#define addrInBytes ((u8_t *)&addr)

    printf("%d.%d.%d.%d", addrInBytes[0], addrInBytes[1],
           addrInBytes[2], addrInBytes[3]);
#undef addrInBytes
}

PUBLIC void writeEtherAddr(addr)
ether_addr_t *addr;
{
#define addrInBytes ((u8_t *)addr->ea_addr)

    printf("%x:%x:%x:%x:%x:%x", addrInBytes[0], addrInBytes[1],
           addrInBytes[2], addrInBytes[3], addrInBytes[4], addrInBytes[5]);
#undef addrInBytes
}

/*
 * $PchId: io.c,v 1.6 1998/10/23 20:24:34 philip Exp $
 */
```

```
/*
io.h

Created Sept 30, 1991 by Philip Homburg

Copyright 1995 Philip Homburg
*/

#ifndef IO_H
#define IO_H

/* Prototypes */

void writeIpAddr ARGS(( ipaddr_t addr ));
void writeEtherAddr ARGS(( ether_addr_t *addr ));

#endif /* IO_H */

/*
 * $PchId: io.h,v 1.4 1995/11/21 06:45:27 philip Exp $
 */
```

```
/*
ip.c

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "buf.h"
#include "event.h"
#include "type.h"

#include "arp.h"
#include "assert.h"
#include "clock.h"
#include "eth.h"
#include "icmp.h"
#include "icmp_lib.h"
#include "io.h"
#include "ip.h"
#include "ip_int.h"
#include "ipr.h"
#include "sr.h"

THIS_FILE

FORWARD void ip_close ARGS(( int fd ));
FORWARD int ip_cancel ARGS(( int fd, int which_operation ));
FORWARD int ip_select ARGS(( int fd, unsigned operations ));

FORWARD void ip_buffree ARGS(( int priority ));
#ifdef BUF_CONSISTENCY_CHECK
FORWARD void ip_bufcheck ARGS(( void ));
#endif
FORWARD void ip_bad_callback ARGS(( struct ip_port *ip_port ));

PUBLIC ip_port_t *ip_port_table;
PUBLIC ip_fd_t ip_fd_table[IP_FD_NR];
PUBLIC ip_ass_t ip_ass_table[IP_ASS_NR];

PUBLIC void ip_prep()
{
    ip_port_table= alloc(ip_conf_nr * sizeof(ip_port_table[0]));
    icmp_prep();
}

PUBLIC void ip_init()
{
    int i, j, result;
    ip_ass_t *ip_ass;
    ip_fd_t *ip_fd;
    ip_port_t *ip_port;
    struct ip_conf *icp;

    assert (BUF_S >= sizeof(struct nwio_ethopt));
    assert (BUF_S >= IP_MAX_HDR_SIZE + ETH_HDR_SIZE);
    assert (BUF_S >= sizeof(nwio_ipopt_t));
    assert (BUF_S >= sizeof(nwio_route_t));

    for (i=0, ip_ass= ip_ass_table; i<IP_ASS_NR; i++, ip_ass++)
    {
        ip_ass->ia_fragments= 0;
        ip_ass->ia_first_time= 0;
        ip_ass->ia_port= 0;
    }

    for (i=0, ip_fd= ip_fd_table; i<IP_FD_NR; i++, ip_fd++)
    {
        ip_fd->if_flags= IFF_EMPTY;
        ip_fd->if_rdbuf_head= 0;
    }

    for (i=0, ip_port= ip_port_table, icp= ip_conf;
        i<ip_conf_nr; i++, ip_port++, icp++)
    {
```

```

ip_port->ip_port= i;
ip_port->ip_flags= IPF_EMPTY;
ip_port->ip_dev_main= (ip_dev_t)ip_bad_callback;
ip_port->ip_dev_set_ipaddr= (ip_dev_t)ip_bad_callback;
ip_port->ip_dev_send= (ip_dev_send_t)ip_bad_callback;
ip_port->ip_dl_type= icp->ic_devtype;
ip_port->ip_mtu= IP_DEF_MTU;
ip_port->ip_mtu_max= IP_MAX_PACKSIZE;

switch(ip_port->ip_dl_type)
{
case IPDL_ETH:
    ip_port->ip_dl.dl_eth.de_port= icp->ic_port;
    result= ipeth_init(ip_port);
    if (result == -1)
        continue;
    assert(result == NW_OK);
    break;
case IPDL_PSIP:
    ip_port->ip_dl.dl_ps.ps_port= icp->ic_port;
    result= ipps_init(ip_port);
    if (result == -1)
        continue;
    assert(result == NW_OK);
    break;
default:
    ip_panic(( "unknown ip_dl_type %d",
               ip_port->ip_dl_type ));
    break;
}
ip_port->ip_loopb_head= NULL;
ip_port->ip_loopb_tail= NULL;
ev_init(&ip_port->ip_loopb_event);
ip_port->ip_routeq_head= NULL;
ip_port->ip_routeq_tail= NULL;
ev_init(&ip_port->ip_routeq_event);
ip_port->ip_flags |= IPF_CONFIGURED;
ip_port->ip_proto_any= NULL;
for (j= 0; j<IP_PROTO_HASH_NR; j++)
    ip_port->ip_proto[j]= NULL;
}

#ifdef BUF_CONSISTENCY_CHECK
    bf_logon(ip_buffree);
#else
    bf_logon(ip_buffree, ip_bufcheck);
#endif

icmp_init();
ipr_init();

for (i=0, ip_port= ip_port_table; i<ip_conf_nr; i++, ip_port++)
{
    if (!(ip_port->ip_flags & IPF_CONFIGURED))
        continue;
    ip_port->ip_frame_id= (ul6_t)get_time();

    sr_add_minor(if2minor(ip_conf[i].ic_ifno, IP_DEV_OFF),
        i, ip_open, ip_close, ip_read,
        ip_write, ip_ioctl, ip_cancel, ip_select);

    (*ip_port->ip_dev_main)(ip_port);
}
}

PRIVATE int ip_cancel (fd, which_operation)
int fd;
int which_operation;
{
    ip_fd_t *ip_fd;
    acc_t *repl_res;
    int result;

    ip_fd= &ip_fd_table[fd];

```

```

    switch (which_operation)
    {
    case SR_CANCEL_IOCTL:
        assert (ip_fd->if_flags & IFF_IOCTL_IP);
        ip_fd->if_flags &= ~IFF_IOCTL_IP;
        repl_res= (*ip_fd->if_get_userdata)(ip_fd->if_srfd,
            (size_t)EINTR, (size_t)0, TRUE);
        assert (!repl_res);
        break;
    case SR_CANCEL_READ:
        assert (ip_fd->if_flags & IFF_READ_IP);
        ip_fd->if_flags &= ~IFF_READ_IP;
        result= (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
            (size_t)EINTR, (acc_t *)0, FALSE);
        assert (!result);
        break;
    case SR_CANCEL_WRITE:
        assert(0);
        assert (ip_fd->if_flags & IFF_WRITE_MASK);
        ip_fd->if_flags &= ~IFF_WRITE_MASK;
        repl_res= (*ip_fd->if_get_userdata)(ip_fd->if_srfd,
            (size_t)EINTR, (size_t)0, FALSE);
        assert (!repl_res);
        break;
    default:
        ip_panic(( "unknown cancel request" ));
        break;
    }
    return NW_OK;
}

PRIVATE int ip_select(fd, operations)
int fd;
unsigned operations;
{
    printf("ip_select: not implemented\n");
    return 0;
}

PUBLIC int ip_open (port, srfd, get_userdata, put_userdata, put_pkt,
    select_res)
int port;
int srfd;
get_userdata_t get_userdata;
put_userdata_t put_userdata;
put_pkt_t put_pkt;
select_res_t select_res;
{
    int i;
    ip_fd_t *ip_fd;
    ip_port_t *ip_port;

    ip_port= &ip_port_table[port];
    if (!(ip_port->ip_flags & IPF_CONFIGURED))
        return ENXIO;

    for (i=0; i<IP_FD_NR && (ip_fd_table[i].if_flags & IFF_INUSE);
        i++);

    if (i>=IP_FD_NR)
    {
        DBLOCK(1, printf("out of fds\n"));
        return EAGAIN;
    }

    ip_fd= &ip_fd_table[i];

    ip_fd->if_flags= IFF_INUSE;

    ip_fd->if_ipopt.nwio_flags= NWIO_DEFAULT;
    ip_fd->if_ipopt.nwio_tos= 0;

```



```

    ip_fd->if_ipopt.nwio_df= FALSE;
    ip_fd->if_ipopt.nwio_ttl= 255;
    ip_fd->if_ipopt.nwio_hdopt.iho_opt_siz= 0;

    ip_fd->if_port= ip_port;
    ip_fd->if_srfd= srfd;
    assert(ip_fd->if_rdbuf_head == NULL);
    ip_fd->if_get_userdata= get_userdata;
    ip_fd->if_put_userdata= put_userdata;
    ip_fd->if_put_pkt= put_pkt;

    return i;
}

PRIVATE void ip_close (fd)
int fd;
{
    ip_fd_t *ip_fd;
    acc_t *pack;

    ip_fd= &ip_fd_table[fd];

    assert ((ip_fd->if_flags & IFF_INUSE) &&
            !(ip_fd->if_flags & IFF_BUSY));

    if (ip_fd->if_flags & IFF_OPTSET)
        ip_unhash_proto(ip_fd);
    while (ip_fd->if_rdbuf_head)
    {
        pack= ip_fd->if_rdbuf_head;
        ip_fd->if_rdbuf_head= pack->acc_ext_link;
        bf_afree(pack);
    }
    ip_fd->if_flags= IFF_EMPTY;
}

PRIVATE void ip_buffree(priority)
int priority;
{
    int i;
    ip_port_t *ip_port;
    ip_fd_t *ip_fd;
    ip_ass_t *ip_ass;
    acc_t *pack, *next_pack;

    for (i= 0, ip_port= ip_port_table; i<ip_conf_nr; i++, ip_port++)
    {
        if (ip_port->ip_dl_type == IPDL_ETH)
        {
            /* Can't free de_frame.
             * bf_check_acc(ip_port->ip_dl.dl_eth.de_frame);
             */
            if (priority == IP_PRI_PORTBUFS)
            {
                next_pack= ip_port->ip_dl.dl_eth.de_arp_head;
                while(next_pack != NULL)
                {
                    pack= next_pack;
                    next_pack= pack->acc_ext_link;
                    bf_afree(pack);
                }
                ip_port->ip_dl.dl_eth.de_arp_head= next_pack;

                next_pack= ip_port->ip_dl.dl_eth.de_q_head;
                while(next_pack != NULL)
                {
                    pack= next_pack;
                    next_pack= pack->acc_ext_link;
                    bf_afree(pack);
                }
                ip_port->ip_dl.dl_eth.de_q_head= next_pack;
            }
        }
        else if (ip_port->ip_dl_type == IPDL_PSIP)

```

```

        {
            if (priority == IP_PRI_PORTBUFS)
            {
                next_pack= ip_port->ip_dl.dl_ps.ps_send_head;
                while (next_pack != NULL)
                {
                    pack= next_pack;
                    next_pack= pack->acc_ext_link;
                    bf_afree(pack);
                }
                ip_port->ip_dl.dl_ps.ps_send_head= next_pack;
            }
        }
        if (priority == IP_PRI_PORTBUFS)
        {
            next_pack= ip_port->ip_loopb_head;
            while(next_pack && next_pack->acc_ext_link)
            {
                pack= next_pack;
                next_pack= pack->acc_ext_link;
                bf_afree(pack);
            }
            if (next_pack)
            {
                if (ev_in_queue(&ip_port->ip_loopb_event))
                {
                    printf(
#if DEBUG
"not freeing ip_loopb_head, ip_loopb_event enqueued\n" );
#endif
                }
                else
                {
                    bf_afree(next_pack);
                    next_pack= NULL;
                }
            }
            ip_port->ip_loopb_head= next_pack;

            next_pack= ip_port->ip_routeq_head;
            while(next_pack && next_pack->acc_ext_link)
            {
                pack= next_pack;
                next_pack= pack->acc_ext_link;
                bf_afree(pack);
            }
            if (next_pack)
            {
                if (ev_in_queue(&ip_port->ip_routeq_event))
                {
                    printf(
#if DEBUG
"not freeing ip_loopb_head, ip_routeq_event enqueued\n" );
#endif
                }
                else
                {
                    bf_afree(next_pack);
                    next_pack= NULL;
                }
            }
            ip_port->ip_routeq_head= next_pack;
        }
    }
    if (priority == IP_PRI_FDBUFS_EXTRA)
    {
        for (i= 0, ip_fd= ip_fd_table; i<IP_FD_NR; i++, ip_fd++)
        {
            while (ip_fd->if_rdbuf_head &&
                    ip_fd->if_rdbuf_head->acc_ext_link)
            {
                pack= ip_fd->if_rdbuf_head;
                ip_fd->if_rdbuf_head= pack->acc_ext_link;
                bf_afree(pack);
            }
        }
    }
}

```

```

    }
}
if (priority == IP_PRI_FDBUFS)
{
    for (i= 0, ip_fd= ip_fd_table; i<IP_FD_NR; i++, ip_fd++)
    {
        while (ip_fd->if_rdbuf_head)
        {
            pack= ip_fd->if_rdbuf_head;
            ip_fd->if_rdbuf_head= pack->acc_ext_link;
            bf_afree(pack);
        }
    }
}
if (priority == IP_PRI_ASSBUFS)
{
    for (i= 0, ip_ass= ip_ass_table; i<IP_ASS_NR; i++, ip_ass++)
    {
        next_pack= ip_ass->ia_frgs;
        while(ip_ass->ia_frgs != NULL)
        {
            pack= ip_ass->ia_frgs;
            ip_ass->ia_frgs= pack->acc_ext_link;
            bf_afree(pack);
        }
        ip_ass->ia_first_time= 0;
    }
}
}

#ifdef BUF_CONSISTENCY_CHECK
PRIVATE void ip_bufcheck()
{
    int i;
    ip_port_t *ip_port;
    ip_fd_t *ip_fd;
    ip_ass_t *ip_ass;
    acc_t *pack;

    for (i= 0, ip_port= ip_port_table; i<ip_conf_nr; i++, ip_port++)
    {
        if (ip_port->ip_dl_type == IPDL_ETH)
        {
            bf_check_acc(ip_port->ip_dl.dl_eth.de_frame);
            for (pack= ip_port->ip_dl.dl_eth.de_q_head; pack;
                 pack= pack->acc_ext_link)
            {
                bf_check_acc(pack);
            }
            for (pack= ip_port->ip_dl.dl_eth.de_arp_head; pack;
                 pack= pack->acc_ext_link)
            {
                bf_check_acc(pack);
            }
        }
        else if (ip_port->ip_dl_type == IPDL_PSIP)
        {
            for (pack= ip_port->ip_dl.dl_ps.ps_send_head; pack;
                 pack= pack->acc_ext_link)
            {
                bf_check_acc(pack);
            }
        }
        for (pack= ip_port->ip_loopb_head; pack;
             pack= pack->acc_ext_link)
        {
            bf_check_acc(pack);
        }
        for (pack= ip_port->ip_routeq_head; pack;
             pack= pack->acc_ext_link)
        {
            bf_check_acc(pack);
        }
    }
}

```

```
    }
    for (i= 0, ip_fd= ip_fd_table; i<IP_FD_NR; i++, ip_fd++)
    {
        for (pack= ip_fd->if_rdbuf_head; pack;
             pack= pack->acc_ext_link)
        {
            bf_check_acc(pack);
        }
    }
    for (i= 0, ip_ass= ip_ass_table; i<IP_ASS_NR; i++, ip_ass++)
    {
        for (pack= ip_ass->ia_frags; pack; pack= pack->acc_ext_link)
            bf_check_acc(pack);
    }
}
#endif /* BUF_CONSISTENCY_CHECK */

PRIVATE void ip_bad_callback(ip_port)
struct ip_port *ip_port;
{
    ip_panic(( "no callback filled in for port %d", ip_port->ip_port ));
}

/*
 * $PchId: ip.c,v 1.19 2005/06/28 14:17:40 philip Exp $
 */
```

```
/*
ip.h

Copyright 1995 Philip Homburg
*/

#ifndef INET_IP_H
#define INET_IP_H

/* Prototypes */

struct acc;

void ip_prep ARGS(( void ));
void ip_init ARGS(( void ));
int ip_open ARGS(( int port, int srfd,
    get_userdata_t get_userdata, put_userdata_t put_userdata,
    put_pkt_t put_pkt, select_res_t select_res ));
int ip_ioctl ARGS(( int fd, ioreq_t req ));
int ip_read ARGS(( int fd, size_t count ));
int ip_write ARGS(( int fd, size_t count ));
int ip_send ARGS(( int fd, struct acc *data, size_t data_len ));

#endif /* INET_IP_H */

/*
 * $PchId: ip.h,v 1.8 2005/06/28 14:17:57 philip Exp $
 */
```

```
/*
generic/ip_eth.c

Ethernet specific part of the IP implementation

Created:      Apr 22, 1993 by Philip Homburg

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "type.h"
#include "arp.h"
#include "assert.h"
#include "buf.h"
#include "clock.h"
#include "eth.h"
#include "event.h"
#include "icmp_lib.h"
#include "io.h"
#include "ip.h"
#include "ip_int.h"

THIS_FILE

typedef struct xmit_hdr
{
    time_t xh_time;
    ipaddr_t xh_ipaddr;
} xmit_hdr_t;

PRIVATE ether_addr_t broadcast_ethaddr=
{
    { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff }
};

PRIVATE ether_addr_t ipmulticast_ethaddr=
{
    { 0x01, 0x00, 0x5e, 0x00, 0x00, 0x00 }
};

FORWARD void do_eth_read ARGS(( ip_port_t *port ));
FORWARD acc_t *get_eth_data ARGS(( int fd, size_t offset,
    size_t count, int for_ioctl ));
FORWARD int put_eth_data ARGS(( int fd, size_t offset,
    acc_t *data, int for_ioctl ));
FORWARD void ipeth_main ARGS(( ip_port_t *port ));
FORWARD void ipeth_set_ipaddr ARGS(( ip_port_t *port ));
FORWARD void ipeth_restart_send ARGS(( ip_port_t *ip_port ));
FORWARD int ipeth_send ARGS(( struct ip_port *ip_port, ipaddr_t dest,
    acc_t *pack, int type ));
FORWARD void ipeth_arp_reply ARGS(( int ip_port_nr, ipaddr_t ipaddr,
    ether_addr_t *dst_ether_ptr ));
FORWARD int ipeth_update_ttl ARGS(( time_t enq_time, time_t now,
    acc_t *eth_pack ));
FORWARD void ip_eth_arrived ARGS(( int port, acc_t *pack,
    size_t pack_size ));

PUBLIC int ipeth_init(ip_port)
ip_port_t *ip_port;
{
    assert(BUF_S >= sizeof(xmit_hdr_t));
    assert(BUF_S >= sizeof(eth_hdr_t));

    ip_port->ip_dl.dl_eth.de_fd= eth_open(ip_port->
        ip_dl.dl_eth.de_port, ip_port->ip_port,
        get_eth_data, put_eth_data, ip_eth_arrived,
        0 /* no select_res */);
    if (ip_port->ip_dl.dl_eth.de_fd < 0)
    {
        DBLOCK(1, printf("ip.c: unable to open eth port\n"));
        return -1;
    }
    ip_port->ip_dl.dl_eth.de_state= IES_EMPTY;
```

```

    ip_port->ip_dl.dl_eth.de_flags= IEF_EMPTY;
    ip_port->ip_dl.dl_eth.de_q_head= NULL;
    ip_port->ip_dl.dl_eth.de_q_tail= NULL;
    ip_port->ip_dl.dl_eth.de_arp_head= NULL;
    ip_port->ip_dl.dl_eth.de_arp_tail= NULL;
    ip_port->ip_dev_main= ipeth_main;
    ip_port->ip_dev_set_ipaddr= ipeth_set_ipaddr;
    ip_port->ip_dev_send= ipeth_send;
    ip_port->ip_mtu= ETH_MAX_PACK_SIZE-ETH_HDR_SIZE;
    ip_port->ip_mtu_max= ip_port->ip_mtu;
    return 0;
}

PRIVATE void ipeth_main(ip_port)
ip_port_t *ip_port;
{
    int result;

    switch (ip_port->ip_dl.dl_eth.de_state)
    {
    case IES_EMPTY:
        ip_port->ip_dl.dl_eth.de_state= IES_SETPROTO;

        result= eth_ioctl(ip_port->ip_dl.dl_eth.de_fd, NWIOSETHOPT);
        if (result == NW_SUSPEND)
            ip_port->ip_dl.dl_eth.de_flags |= IEF_SUSPEND;
        if (result<0)
        {
            DBLOCK(1, printf("eth_ioctl(..0x%lx)=%d\n",
                (unsigned long)NWIOSETHOPT, result));
            return;
        }
        if (ip_port->ip_dl.dl_eth.de_state != IES_SETPROTO)
            return;
        /* drops through */
    case IES_SETPROTO:
        result= arp_set_cb(ip_port->ip_dl.dl_eth.de_port,
            ip_port->ip_port,
            ipeth_arp_reply);
        if (result != NW_OK)
        {
            printf("ipeth_main: arp_set_cb failed: %d\n",
                result);
            return;
        }

        /* Wait until the interface is configured up. */
        ip_port->ip_dl.dl_eth.de_state= IES_GETIPADDR;
        if (!(ip_port->ip_flags & IPF_IPADDRSET))
        {
            ip_port->ip_dl.dl_eth.de_flags |= IEF_SUSPEND;
            return;
        }

        /* fall through */
    case IES_GETIPADDR:
        ip_port->ip_dl.dl_eth.de_state= IES_MAIN;
        do_eth_read(ip_port);
        return;
    default:
        ip_panic(("unknown state: %d",
            ip_port->ip_dl.dl_eth.de_state));
    }
}

PRIVATE acc_t *get_eth_data (fd, offset, count, for_ioctl)
int fd;
size_t offset;
size_t count;
int for_ioctl;
{
    ip_port_t *ip_port;
    acc_t *data;
    int result;

```

```

ip_port= &ip_port_table[fd];

switch (ip_port->ip_dl.dl_eth.de_state)
{
case IES_SETPROTO:
    if (!count)
    {
        result= (int)offset;
        if (result<0)
        {
            ip_port->ip_dl.dl_eth.de_state= IES_ERROR;
            break;
        }
        if (ip_port->ip_dl.dl_eth.de_flags & IEF_SUSPEND)
            ipeth_main(ip_port);
        return NW_OK;
    }
    assert ((!offset) && (count == sizeof(struct nwio_ethopt)));
    {
        struct nwio_ethopt *ethopt;
        acc_t *acc;

        acc= bf_memreq(sizeof(*ethopt));
        ethopt= (struct nwio_ethopt *)ptr2acc_data(acc);
        ethopt->nweo_flags= NWEO_COPY|NWEO_EN_BROAD|
            NWEO_EN_MULTI|NWEO_TYPESPEC;
        ethopt->nweo_type= HTONS(ETH_IP_PROTO);
        return acc;
    }

case IES_MAIN:
    if (!count)
    {
        result= (int)offset;
        if (result<0)
            ip_warning(( "error on write: %d\n", result ));
        bf_afree (ip_port->ip_dl.dl_eth.de_frame);
        ip_port->ip_dl.dl_eth.de_frame= 0;

        if (ip_port->ip_dl.dl_eth.de_flags & IEF_WRITE_SP)
        {
            ip_port->ip_dl.dl_eth.de_flags &=
                ~IEF_WRITE_SP;
            ipeth_restart_send(ip_port);
        }
        return NW_OK;
    }
    data= bf_cut (ip_port->ip_dl.dl_eth.de_frame, offset, count);
    assert (data);
    return data;
default:
    printf(
        "get_eth_data(%d, 0x%d, 0x%d) called but ip_state=0x%x\n",
        fd, offset, count, ip_port->ip_dl.dl_eth.de_state);
    break;
}
return 0;
}

PRIVATE int put_eth_data (port, offset, data, for_ioctl)
int port;
size_t offset;
acc_t *data;
int for_ioctl;
{
    ip_port_t *ip_port;
    int result;

    ip_port= &ip_port_table[port];

    assert(0);

    if (ip_port->ip_dl.dl_eth.de_flags & IEF_READ_IP)

```



```

    {
        if (!data)
        {
            result= (int)offset;
            if (result<0)
            {
                DBLOCK(1, printf(
                    "ip.c: put_eth_data(..,%d,..)\n", result));
                return NW_OK;
            }
            if (ip_port->ip_dl.dl_eth.de_flags & IEF_READ_SP)
            {
                ip_port->ip_dl.dl_eth.de_flags &=
                    ~(IEF_READ_IP|IEF_READ_SP);
                do_eth_read(ip_port);
            }
            else
                ip_port->ip_dl.dl_eth.de_flags &= ~IEF_READ_IP;
            return NW_OK;
        }
        assert (!offset);
        /* Warning: the above assertion is illegal; puts and
           gets of data can be brokenup in any piece the server
           likes. However we assume that the server is eth.c
           and it transfers only whole packets. */
        ip_eth_arrived(port, data, bf_bufsize(data));
        return NW_OK;
    }
    printf("ip_port->ip_dl.dl_eth.de_state= 0x%x",
        ip_port->ip_dl.dl_eth.de_state);
    ip_panic (( "strange status" ));
}

PRIVATE void ipeth_set_ipaddr(ip_port)
ip_port_t *ip_port;
{
    arp_set_ipaddr (ip_port->ip_dl.dl_eth.de_port, ip_port->ip_ipaddr);
    if (ip_port->ip_dl.dl_eth.de_state == IES_GETIPADDR)
        ipeth_main(ip_port);
}

PRIVATE int ipeth_send(ip_port, dest, pack, type)
struct ip_port *ip_port;
ipaddr_t dest;
acc_t *pack;
int type;
{
    int i, r;
    acc_t *eth_pack, *tail;
    size_t pack_size;
    eth_hdr_t *eth_hdr;
    xmit_hdr_t *xmit_hdr;
    ipaddr_t hostpart, tmpaddr;
    time_t t;
    u32_t *p;

    /* Start optimistic: the arp will succeed without blocking and the
     * ethernet packet can be sent without blocking also. Start with
     * the allocation of the ethernet header.
     */
    eth_pack= bf_memreq(sizeof(*eth_hdr));
    assert(eth_pack->acc_next == NULL);
    eth_pack->acc_next= pack;
    pack_size= bf_bufsize(eth_pack);
    if (pack_size<ETH_MIN_PACK_SIZE)
    {
        tail= bf_memreq(ETH_MIN_PACK_SIZE-pack_size);

        /* Clear padding */
        for (i= (ETH_MIN_PACK_SIZE-pack_size)/sizeof(*p),
            p= (u32_t *)ptr2acc_data(tail);
            i >= 0; i--, p++)
        {
            *p= 0xdeadbeef;
        }
    }
}

```

```

    }

    eth_pack= bf_append(eth_pack, tail);
}
eth_hdr= (eth_hdr_t *)ptr2acc_data(eth_pack);

/* Lookup the ethernet address */
if (type != IP_LT_NORMAL)
{
    if (type == IP_LT_BROADCAST)
        eth_hdr->eh_dst= broadcast_ethaddr;
    else
    {
        tmpaddr= ntohl(dest);
        eth_hdr->eh_dst= ipmulticast_ethaddr;
        eth_hdr->eh_dst.ea_addr[5]= tmpaddr & 0xff;
        eth_hdr->eh_dst.ea_addr[4]= (tmpaddr >> 8) & 0xff;
        eth_hdr->eh_dst.ea_addr[3]= (tmpaddr >> 16) & 0x7f;
    }
}
else
{
    if ((dest ^ ip_port->ip_ipaddr) & ip_port->ip_subnetmask)
    {
        ip_panic(( "invalid destination" ));
    }

    hostpart= (dest & ~ip_port->ip_subnetmask);
    assert(dest != ip_port->ip_ipaddr);

    r= arp_ip_eth(ip_port->ip_dl.dl_eth.de_port,
        dest, &eth_hdr->eh_dst);
    if (r == NW_SUSPEND)
    {
        /* Unfortunately, the arp takes some time, use
         * the ethernet header to store the next hop
         * ip address and the current time.
         */
        xmit_hdr= (xmit_hdr_t *)eth_hdr;
        xmit_hdr->xh_time= get_time();
        xmit_hdr->xh_ipaddr= dest;
        eth_pack->acc_ext_link= NULL;
        if (ip_port->ip_dl.dl_eth.de_arp_head == NULL)
            ip_port->ip_dl.dl_eth.de_arp_head= eth_pack;
        else
        {
            ip_port->ip_dl.dl_eth.de_arp_tail->
                acc_ext_link= eth_pack;
        }
        ip_port->ip_dl.dl_eth.de_arp_tail= eth_pack;
        return NW_OK;
    }
    if (r == EDSTNOTRCH)
    {
        bf_afree(eth_pack);
        return EDSTNOTRCH;
    }
    assert(r == NW_OK);
}

/* If we have no write in progress, we can try to send the ethernet
 * packet using eth_send. If the IP packet is larger than mtu,
 * enqueue the packet and let ipeth_restart_send deal with it.
 */
pack_size= bf_bufsize(eth_pack);
if (ip_port->ip_dl.dl_eth.de_frame == NULL && pack_size <=
    ip_port->ip_mtu + sizeof(*eth_hdr))
{
    r= eth_send(ip_port->ip_dl.dl_eth.de_fd,
        eth_pack, pack_size);
    if (r == NW_OK)
        return NW_OK;

    /* A non-blocking send is not possible, start a regular

```

```

        * send.
        */
    assert(r == NW_WOULDBLOCK);
    ip_port->ip_dl.dl_eth.de_frame= eth_pack;
    r= eth_write(ip_port->ip_dl.dl_eth.de_fd, pack_size);
    if (r == NW_SUSPEND)
    {
        assert(!(ip_port->ip_dl.dl_eth.de_flags &
            IEF_WRITE_SP));
        ip_port->ip_dl.dl_eth.de_flags |= IEF_WRITE_SP;
    }
    assert(r == NW_OK || r == NW_SUSPEND);
    return NW_OK;
}

/* Enqueue the packet, and store the current time, in the
 * space for the ethernet source address.
 */
t= get_time();
assert(sizeof(t) <= sizeof(eth_hdr->eh_src));
memcpy(&eth_hdr->eh_src, &t, sizeof(t));

eth_pack->acc_ext_link= NULL;
if (ip_port->ip_dl.dl_eth.de_q_head == NULL)
    ip_port->ip_dl.dl_eth.de_q_head= eth_pack;
else
{
    ip_port->ip_dl.dl_eth.de_q_tail->acc_ext_link= eth_pack;
}
ip_port->ip_dl.dl_eth.de_q_tail= eth_pack;
if (ip_port->ip_dl.dl_eth.de_frame == NULL)
    ipeth_restart_send(ip_port);
return NW_OK;
}

PRIVATE void ipeth_restart_send(ip_port)
ip_port_t *ip_port;
{
    time_t now, enq_time;
    int i, r;
    acc_t *eth_pack, *ip_pack, *next_eth_pack, *next_part, *tail;
    size_t pack_size;
    eth_hdr_t *eth_hdr, *next_eth_hdr;
    u32_t *p;

    now= get_time();

    while (ip_port->ip_dl.dl_eth.de_q_head != NULL)
    {
        eth_pack= ip_port->ip_dl.dl_eth.de_q_head;
        ip_port->ip_dl.dl_eth.de_q_head= eth_pack->acc_ext_link;

        eth_hdr= (eth_hdr_t *)ptr2acc_data(eth_pack);

        pack_size= bf_bufsize(eth_pack);

        if (pack_size > ip_port->ip_mtu+sizeof(*eth_hdr))
        {
            /* Split the IP packet */
            assert(eth_pack->acc_linkC == 1);
            ip_pack= eth_pack->acc_next; eth_pack->acc_next= NULL;
            next_part= ip_pack; ip_pack= NULL;
            ip_pack= ip_split_pack(ip_port, &next_part,
                ip_port->ip_mtu);

            if (ip_pack == NULL)
            {
                bf_afree(eth_pack);
                continue;
            }

            eth_pack->acc_next= ip_pack; ip_pack= NULL;

            /* Allocate new ethernet header */
            next_eth_pack= bf_memreq(sizeof(*next_eth_hdr));

```

```

        next_eth_hdr= (eth_hdr_t *)ptr2acc_data(next_eth_pack);
        *next_eth_hdr= *eth_hdr;
        next_eth_pack->acc_next= next_part;

        next_eth_pack->acc_ext_link= NULL;
        if (ip_port->ip_dl.dl_eth.de_q_head == NULL)
            ip_port->ip_dl.dl_eth.de_q_head= next_eth_pack;
        else
        {
            ip_port->ip_dl.dl_eth.de_q_tail->acc_ext_link=
                next_eth_pack;
        }
        ip_port->ip_dl.dl_eth.de_q_tail= next_eth_pack;

        pack_size= bf_bufsize(eth_pack);
    }

    memcpy(&enq_time, &eth_hdr->eh_src, sizeof(enq_time));
    if (enq_time + HZ < now)
    {
        r= ipeth_update_ttl(enq_time, now, eth_pack);
        if (r == ETIMEDOUT)
        {
            ip_pack= bf_delhead(eth_pack, sizeof(*eth_hdr));
            eth_pack= NULL;
            icmp_snd_time_exceeded(ip_port->ip_port,
                ip_pack, ICMP_TTL_EXC);
            continue;
        }
        assert(r == NW_OK);
    }

    if (pack_size<ETH_MIN_PACK_SIZE)
    {
        tail= bf_memreq(ETH_MIN_PACK_SIZE-pack_size);

        /* Clear padding */
        for (i= (ETH_MIN_PACK_SIZE-pack_size)/sizeof(*p),
            p= (u32_t *)ptr2acc_data(tail);
            i >= 0; i--, p++)
        {
            *p= 0xdeadbeef;
        }

        eth_pack= bf_append(eth_pack, tail);
        pack_size= ETH_MIN_PACK_SIZE;
    }

    assert(ip_port->ip_dl.dl_eth.de_frame == NULL);

    r= eth_send(ip_port->ip_dl.dl_eth.de_fd, eth_pack, pack_size);
    if (r == NW_OK)
        continue;

    /* A non-blocking send is not possible, start a regular
     * send.
     */
    assert(r == NW_WOULD_BLOCK);
    ip_port->ip_dl.dl_eth.de_frame= eth_pack;
    r= eth_write(ip_port->ip_dl.dl_eth.de_fd, pack_size);
    if (r == NW_SUSPEND)
    {
        assert(!(ip_port->ip_dl.dl_eth.de_flags &
            IEF_WRITE_SP));
        ip_port->ip_dl.dl_eth.de_flags |= IEF_WRITE_SP;
        return;
    }
    assert(r == NW_OK);
}
}

```

```

PRIVATE void ipeth_arp_reply(ip_port_nr, ipaddr, eth_addr)
int ip_port_nr;

```

```
ipaddr_t ipaddr;
ether_addr_t *eth_addr;
{
    acc_t *prev, *eth_pack;
    int r;
    xmit_hdr_t *xmit_hdr;
    ip_port_t *ip_port;
    time_t t;
    eth_hdr_t *eth_hdr;
    ether_addr_t tmp_eth_addr;

    assert (ip_port_nr >= 0 && ip_port_nr < ip_conf_nr);
    ip_port= &ip_port_table[ip_port_nr];

    for (;;)
    {
        for (prev= 0, eth_pack= ip_port->ip_dl.dl_eth.de_arp_head;
             eth_pack;
             prev= eth_pack, eth_pack= eth_pack->acc_ext_link)
        {
            xmit_hdr= (xmit_hdr_t *)ptr2acc_data(eth_pack);
            if (xmit_hdr->xh_ipaddr == ipaddr)
                break;
        }

        if (eth_pack == NULL)
        {
            /* No packet found. */
            break;
        }

        /* Delete packet from the queue. */
        if (prev == NULL)
        {
            ip_port->ip_dl.dl_eth.de_arp_head=
                eth_pack->acc_ext_link;
        }
        else
        {
            prev->acc_ext_link= eth_pack->acc_ext_link;
            if (prev->acc_ext_link == NULL)
                ip_port->ip_dl.dl_eth.de_arp_tail= prev;
        }

        if (eth_addr == NULL)
        {
            /* Destination is unreachable, delete packet. */
            bf_afree(eth_pack);
            continue;
        }

        /* Fill in the ethernet address and put the packet on the
         * transmit queue.
         */
        t= xmit_hdr->xh_time;
        eth_hdr= (eth_hdr_t *)ptr2acc_data(eth_pack);
        eth_hdr->eh_dst= *eth_addr;
        memcpy(&eth_hdr->eh_src, &t, sizeof(t));

        eth_pack->acc_ext_link= NULL;
        if (ip_port->ip_dl.dl_eth.de_q_head == NULL)
            ip_port->ip_dl.dl_eth.de_q_head= eth_pack;
        else
        {
            ip_port->ip_dl.dl_eth.de_q_tail->acc_ext_link=
                eth_pack;
        }
        ip_port->ip_dl.dl_eth.de_q_tail= eth_pack;
    }

    /* Try to get some more ARPs in progress. */
    while (ip_port->ip_dl.dl_eth.de_arp_head)
    {
        eth_pack= ip_port->ip_dl.dl_eth.de_arp_head;
```

```

xmit_hdr= (xmit_hdr_t *)ptr2acc_data(eth_pack);
r= arp_ip_eth(ip_port->ip_dl.dl_eth.de_port,
              xmit_hdr->xh_ipaddr, &tmp_eth_addr);
if (r == NW_SUSPEND)
    break;                                /* Normal case */

/* Dequeue the packet */
ip_port->ip_dl.dl_eth.de_arp_head= eth_pack->acc_ext_link;

if (r == EDSTNOTRCH)
{
    bf_afree(eth_pack);
    continue;
}
assert(r == NW_OK);

/* Fill in the ethernet address and put the packet on the
 * transmit queue.
 */
t= xmit_hdr->xh_time;
eth_hdr= (eth_hdr_t *)ptr2acc_data(eth_pack);
eth_hdr->eh_dst= tmp_eth_addr;
memcpy(&eth_hdr->eh_src, &t, sizeof(t));

eth_pack->acc_ext_link= NULL;
if (ip_port->ip_dl.dl_eth.de_q_head == NULL)
    ip_port->ip_dl.dl_eth.de_q_head= eth_pack;
else
{
    ip_port->ip_dl.dl_eth.de_q_tail->acc_ext_link=
        eth_pack;
}
ip_port->ip_dl.dl_eth.de_q_tail= eth_pack;
}

/* Restart sending ethernet packets. */
if (ip_port->ip_dl.dl_eth.de_frame == NULL)
    ipeth_restart_send(ip_port);
}

```

```

PRIVATE int ipeth_update_ttl(enq_time, now, eth_pack)
time_t enq_time;
time_t now;
acc_t *eth_pack;
{
    int ttl_diff;
    ip_hdr_t *ip_hdr;
    u32_t sum;
    ul6_t word;
    acc_t *ip_pack;

    ttl_diff= (now-enq_time)/HZ;
    enq_time += ttl_diff*HZ;
    assert(enq_time <= now && enq_time + HZ > now);

    ip_pack= eth_pack->acc_next;
    assert(ip_pack->acc_length >= sizeof(*ip_hdr));
    assert(ip_pack->acc_linkC == 1 &&
           ip_pack->acc_buffer->buf_linkC == 1);

    ip_hdr= (ip_hdr_t *)ptr2acc_data(ip_pack);
    if (ip_hdr->ih_ttl <= ttl_diff)
        return ETIMEDOUT;
    sum= (ul6_t)~ip_hdr->ih_hdr_chk;
    word= *(ul6_t *)&ip_hdr->ih_ttl;
    if (word > sum)
        sum += 0xffff - word;
    else
        sum -= word;
    ip_hdr->ih_ttl -= ttl_diff;
    word= *(ul6_t *)&ip_hdr->ih_ttl;
    sum += word;
    if (sum > 0xffff)
        sum -= 0xffff;
}

```

```
    assert(!(sum & 0xffff0000));
    ip_hdr->ih_hdr_chk= ~sum;
    assert(ip_hdr->ih_ttl > 0);
    return NW_OK;
}

PRIVATE void do_eth_read(ip_port)
ip_port_t *ip_port;
{
    int result;

    assert(!(ip_port->ip_dl.dl_eth.de_flags & IEF_READ_IP));

    for (;;)
    {
        ip_port->ip_dl.dl_eth.de_flags |= IEF_READ_IP;

        result= eth_read (ip_port->ip_dl.dl_eth.de_fd,
                          ETH_MAX_PACK_SIZE);
        if (result == NW_SUSPEND)
        {
            assert(!(ip_port->ip_dl.dl_eth.de_flags &
                    IEF_READ_SP));
            ip_port->ip_dl.dl_eth.de_flags |= IEF_READ_SP;
            return;
        }
        ip_port->ip_dl.dl_eth.de_flags &= ~IEF_READ_IP;
        if (result<0)
        {
            return;
        }
    }
}

PRIVATE void ip_eth_arrived(port, pack, pack_size)
int port;
acc_t *pack;
size_t pack_size;
{
    int broadcast;
    ip_port_t *ip_port;

    ip_port= &ip_port_table[port];
    broadcast= (*(u8_t *)ptr2acc_data(pack) & 1);

    pack= bf_delhead(pack, ETH_HDR_SIZE);

    if (broadcast)
        ip_arrived_broadcast(ip_port, pack);
    else
        ip_arrived(ip_port, pack);
}

/*
 * $PchId: ip_eth.c,v 1.25 2005/06/28 14:18:10 philip Exp $
 */
```

```

/*
ip_int.h

Copyright 1995 Philip Homburg
*/

#ifndef INET_IP_INT_H
#define INET_IP_INT_H

#define IP_FD_NR          (8*IP_PORT_MAX)
#define IP_ASS_NR         3

#define IP_42BSD_BCAST    1          /* hostnumber 0 is also network
                                      broadcast */

#define IP_LT_NORMAL      0          /* Normal */
#define IP_LT_BROADCAST   1          /* Broadcast */
#define IP_LT_MULTICAST   2          /* Multicast */

struct ip_port;
struct ip_fd;
typedef void (*ip_dev_t) ARGS(( struct ip_port *ip_port ));
typedef int (*ip_dev_send_t) ARGS(( struct ip_port *ip_port, ipaddr_t dest,
                                   acc_t *pack, int type ));

#define IP_PROTO_HASH_NR    32

typedef struct ip_port
{
    int ip_flags, ip_dl_type;
    int ip_port;
    union
    {
        struct
        {
            int de_state;
            int de_flags;
            int de_port;
            int de_fd;
            acc_t *de_frame;
            acc_t *de_q_head;
            acc_t *de_q_tail;
            acc_t *de_arp_head;
            acc_t *de_arp_tail;
        } dl_eth;
        struct
        {
            int ps_port;
            acc_t *ps_send_head;
            acc_t *ps_send_tail;
        } dl_ps;
    } ip_dl;
    ipaddr_t ip_ipaddr;
    ipaddr_t ip_subnetmask;
    ipaddr_t ip_classfulmask;
    ul6_t ip_frame_id;
    ul6_t ip_mtu;
    ul6_t ip_mtu_max;          /* Max MTU for this kind of network */
    ip_dev_t ip_dev_main;
    ip_dev_t ip_dev_set_ipaddr;
    ip_dev_send_t ip_dev_send;
    acc_t *ip_loopb_head;
    acc_t *ip_loopb_tail;
    event_t ip_loopb_event;
    acc_t *ip_routeq_head;
    acc_t *ip_routeq_tail;
    event_t ip_routeq_event;
    struct ip_fd *ip_proto_any;
    struct ip_fd *ip_proto[IP_PROTO_HASH_NR];
} ip_port_t;

#define IES_EMPTY          0x0
#define IES_SETPROTO       0x1
#define IES_GETIPADDR      0x2

```



```

#define IES_MAIN          0x3
#define IES_ERROR         0x4

#define IEF_EMPTY         0x1
#define IEF_SUSPEND       0x8
#define IEF_READ_IP       0x10
#define IEF_READ_SP       0x20
#define IEF_WRITE_SP      0x80

#define IPF_EMPTY         0x0
#define IPF_CONFIGURED    0x1
#define IPF_IPADDRSET     0x2
#define IPF_NETMASKSET    0x4
#define IPF_SUBNET_BCAST  0x8    /* Subset support subnet broadcasts */

#define IPDL_ETH          NETTYPE_ETH
#define IPDL_PSIP         NETTYPE_PSIP

typedef struct ip_ass
{
    acc_t *ia_frags;
    int ia_min_ttl;
    ip_port_t *ia_port;
    time_t ia_first_time;
    ipaddr_t ia_srcaddr, ia_dstaddr;
    int ia_proto, ia_id;
} ip_ass_t;

typedef struct ip_fd
{
    int if_flags;
    struct nwio_ipopt if_ipopt;
    ip_port_t *if_port;
    struct ip_fd *if_proto_next;
    int if_srfd;
    acc_t *if_rdbuf_head;
    acc_t *if_rdbuf_tail;
    get_userdata_t if_get_userdata;
    put_userdata_t if_put_userdata;
    put_pkt_t if_put_pkt;
    time_t if_exp_time;
    size_t if_rd_count;
    ioreq_t if_ioctl;
} ip_fd_t;

#define IFF_EMPTY         0x00
#define IFF_INUSE         0x01
#define IFF_OPTSET        0x02
#define IFF_BUSY          0x1C
#    define IFF_READ_IP    0x04
#    define IFF_IOCTL_IP   0x08

typedef enum nettype
{
    IPNT_ZERO,                /* 0.xx.xx.xx */
    IPNT_CLASS_A,             /* 1.xx.xx.xx .. 126.xx.xx.xx */
    IPNT_LOCAL,               /* 127.xx.xx.xx */
    IPNT_CLASS_B,             /* 128.xx.xx.xx .. 191.xx.xx.xx */
    IPNT_CLASS_C,             /* 192.xx.xx.xx .. 223.xx.xx.xx */
    IPNT_CLASS_D,             /* 224.xx.xx.xx .. 239.xx.xx.xx */
    IPNT_CLASS_E,             /* 240.xx.xx.xx .. 247.xx.xx.xx */
    IPNT_MARTIAN,             /* 248.xx.xx.xx .. 254.xx.xx.xx + others */
    IPNT_BROADCAST            /* 255.255.255.255 */
} nettype_t;

struct nwio_ipconf;

/* ip_eth.c */
int ipeth_init ARGS(( ip_port_t *ip_port ));

/* ip_ioctl.c */
void ip_hash_proto ARGS(( ip_fd_t *ip_fd ));
void ip_unhash_proto ARGS(( ip_fd_t *ip_fd ));
int ip_setconf ARGS(( int ip_port, struct nwio_ipconf *ipconfp ));

```

```
/* ip_lib.c */
ipaddr_t ip_get_netmask ARGS(( ipaddr_t hostaddr ));
ipaddr_t ip_get_ifaddr ARGS(( int ip_port_nr ));
int ip_chk_hdropt ARGS(( u8_t *opt, int optlen ));
void ip_print_frags ARGS(( acc_t *acc ));
nettype_t ip_nettype ARGS(( ipaddr_t ipaddr ));
ipaddr_t ip_netmask ARGS(( nettype_t nettype ));
char *ip_nettoa ARGS(( nettype_t nettype ));

/* ip_ps.c */
int ipps_init ARGS(( ip_port_t *ip_port ));
void ipps_get ARGS(( int ip_port_nr ));
void ipps_put ARGS(( int ip_port_nr, ipaddr_t nexthop, acc_t *pack ));

/* ip_read.c */
void ip_port_arrive ARGS(( ip_port_t *port, acc_t *pack, ip_hdr_t *ip_hdr ));
void ip_arrived ARGS(( ip_port_t *port, acc_t *pack ));
void ip_arrived_broadcast ARGS(( ip_port_t *port, acc_t *pack ));
void ip_process_loopb ARGS(( event_t *ev, ev_arg_t arg ));
void ip_packet2user ARGS(( ip_fd_t *ip_fd, acc_t *pack, time_t exp_time,
    size_t data_len ));

/* ip_write.c */
void dll_eth_write_frame ARGS(( ip_port_t *port ));
acc_t *ip_split_pack ARGS(( ip_port_t *ip_port, acc_t **ref_last, int mtu ));
void ip_hdr_chksum ARGS(( ip_hdr_t *ip_hdr, int ip_hdr_len ));

extern ip_fd_t ip_fd_table[IP_FD_NR];
extern ip_port_t *ip_port_table;
extern ip_ass_t ip_ass_table[IP_ASS_NR];

#define NWIO_DEFAULT      (NWIO_EN_LOC | NWIO_EN_BROAD | NWIO_REMANY | \
    NWIO_RWDATALL | NWIO_HDR_O_SPEC)

#endif /* INET_IP_INT_H */

/*
 * $PchId: ip_int.h,v 1.19 2004/08/03 16:24:23 philip Exp $
 */
```

```
/*
ip_ioctl.c

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "buf.h"
#include "event.h"
#include "type.h"

#include "arp.h"
#include "assert.h"
#include "clock.h"
#include "icmp_lib.h"
#include "ip.h"
#include "ip_int.h"
#include "ipr.h"

THIS_FILE

FORWARD int ip_checkopt ARGS(( ip_fd_t *ip_fd ));
FORWARD void reply_thr_get ARGS(( ip_fd_t *ip_fd, size_t
    reply, int for_ioctl ));
FORWARD void report_addr ARGS(( ip_port_t *ip_port ));

PUBLIC int ip_ioctl (fd, req)
int fd;
ioreq_t req;
{
    ip_fd_t *ip_fd;
    ip_port_t *ip_port;
    nwio_ipopt_t *ipopt;
    nwio_ipopt_t oldopt, newopt;
    nwio_ipconf2_t *ipconf2;
    nwio_ipconf_t *ipconf;
    nwio_route_t *route_ent;
    acc_t *data;
    int result;
    unsigned int new_en_flags, new_di_flags,
        old_en_flags, old_di_flags;
    unsigned long new_flags;
    int ent_no, r;
    nwio_ipconf_t ipconf_var;

    assert (fd>=0 && fd<=IP_FD_NR);
    ip_fd= &ip_fd_table[fd];

    assert (ip_fd->if_flags & IFF_INUSE);

    switch (req)
    {
    case NWIOSIPOPT:
        ip_port= ip_fd->if_port;

        if (!(ip_port->ip_flags & IPF_IPADDRSET))
        {
            ip_fd->if_ioctl= NWIOSIPOPT;
            ip_fd->if_flags |= IFF_IOCTL_IP;
            return NW_SUSPEND;
        }
        ip_fd->if_flags &= ~IFF_IOCTL_IP;

        data= (*ip_fd->if_get_userdata)(ip_fd->if_srfd, 0,
            sizeof(nwio_ipopt_t), TRUE);

        data= bf_packIfLess (data, sizeof(nwio_ipopt_t));
        assert (data->acc_length == sizeof(nwio_ipopt_t));

        ipopt= (nwio_ipopt_t *)ptr2acc_data(data);
        oldopt= ip_fd->if_ipopt;
        newopt= *ipopt;

        old_en_flags= oldopt.nwio_flags & 0xffff;
```

```
old_di_flags= (oldopt.nwio_flags >> 16) & 0xffff;
new_en_flags= newopt.nwio_flags & 0xffff;
new_di_flags= (newopt.nwio_flags >> 16) & 0xffff;
if (new_en_flags & new_di_flags)
{
    bf_afree(data);
    reply_thr_get (ip_fd, EBADMODE, TRUE);
    return NW_OK;
}

/* NWIO_ACC_MASK */
if (new_di_flags & NWIO_ACC_MASK)
{
    bf_afree(data);
    reply_thr_get (ip_fd, EBADMODE, TRUE);
    return NW_OK;
    /* access modes can't be disable */
}

if (!(new_en_flags & NWIO_ACC_MASK))
    new_en_flags |= (old_en_flags & NWIO_ACC_MASK);

/* NWIO_LOC_MASK */
if (((new_en_flags|new_di_flags) & NWIO_LOC_MASK))
{
    new_en_flags |= (old_en_flags & NWIO_LOC_MASK);
    new_di_flags |= (old_di_flags & NWIO_LOC_MASK);
}

/* NWIO_BROAD_MASK */
if (((new_en_flags|new_di_flags) & NWIO_BROAD_MASK))
{
    new_en_flags |= (old_en_flags & NWIO_BROAD_MASK);
    new_di_flags |= (old_di_flags & NWIO_BROAD_MASK);
}

/* NWIO_REM_MASK */
if (((new_en_flags|new_di_flags) & NWIO_REM_MASK))
{
    new_en_flags |= (old_en_flags & NWIO_REM_MASK);
    new_di_flags |= (old_di_flags & NWIO_REM_MASK);
    newopt.nwio_rem= oldopt.nwio_rem;
}

/* NWIO_PROTO_MASK */
if (((new_en_flags|new_di_flags) & NWIO_PROTO_MASK))
{
    new_en_flags |= (old_en_flags & NWIO_PROTO_MASK);
    new_di_flags |= (old_di_flags & NWIO_PROTO_MASK);
    newopt.nwio_proto= oldopt.nwio_proto;
}

/* NWIO_HDR_O_MASK */
if (((new_en_flags|new_di_flags) & NWIO_HDR_O_MASK))
{
    new_en_flags |= (old_en_flags & NWIO_HDR_O_MASK);
    new_di_flags |= (old_di_flags & NWIO_HDR_O_MASK);
    newopt.nwio_tos= oldopt.nwio_tos;
    newopt.nwio_ttl= oldopt.nwio_ttl;
    newopt.nwio_df= oldopt.nwio_df;
    newopt.nwio_hdopt= oldopt.nwio_hdopt;
}

/* NWIO_RW_MASK */
if (((new_en_flags|new_di_flags) & NWIO_RW_MASK))
{
    new_en_flags |= (old_en_flags & NWIO_RW_MASK);
    new_di_flags |= (old_di_flags & NWIO_RW_MASK);
}

new_flags= ((unsigned long)new_di_flags << 16) | new_en_flags;

if ((new_flags & NWIO_RWDATONLY) && (new_flags &
    (NWIO_REMANY|NWIO_PROTOANY|NWIO_HDR_O_ANY)))
```

```

    {
        bf_afree(data);
        reply_thr_get(ip_fd, EBADMODE, TRUE);
        return NW_OK;
    }

    if (ip_fd->if_flags & IFF_OPTSET)
        ip_unhash_proto(ip_fd);

    newopt.nwio_flags= new_flags;
    ip_fd->if_ipopt= newopt;

    result= ip_checkopt(ip_fd);

    if (result<0)
        ip_fd->if_ipopt= oldopt;

    bf_afree(data);
    reply_thr_get (ip_fd, result, TRUE);
    return NW_OK;

case NWIOGIPOPT:
    data= bf_memreq(sizeof(nwio_ipopt_t));

    ipopt= (nwio_ipopt_t *)ptr2acc_data(data);

    *ipopt= ip_fd->if_ipopt;

    result= (*ip_fd->if_put_userdata)(ip_fd->if_srfd, 0, data,
                                     TRUE);
    return (*ip_fd->if_put_userdata)(ip_fd->if_srfd, result,
                                     (acc_t *)0, TRUE);

case NWIOSIPCONF2:
case NWIOSIPCONF:
    ip_port= ip_fd->if_port;

    if (req == NWIOSIPCONF2)
    {
        data= (*ip_fd->if_get_userdata)(ip_fd->if_srfd, 0,
                                         sizeof(*ipconf2), TRUE);
        data= bf_packIffLess (data, sizeof(*ipconf2));
        assert (data->acc_length == sizeof(*ipconf2));

        ipconf2= (nwio_ipconf2_t *)ptr2acc_data(data);

        ipconf= &ipconf_var;
        ipconf->nwic_flags= ipconf2->nwic_flags;
        ipconf->nwic_ipaddr= ipconf2->nwic_ipaddr;
        ipconf->nwic_netmask= ipconf2->nwic_netmask;
        ipconf->nwic_flags &= ~NWIC_MTU_SET;
    }
    else
    {
        data= (*ip_fd->if_get_userdata)(ip_fd->if_srfd, 0,
                                         sizeof(*ipconf), TRUE);
        data= bf_packIffLess (data, sizeof(*ipconf));
        assert (data->acc_length == sizeof(*ipconf));

        ipconf= (nwio_ipconf_t *)ptr2acc_data(data);
    }
    r= ip_setconf(ip_port-ip_port_table, ipconf);
    bf_afree(data);
    return (*ip_fd->if_put_userdata)(ip_fd-> if_srfd, r,
                                     (acc_t *)0, TRUE);

case NWIOGIPCONF2:
    ip_port= ip_fd->if_port;

    if (!(ip_port->ip_flags & IPF_IPADDRSET))
    {
        ip_fd->if_ioctl= NWIOGIPCONF2;
        ip_fd->if_flags |= IFF_IOCTL_IP;
        return NW_SUSPEND;
    }

```

```

    }
    ip_fd->if_flags &= ~IFF_IOCTL_IP;
    data= bf_memreq(sizeof(nwio_ipconf_t));
    ipconf2= (nwio_ipconf2_t *)ptr2acc_data(data);
    ipconf2->nwic_flags= NWIC_IPADDR_SET;
    ipconf2->nwic_ipaddr= ip_port->ip_ipaddr;
    ipconf2->nwic_netmask= ip_port->ip_subnetmask;
    if (ip_port->ip_flags & IPF_NETMASKSET)
        ipconf2->nwic_flags |= NWIC_NETMASK_SET;

    result= (*ip_fd->if_put_userdata)(ip_fd->if_srfd, 0, data,
                                     TRUE);
    return (*ip_fd->if_put_userdata)(ip_fd->if_srfd, result,
                                     (acc_t *)0, TRUE);

case NWIOGIPCONF:
    ip_port= ip_fd->if_port;

    if (!(ip_port->ip_flags & IPF_IPADDRSET))
    {
        ip_fd->if_ioctl= NWIOGIPCONF;
        ip_fd->if_flags |= IFF_IOCTL_IP;
        return NW_SUSPEND;
    }
    ip_fd->if_flags &= ~IFF_IOCTL_IP;
    data= bf_memreq(sizeof(*ipconf));
    ipconf= (nwio_ipconf_t *)ptr2acc_data(data);
    ipconf->nwic_flags= NWIC_IPADDR_SET;
    ipconf->nwic_ipaddr= ip_port->ip_ipaddr;
    ipconf->nwic_netmask= ip_port->ip_subnetmask;
    if (ip_port->ip_flags & IPF_NETMASKSET)
        ipconf->nwic_flags |= NWIC_NETMASK_SET;
    ipconf->nwic_mtu= ip_port->ip_mtu;

    result= (*ip_fd->if_put_userdata)(ip_fd->if_srfd, 0, data,
                                     TRUE);
    return (*ip_fd->if_put_userdata)(ip_fd->if_srfd, result,
                                     (acc_t *)0, TRUE);

case NWIOGIPORROUTE:
    data= (*ip_fd->if_get_userdata)(ip_fd->if_srfd,
                                    0, sizeof(nwio_route_t), TRUE);
    if (data == NULL)
    {
        return (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
                                         EFAULT, NULL, TRUE);
    }

    data= bf_packIfLess (data, sizeof(nwio_route_t) );
    route_ent= (nwio_route_t *)ptr2acc_data(data);
    ent_no= route_ent->nwr_ent_no;
    bf_afree(data);

    data= bf_memreq(sizeof(nwio_route_t));
    route_ent= (nwio_route_t *)ptr2acc_data(data);
    result= ipr_get_oroute(ent_no, route_ent);
    if (result < 0)
        bf_afree(data);
    else
    {
        assert(result == NW_OK);
        result= (*ip_fd->if_put_userdata)(ip_fd->if_srfd, 0,
                                         data, TRUE);
    }
    return (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
                                     result, (acc_t *)0, TRUE);

case NWIOSIPORROUTE:
    data= (*ip_fd->if_get_userdata)(ip_fd->if_srfd,
                                    0, sizeof(nwio_route_t), TRUE);
    if (data == NULL)
    {
        return (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
                                         EFAULT, NULL, TRUE);
    }

```

```
}
if (!(ip_fd->if_port->ip_flags & IPF_IPADDRSET))
{
    /* Interface is down, no changes allowed */
    return (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
        EINVAL, NULL, TRUE);
}

data= bf_packIfLess (data, sizeof(nwio_route_t) );
route_ent= (nwio_route_t *)ptr2acc_data(data);
result= ipr_add_oroute(ip_fd->if_port-ip_port_table,
    route_ent->nwr_dest, route_ent->nwr_netmask,
    route_ent->nwr_gateway, (time_t)0,
    route_ent->nwr_dist, route_ent->nwr_mtu,
    !(route_ent->nwr_flags & NWR_STATIC),
    route_ent->nwr_pref, NULL);
bf_afree(data);

return (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
    result, (acc_t *)0, TRUE);

case NWIODIPOROUTE:
    data= (*ip_fd->if_get_userdata)(ip_fd->if_srfd,
        0, sizeof(nwio_route_t), TRUE);
    if (data == NULL)
    {
        return (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
            EFAULT, NULL, TRUE);
    }

    data= bf_packIfLess (data, sizeof(nwio_route_t) );
    route_ent= (nwio_route_t *)ptr2acc_data(data);
    result= ipr_del_oroute(ip_fd->if_port-ip_port_table,
        route_ent->nwr_dest, route_ent->nwr_netmask,
        route_ent->nwr_gateway,
        !(route_ent->nwr_flags & NWR_STATIC));
    bf_afree(data);

    return (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
        result, (acc_t *)0, TRUE);

case NWIOGIPIROUTE:
    data= (*ip_fd->if_get_userdata)(ip_fd->if_srfd,
        0, sizeof(nwio_route_t), TRUE);
    if (data == NULL)
    {
        return (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
            EFAULT, NULL, TRUE);
    }

    data= bf_packIfLess (data, sizeof(nwio_route_t) );
    route_ent= (nwio_route_t *)ptr2acc_data(data);
    ent_no= route_ent->nwr_ent_no;
    bf_afree(data);

    data= bf_memreq(sizeof(nwio_route_t));
    route_ent= (nwio_route_t *)ptr2acc_data(data);
    result= ipr_get_iroute(ent_no, route_ent);
    if (result < 0)
        bf_afree(data);
    else
    {
        assert(result == NW_OK);
        result= (*ip_fd->if_put_userdata)(ip_fd->if_srfd, 0,
            data, TRUE);
    }
    return (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
        result, (acc_t *)0, TRUE);

case NWIOSIPIROUTE:
    data= (*ip_fd->if_get_userdata)(ip_fd->if_srfd,
        0, sizeof(nwio_route_t), TRUE);
    if (data == NULL)
    {
```

```

        return (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
            EFAULT, NULL, TRUE);
    }
    if (!(ip_fd->if_port->ip_flags & IPF_IPADDRSET))
    {
        /* Interface is down, no changes allowed */
        return (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
            EINVAL, NULL, TRUE);
    }

    data= bf_packIfLess (data, sizeof(nwio_route_t) );
    route_ent= (nwio_route_t *)ptr2acc_data(data);
    result= ipr_add_iroute(ip_fd->if_port-ip_port_table,
        route_ent->nwr_dest, route_ent->nwr_netmask,
        route_ent->nwr_gateway,
        (route_ent->nwr_flags & NWRF_UNREACHABLE) ?
            IRTD_UNREACHABLE : route_ent->nwr_dist,
        route_ent->nwr_mtu,
        !(route_ent->nwr_flags & NWRF_STATIC), NULL);
    bf_afree(data);

    return (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
        result, (acc_t *)0, TRUE);

case NWIODIPIROUTE:
    data= (*ip_fd->if_get_userdata)(ip_fd->if_srfd,
        0, sizeof(nwio_route_t), TRUE);
    if (data == NULL)
    {
        return (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
            EFAULT, NULL, TRUE);
    }

    data= bf_packIfLess (data, sizeof(nwio_route_t) );
    route_ent= (nwio_route_t *)ptr2acc_data(data);
    result= ipr_del_iroute(ip_fd->if_port-ip_port_table,
        route_ent->nwr_dest, route_ent->nwr_netmask,
        route_ent->nwr_gateway,
        !(route_ent->nwr_flags & NWRF_STATIC));
    bf_afree(data);

    return (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
        result, (acc_t *)0, TRUE);

    /* The following ARP ioctls are only valid if the
     * underlying device is an ethernet.
     */
case NWIOARPGIP:
case NWIOARPGNEXT:
case NWIOARPSIP:
case NWIOARPDIP:
    ip_port= ip_fd->if_port;

    if (ip_port->ip_dl_type != IPDL_ETH)
    {
        return (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
            EBADIOCTL, (acc_t *)0, TRUE);
    }
    result= arp_ioctl(ip_port->ip_dl.eth.de_port,
        ip_fd->if_srfd, req, ip_fd->if_get_userdata,
        ip_fd->if_put_userdata);
    assert (result != SUSPEND);
    return (*ip_fd->if_put_userdata)(ip_fd->if_srfd, result,
        (acc_t *)0, TRUE);

default:
    break;
}
DBLOCK(1, printf("replying EBADIOCTL: 0x%x\n", req));
return (*ip_fd->if_put_userdata)(ip_fd->if_srfd, EBADIOCTL,
    (acc_t *)0, TRUE);
}

```

```
PUBLIC void ip_hash_proto(ip_fd)
```



```
ip_fd_t *ip_fd;
{
    ip_port_t *ip_port;
    int hash;

    ip_port= ip_fd->if_port;
    if (ip_fd->if_ipopt.nwio_flags & NWIO_PROTOANY)
    {
        ip_fd->if_proto_next= ip_port->ip_proto_any;
        ip_port->ip_proto_any= ip_fd;
    }
    else
    {
        hash= ip_fd->if_ipopt.nwio_proto & (IP_PROTO_HASH_NR-1);
        ip_fd->if_proto_next= ip_port->ip_proto[hash];
        ip_port->ip_proto[hash]= ip_fd;
    }
}

PUBLIC void ip_unhash_proto(ip_fd)
ip_fd_t *ip_fd;
{
    ip_port_t *ip_port;
    ip_fd_t *prev, *curr, **ip_fd_p;
    int hash;

    ip_port= ip_fd->if_port;
    if (ip_fd->if_ipopt.nwio_flags & NWIO_PROTOANY)
    {
        ip_fd_p= &ip_port->ip_proto_any;
    }
    else
    {
        hash= ip_fd->if_ipopt.nwio_proto & (IP_PROTO_HASH_NR-1);
        ip_fd_p= &ip_port->ip_proto[hash];
    }
    for (prev= NULL, curr= *ip_fd_p; curr;
        prev= curr, curr= curr->if_proto_next)
    {
        if (curr == ip_fd)
            break;
    }
    assert(curr);
    if (prev)
        prev->if_proto_next= curr->if_proto_next;
    else
        *ip_fd_p= curr->if_proto_next;
}

PUBLIC int ip_setconf(ip_port_nr, ipconf)
int ip_port_nr;
nwio_ipconf_t *ipconf;
{
    int i, old_ip_flags, do_report;
    ip_port_t *ip_port;
    ip_fd_t *ip_fd;
    ipaddr_t ipaddr;
    u32_t mtu;

    ip_port= &ip_port_table[ip_port_nr];

    old_ip_flags= ip_port->ip_flags;

    if (ipconf->nwic_flags & ~NWIC_FLAGS)
        return EBADMODE;

    do_report= 0;
    if (ipconf->nwic_flags & NWIC_MTU_SET)
    {
        mtu= ipconf->nwic_mtu;
        if (mtu < IP_MIN_MTU || mtu > ip_port->ip_mtu_max)
            return EINVAL;
        ip_port->ip_mtu= mtu;
        do_report= 1;
    }
}
```

```

    }

    if (ipconf->nwic_flags & NWIC_NETMASK_SET)
    {
        ip_port->ip_subnetmask= ipconf->nwic_netmask;
        ip_port->ip_flags |= IPF_NETMASKSET|IPF_SUBNET_BCAST;
        if (ntohl(ip_port->ip_subnetmask) >= 0xfffffffffe)
            ip_port->ip_flags &= ~IPF_SUBNET_BCAST;
        do_report= 1;
    }

    if (ipconf->nwic_flags & NWIC_IPADDR_SET)
    {
        ipaddr= ipconf->nwic_ipaddr;
        ip_port->ip_ipaddr= ipaddr;
        ip_port->ip_flags |= IPF_IPADDRSET;
        ip_port->ip_classfulmask=
            ip_netmask(ip_nettype(ipaddr));
        if (!(ip_port->ip_flags & IPF_NETMASKSET))
        {
            ip_port->ip_subnetmask= ip_port->ip_classfulmask;
        }
        if (ipaddr == HTONL(0x00000000))
        {
            /* Special case. Use 0.0.0.0 to shutdown interface. */
            ip_port->ip_flags &= ~(IPF_IPADDRSET|IPF_NETMASKSET);
            ip_port->ip_subnetmask= HTONL(0x00000000);
        }
        (*ip_port->ip_dev_set_ipaddr)(ip_port);

        /* revive calls waiting for an ip addresses */
        for (i=0, ip_fd= ip_fd_table; i<IP_FD_NR; i++, ip_fd++)
        {
            if (!(ip_fd->if_flags & IFF_INUSE))
                continue;
            if (ip_fd->if_port != ip_port)
                continue;
            if (ip_fd->if_flags & IFF_IOCTL_IP)
                ip_ioctl (i, ip_fd->if_ioctl);
        }

        do_report= 1;
    }

    ipr_chk_itab(ip_port-ip_port_table, ip_port->ip_ipaddr,
        ip_port->ip_subnetmask);
    ipr_chk_otab(ip_port-ip_port_table, ip_port->ip_ipaddr,
        ip_port->ip_subnetmask);
    if (do_report)
        report_addr(ip_port);

    return 0;
}

PRIVATE int ip_checkopt (ip_fd)
ip_fd_t *ip_fd;
{
    /* bug: we don't check access modes yet */

    unsigned long flags;
    unsigned int en_di_flags;
    acc_t *pack;
    int result;

    flags= ip_fd->if_ipopt.nwio_flags;
    en_di_flags= (flags >>16) | (flags & 0xffff);

    if (flags & NWIO_HDR_O_SPEC)
    {
        result= ip_chk_hdopt (ip_fd->if_ipopt.nwio_hdopt.iho_data,
            ip_fd->if_ipopt.nwio_hdopt.iho_opt_siz);
        if (result<0)
            return result;
    }

    if ((en_di_flags & NWIO_ACC_MASK) &&

```

```

        (en_di_flags & NWIO_LOC_MASK) &&
        (en_di_flags & NWIO_BROAD_MASK) &&
        (en_di_flags & NWIO_REM_MASK) &&
        (en_di_flags & NWIO_PROTO_MASK) &&
        (en_di_flags & NWIO_HDR_O_MASK) &&
        (en_di_flags & NWIO_RW_MASK))
    {
        ip_fd->if_flags |= IFF_OPTSET;

        ip_hash_proto(ip_fd);
    }

    else
        ip_fd->if_flags &= ~IFF_OPTSET;

    while (ip_fd->if_rdbuf_head)
    {
        pack= ip_fd->if_rdbuf_head;
        ip_fd->if_rdbuf_head= pack->acc_ext_link;
        bf_afree(pack);
    }
    return NW_OK;
}

PRIVATE void reply_thr_get(ip_fd, reply, for_ioctl)
ip_fd_t *ip_fd;
size_t reply;
int for_ioctl;
{
    acc_t *result;
    result= (ip_fd->if_get_userdata)(ip_fd->if_srfd, reply,
        (size_t)0, for_ioctl);
    assert (!result);
}

PRIVATE void report_addr(ip_port)
ip_port_t *ip_port;
{
    int i, hdr_len;
    ip_fd_t *ip_fd;
    acc_t *pack;
    ip_hdr_t *ip_hdr;

    pack= bf_memreq(IP_MIN_HDR_SIZE);
    ip_hdr= (ip_hdr_t *)ptr2acc_data(pack);

    hdr_len= IP_MIN_HDR_SIZE;
    ip_hdr->ih_vers_ihl= (IP_VERSION << 4) | (hdr_len/4);
    ip_hdr->ih_tos= 0;
    ip_hdr->ih_length= htons(ip_port->ip_mtu);
    ip_hdr->ih_id= 0;
    ip_hdr->ih_flags_fragoff= 0;
    ip_hdr->ih_ttl= 0;
    ip_hdr->ih_proto= 0;
    ip_hdr->ih_src= ip_port->ip_ipaddr;
    ip_hdr->ih_dst= ip_port->ip_subnetmask;
    ip_hdr_chksum(ip_hdr, hdr_len);

    for (i=0, ip_fd= ip_fd_table; i<IP_FD_NR; i++, ip_fd++)
    {
        if (!(ip_fd->if_flags & IFF_INUSE))
        {
            continue;
        }
        if (ip_fd->if_port != ip_port)
        {
            continue;
        }

        /* Deliver packet to user */
        pack->acc_linkC++;
        ip_packet2user(ip_fd, pack, 255, IP_MIN_HDR_SIZE);
    }
    bf_afree(pack); pack= NULL;
}

```

```
}  
/*  
 * $PchId: ip_ioctl.c,v 1.22 2004/08/03 11:10:08 philip Exp $  
 */
```

```
/*
ip_lib.c

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "buf.h"
#include "event.h"
#include "type.h"

#include "assert.h"
#include "io.h"
#include "ip_int.h"

THIS_FILE

PUBLIC ipaddr_t ip_get_netmask (hostaddr)
ipaddr_t hostaddr;
{
    return ip_netmask(ip_nettype(hostaddr));
}

PUBLIC int ip_chk_hdropt (opt, optlen)
u8_t *opt;
int optlen;
{
    int i, security_present= FALSE, lose_source_present= FALSE,
        strict_source_present= FALSE, record_route_present= FALSE,
        timestamp_present= FALSE;

    assert (!(optlen & 3));
    i= 0;
    while (i<optlen)
    {
        DBLOCK(2, printf("opt= %d\n", *opt));

        switch (*opt)
        {
        case IP_OPT_EOL:          /* End of Option list */
            return NW_OK;
        case IP_OPT_NOP:         /* No Operation */
            i++;
            opt++;
            break;
        case IP_OPT_SEC:         /* Security */
            if (security_present)
                return EINVAL;
            security_present= TRUE;
            if (opt[1] != 11)
                return EINVAL;
            i += opt[1];
            opt += opt[1];
            break;
        case IP_OPT_LSRR:        /* Lose Source and Record Route */
            if (lose_source_present)
            {
                DBLOCK(1, printf("2nd lose source route\n"));
                return EINVAL;
            }
            lose_source_present= TRUE;
            if (opt[1]<3)
            {
                DBLOCK(1,
                    printf("wrong length in source route\n"));
                return EINVAL;
            }
            i += opt[1];
            opt += opt[1];
            break;
        case IP_OPT_SSRR:        /* Strict Source and Record Route */
            if (strict_source_present)
                return EINVAL;
            strict_source_present= TRUE;
        }
    }
}
```

```

        if (opt[1]<3)
            return EINVAL;
        i += opt[1];
        opt += opt[1];
        break;
    case IP_OPT_RR: /* Record Route */
        if (record_route_present)
            return EINVAL;
        record_route_present= TRUE;
        if (opt[1]<3)
            return EINVAL;
        i += opt[1];
        opt += opt[1];
        break;
    case IP_OPT_TS: /* Timestamp */
        if (timestamp_present)
            return EINVAL;
        timestamp_present= TRUE;
        if (opt[1] != 4)
            return EINVAL;
        switch (opt[3] & 0xff)
        {
            case 0:
            case 1:
            case 3:
                break;
            default:
                return EINVAL;
        }
        i += opt[1];
        opt += opt[1];
        break;
    case IP_OPT_RTRALT:
        if (opt[1] != 4)
            return EINVAL;
        i += opt[1];
        opt += opt[1];
        break;
    default:
        return EINVAL;
}
}
if (i > optlen)
{
    DBLOCK(1, printf("option of wrong length\n"));
    return EINVAL;
}
return NW_OK;
}

PUBLIC void ip_print_frags(acc)
acc_t *acc;
{
    #if DEBUG
        ip_hdr_t *ip_hdr;
        int first;

        if (!acc)
            printf("(null)");

        for (first= 1; acc; acc= acc->acc_ext_link, first= 0)
        {
            assert (acc->acc_length >= IP_MIN_HDR_SIZE);
            ip_hdr= (ip_hdr_t *)ptr2acc_data(acc);
            if (first)
            {
                writeIpAddr(ip_hdr->ih_src);
                printf(">");
                writeIpAddr(ip_hdr->ih_dst);
            }
            printf(" { %x:%d@%d%c", ntohs(ip_hdr->ih_id),
                ntohs(ip_hdr->ih_length),
                (ntohs(ip_hdr->ih_flags_fragoff) & IH_FRAGOFF_MASK)*8,
                (ntohs(ip_hdr->ih_flags_fragoff) & IH_MORE_FRAGS) ?

```

```
        '+' : '\0');
    }
#endif
}

PUBLIC ipaddr_t ip_get_ifaddr(port_nr)
int port_nr;
{
    assert(port_nr >= 0 && port_nr < ip_conf_nr);

    return ip_port_table[port_nr].ip_ipaddr;
}

PUBLIC nettype_t ip_nettype(ipaddr)
ipaddr_t ipaddr;
{
    u8_t highbyte;
    nettype_t nettype;

    ipaddr= ntohl(ipaddr);
    highbyte= (ipaddr >> 24) & 0xff;
    if (highbyte == 0)
    {
        if (ipaddr == 0)
            nettype= IPNT_ZERO;
        else
            nettype= IPNT_MARTIAN;
    }
    else if (highbyte < 127)
        nettype= IPNT_CLASS_A;
    else if (highbyte == 127)
        nettype= IPNT_LOCAL;
    else if (highbyte < 192)
        nettype= IPNT_CLASS_B;
    else if (highbyte < 224)
        nettype= IPNT_CLASS_C;
    else if (highbyte < 240)
        nettype= IPNT_CLASS_D;
    else if (highbyte < 248)
        nettype= IPNT_CLASS_E;
    else if (highbyte < 255)
        nettype= IPNT_MARTIAN;
    else
    {
        if (ipaddr == (ipaddr_t)-1)
            nettype= IPNT_BROADCAST;
        else
            nettype= IPNT_MARTIAN;
    }
    return nettype;
}

PUBLIC ipaddr_t ip_netmask(nettype)
nettype_t nettype;
{
    switch(nettype)
    {
        case IPNT_ZERO:          return HTONL(0x00000000);
        case IPNT_CLASS_A:      return HTONL(0xff000000);
        case IPNT_CLASS_B:      return HTONL(0xffff0000);
        case IPNT_CLASS_C:      return HTONL(0xffffffff00);
        default:                 return HTONL(0xffffffff);
    }
}

#if 0
PUBLIC char *ip_nettoa(nettype)
nettype_t nettype;
{
    switch(nettype)
    {
        case IPNT_ZERO:          return "zero";
        case IPNT_CLASS_A:      return "class A";
    }
}
```

```
    case IPNT_LOCAL:      return "local" ;
    case IPNT_CLASS_B:    return "class B" ;
    case IPNT_CLASS_C:    return "class C" ;
    case IPNT_CLASS_D:    return "class D" ;
    case IPNT_CLASS_E:    return "class E" ;
    case IPNT_MARTIAN:    return "martian" ;
    case IPNT_BROADCAST:  return "broadcast" ;
    default:              return "<unknown>" ;
}
#endif

/*
 * $PchId: ip_lib.c,v 1.10 2002/06/08 21:35:52 philip Exp $
 */
```



```

/*
generic/ip_ps.c

pseudo IP specific part of the IP implementation

Created:      Apr 23, 1993 by Philip Homburg

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "assert.h"
#include "type.h"
#include "buf.h"
#include "event.h"
#include "ip.h"
#include "ip_int.h"
#include "psip.h"

THIS_FILE

FORWARD void ipps_main ARGS(( ip_port_t *ip_port ));
FORWARD void ipps_set_ipaddr ARGS(( ip_port_t *ip_port ));
FORWARD int ipps_send ARGS(( struct ip_port *ip_port, ipaddr_t dest,
                           acc_t *pack, int type ));

PUBLIC int ipps_init(ip_port)
ip_port_t *ip_port;
{
    int result;

    result= psip_enable(ip_port->ip_dl.dl_ps.ps_port, ip_port->ip_port);
    if (result == -1)
        return -1;
    ip_port->ip_dl.dl_ps.ps_send_head= NULL;
    ip_port->ip_dl.dl_ps.ps_send_tail= NULL;
    ip_port->ip_dev_main= ipps_main;
    ip_port->ip_dev_set_ipaddr= ipps_set_ipaddr;
    ip_port->ip_dev_send= ipps_send;
    return result;
}

PUBLIC void ipps_get(ip_port_nr)
int ip_port_nr;
{
    int result;
    ipaddr_t dest;
    acc_t *acc, *pack, *next_part;
    ip_port_t *ip_port;

    assert(ip_port_nr >= 0 && ip_port_nr < ip_conf_nr);
    ip_port= &ip_port_table[ip_port_nr];
    assert(ip_port->ip_dl_type == IPDL_PSIP);

    while (ip_port->ip_dl.dl_ps.ps_send_head != NULL)
    {
        pack= ip_port->ip_dl.dl_ps.ps_send_head;
        ip_port->ip_dl.dl_ps.ps_send_head= pack->acc_ext_link;

        /* Extract nexthop address */
        pack= bf_packIffLess(pack, sizeof(dest));
        dest= *(ipaddr_t *)ptr2acc_data(pack);
        pack= bf_delhead(pack, sizeof(dest));

        if (bf_bufsize(pack) > ip_port->ip_mtu)
        {
            next_part= pack;
            pack= ip_split_pack(ip_port, &next_part,
                               ip_port->ip_mtu);
            if (pack == NULL)
                continue;

            /* Prepend nexthop address */
            acc= bf_memreq(sizeof(dest));

```

```

        *(ipaddr_t *) (ptr2acc_data(acc)) = dest;
        acc->acc_next = next_part;
        next_part = acc; acc = NULL;

        assert(next_part->acc_linkC == 1);
        next_part->acc_ext_link = NULL;
        if (ip_port->ip_dl.dl_ps.ps_send_head)
        {
            ip_port->ip_dl.dl_ps.ps_send_tail->
                acc_ext_link = next_part;
        }
        else
        {
            ip_port->ip_dl.dl_ps.ps_send_head =
                next_part;
        }
        ip_port->ip_dl.dl_ps.ps_send_tail = next_part;
    }

    result = psip_send(ip_port->ip_dl.dl_ps.ps_port, dest, pack);
    if (result != NW_SUSPEND)
    {
        assert(result == NW_OK);
        continue;
    }

    /* Prepend nexthop address */
    acc = bf_memreq(sizeof(dest));
    *(ipaddr_t *) (ptr2acc_data(acc)) = dest;
    acc->acc_next = pack;
    pack = acc; acc = NULL;

    pack->acc_ext_link = ip_port->ip_dl.dl_ps.ps_send_head;
    ip_port->ip_dl.dl_ps.ps_send_head = pack;
    if (pack->acc_ext_link == NULL)
        ip_port->ip_dl.dl_ps.ps_send_tail = pack;
    break;
}

}

PUBLIC void ipps_put(ip_port_nr, nexthop, pack)
int ip_port_nr;
ipaddr_t nexthop;
acc_t *pack;
{
    ip_port_t *ip_port;

    assert(ip_port_nr >= 0 && ip_port_nr < ip_conf_nr);
    ip_port = &ip_port_table[ip_port_nr];
    assert(ip_port->ip_dl_type == IPDL_PSIP);
    if (nexthop == HTONL(0xffffffff))
        ip_arrived_broadcast(ip_port, pack);
    else
        ip_arrived(ip_port, pack);
}

PRIVATE void ipps_main(ip_port)
ip_port_t *ip_port;
{
    /* nothing to do */
}

PRIVATE void ipps_set_ipaddr(ip_port)
ip_port_t *ip_port;
{
}

PRIVATE int ipps_send(ip_port, dest, pack, type)
struct ip_port *ip_port;
ipaddr_t dest;
acc_t *pack;
int type;
{
    int result;

```

```
acc_t *acc, *next_part;

if (type != IP_LT_NORMAL)
{
    ip_arrived_broadcast(ip_port, bf_dupacc(pack));

    /* Map all broadcasts to the on-link broadcast address.
     * This saves the application from having to find out
     * if the destination is a subnet broadcast.
     */
    dest= HTONL(0xffffffff);
}

/* Note that allocating a packet may trigger a cleanup action,
 * which may cause the send queue to become empty.
 */
while (ip_port->ip_dl.dl_ps.ps_send_head != NULL)
{
    acc= bf_memreq(sizeof(dest));

    if (ip_port->ip_dl.dl_ps.ps_send_head == NULL)
    {
        bf_afree(acc); acc= NULL;
        continue;
    }

    /* Prepend nexthop address */
    *(ipaddr_t *) (ptr2acc_data(acc))= dest;
    acc->acc_next= pack;
    pack= acc; acc= NULL;

    assert(pack->acc_linkC == 1);
    pack->acc_ext_link= NULL;

    ip_port->ip_dl.dl_ps.ps_send_tail->acc_ext_link= pack;
    ip_port->ip_dl.dl_ps.ps_send_tail= pack;

    return NW_OK;
}

while (pack)
{
    if (bf_bufsize(pack) > ip_port->ip_mtu)
    {
        next_part= pack;
        pack= ip_split_pack(ip_port, &next_part,
            ip_port->ip_mtu);
        if (pack == NULL)
        {
            return NW_OK;
        }

        /* Prepend nexthop address */
        acc= bf_memreq(sizeof(dest));
        *(ipaddr_t *) (ptr2acc_data(acc))= dest;
        acc->acc_next= next_part;
        next_part= acc; acc= NULL;

        assert(next_part->acc_linkC == 1);
        next_part->acc_ext_link= NULL;
        ip_port->ip_dl.dl_ps.ps_send_head= next_part;
        ip_port->ip_dl.dl_ps.ps_send_tail= next_part;
    }
    result= psip_send(ip_port->ip_dl.dl_ps.ps_port, dest, pack);
    if (result == NW_SUSPEND)
    {
        /* Prepend nexthop address */
        acc= bf_memreq(sizeof(dest));
        *(ipaddr_t *) (ptr2acc_data(acc))= dest;
        acc->acc_next= pack;
        pack= acc; acc= NULL;

        assert(pack->acc_linkC == 1);
        pack->acc_ext_link= ip_port->ip_dl.dl_ps.ps_send_head;
    }
}
```

```
        ip_port->ip_dl.dl_ps.ps_send_head= pack;
        if (!pack->acc_ext_link)
            ip_port->ip_dl.dl_ps.ps_send_tail= pack;
        break;
    }
    assert(result == NW_OK);
    pack= ip_port->ip_dl.dl_ps.ps_send_head;
    if (!pack)
        break;
    ip_port->ip_dl.dl_ps.ps_send_head= pack->acc_ext_link;

    /* Extract nexthop address */
    pack= bf_packIffLess(pack, sizeof(dest));
    dest= *(ipaddr_t *)ptr2acc_data(pack);
    pack= bf_delhead(pack, sizeof(dest));
}

return NW_OK;
}

#if 0
int ipps_check(ip_port_t *ip_port)
{
    int n, bad;
    acc_t *prev, *curr;

    for (n= 0, prev= NULL, curr= ip_port->ip_dl.dl_ps.ps_send_head;
         curr; prev= curr, curr= curr->acc_ext_link)
    {
        n++;
    }
    bad= 0;
    if (prev != NULL && prev != ip_port->ip_dl.dl_ps.ps_send_tail_)
    {
        printf("ipps_check, ip[%d]: wrong tail: got %p, expected %p\n",
               ip_port-ip_port_table,
               ip_port->ip_dl.dl_ps.ps_send_tail_, prev);
        bad++;
    }
    if (n != ip_port->ip_dl.dl_ps.ps_send_nr)
    {
        printf("ipps_check, ip[%d]: wrong count: got %d, expected %d\n",
               ip_port-ip_port_table,
               ip_port->ip_dl.dl_ps.ps_send_nr, n);
        bad++;
    }
    return bad == 0;
}
#endif

/*
 * $PchId: ip_ps.c,v 1.15 2003/01/21 15:57:52 philip Exp $
 */
```

```
/*
ip_read.c

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "buf.h"
#include "clock.h"
#include "event.h"
#include "type.h"

#include "assert.h"
#include "icmp_lib.h"
#include "io.h"
#include "ip.h"
#include "ip_int.h"
#include "ipr.h"

THIS_FILE

FORWARD ip_ass_t *find_ass_ent ARGS(( ip_port_t *ip_port, U16_t id,
    int proto, ipaddr_t src, ipaddr_t dst ));
FORWARD acc_t *merge_frags ARGS(( acc_t *first, acc_t *second ));
FORWARD int ip_frag_chk ARGS(( acc_t *pack ));
FORWARD acc_t *reassemble ARGS(( ip_port_t *ip_port, acc_t *pack,
    ip_hdr_t *ip_hdr ));
FORWARD void route_packets ARGS(( event_t *ev, ev_arg_t ev_arg ));
FORWARD int broadcast_dst ARGS(( ip_port_t *ip_port, ipaddr_t dest ));

PUBLIC int ip_read (fd, count)
int fd;
size_t count;
{
    ip_fd_t *ip_fd;
    acc_t *pack;

    ip_fd= &ip_fd_table[fd];
    if (!(ip_fd->if_flags & IFF_OPTSET))
    {
        return (ip_fd->if_put_userdata)(ip_fd->if_srfd, EBADMODE,
            (acc_t *)0, FALSE);
    }

    ip_fd->if_rd_count= count;

    ip_fd->if_flags |= IFF_READ_IP;
    if (ip_fd->if_rdbuf_head)
    {
        if (get_time() <= ip_fd->if_exp_time)
        {
            pack= ip_fd->if_rdbuf_head;
            ip_fd->if_rdbuf_head= pack->acc_ext_link;
            ip_packet2user (ip_fd, pack, ip_fd->if_exp_time,
                bf_bufsize(pack));
            assert(!(ip_fd->if_flags & IFF_READ_IP));
            return NW_OK;
        }
        while (ip_fd->if_rdbuf_head)
        {
            pack= ip_fd->if_rdbuf_head;
            ip_fd->if_rdbuf_head= pack->acc_ext_link;
            bf_afree(pack);
        }
    }
    return NW_SUSPEND;
}

PRIVATE acc_t *reassemble (ip_port, pack, pack_hdr)
ip_port_t *ip_port;
acc_t *pack;
ip_hdr_t *pack_hdr;
{
    ip_ass_t *ass_ent;
```

```

size_t pack_hdr_len, pack_data_len, pack_offset, tmp_offset;
ul6_t pack_flags_fragoff;
acc_t *prev_acc, *curr_acc, *next_acc, *head_acc, *tmp_acc;
ip_hdr_t *tmp_hdr;
time_t first_time;

ass_ent= find_ass_ent (ip_port, pack_hdr->ih_id,
                      pack_hdr->ih_proto, pack_hdr->ih_src, pack_hdr->ih_dst);

pack_flags_fragoff= ntohs(pack_hdr->ih_flags_fragoff);
pack_hdr_len= (pack_hdr->ih_vers_ihl & IH_IHL_MASK) * 4;
pack_data_len= ntohs(pack_hdr->ih_length)-pack_hdr_len;
pack_offset= (pack_flags_fragoff & IH_FRAGOFF_MASK)*8;
pack->acc_ext_link= NULL;

head_acc= ass_ent->ia_frags;
ass_ent->ia_frags= NULL;
if (head_acc == NULL)
{
    ass_ent->ia_frags= pack;
    return NULL;
}

prev_acc= NULL;
curr_acc= NULL;
next_acc= head_acc;

while(next_acc)
{
    tmp_hdr= (ip_hdr_t *)ptr2acc_data(next_acc);
    tmp_offset= (ntohs(tmp_hdr->ih_flags_fragoff) &
                IH_FRAGOFF_MASK)*8;

    if (pack_offset < tmp_offset)
        break;

    prev_acc= curr_acc;
    curr_acc= next_acc;
    next_acc= next_acc->acc_ext_link;
}
if (curr_acc == NULL)
{
    assert(prev_acc == NULL);
    assert(next_acc != NULL);

    curr_acc= merge_frags(pack, next_acc);
    head_acc= curr_acc;
}
else
{
    curr_acc= merge_frags(curr_acc, pack);
    if (next_acc != NULL)
        curr_acc= merge_frags(curr_acc, next_acc);
    if (prev_acc != NULL)
        prev_acc->acc_ext_link= curr_acc;
    else
        head_acc= curr_acc;
}
ass_ent->ia_frags= head_acc;

pack= ass_ent->ia_frags;
pack_hdr= (ip_hdr_t *)ptr2acc_data(pack);
pack_flags_fragoff= ntohs(pack_hdr->ih_flags_fragoff);

if (!(pack_flags_fragoff & (IH_FRAGOFF_MASK|IH_MORE_FRAGS)))
    /* it's now a complete packet */
    {
        first_time= ass_ent->ia_first_time;

        ass_ent->ia_frags= 0;
        ass_ent->ia_first_time= 0;

        while (pack->acc_ext_link)
            {

```

```

        tmp_acc= pack->acc_ext_link;
        pack->acc_ext_link= tmp_acc->acc_ext_link;
        bf_afree(tmp_acc);
    }
    if ((ass_ent->ia_min_ttl) * HZ + first_time <
        get_time())
    {
        if (broadcast_dst(ip_port, pack_hdr->ih_dst))
        {
            DBLOCK(1, printf(
"ip_read'reassemble: reassembly timeout for broadcast packet\n"));
            bf_afree(pack); pack= NULL;
            return NULL;
        }
        icmp_snd_time_exceeded(ip_port->ip_port, pack,
            ICMP_FRAG_REASSEM);
    }
    else
        return pack;
}
return NULL;
}

PRIVATE acc_t *merge_frags (first, second)
acc_t *first, *second;
{
    ip_hdr_t *first_hdr, *second_hdr;
    size_t first_hdr_size, second_hdr_size, first_datasize, second_datasize,
        first_offset, second_offset;
    acc_t *cut_second, *tmp_acc;

    if (!second)
    {
        first->acc_ext_link= NULL;
        return first;
    }

    assert (first->acc_length >= IP_MIN_HDR_SIZE);
    assert (second->acc_length >= IP_MIN_HDR_SIZE);

    first_hdr= (ip_hdr_t *)ptr2acc_data(first);
    first_offset= (ntohs(first_hdr->ih_flags_fragoff) &
        IH_FRAGOFF_MASK) * 8;
    first_hdr_size= (first_hdr->ih_vers_ihl & IH_IHL_MASK) * 4;
    first_datasize= ntohs(first_hdr->ih_length) - first_hdr_size;

    second_hdr= (ip_hdr_t *)ptr2acc_data(second);
    second_offset= (ntohs(second_hdr->ih_flags_fragoff) &
        IH_FRAGOFF_MASK) * 8;
    second_hdr_size= (second_hdr->ih_vers_ihl & IH_IHL_MASK) * 4;
    second_datasize= ntohs(second_hdr->ih_length) - second_hdr_size;

    assert (first_hdr_size + first_datasize == bf_bufsize(first));
    assert (second_hdr_size + second_datasize == bf_bufsize(second));
    assert (second_offset >= first_offset);

    if (second_offset > first_offset+first_datasize)
    {
        DBLOCK(1, printf("ip fragments out of order\n"));
        first->acc_ext_link= second;
        return first;
    }

    if (second_offset + second_datasize <= first_offset +
        first_datasize)
    {
        /* May cause problems if we try to merge. */
        bf_afree(first);
        return second;
    }

    if (!(second_hdr->ih_flags_fragoff & HTONS(IH_MORE_FRAGS)))
        first_hdr->ih_flags_fragoff &= ~HTONS(IH_MORE_FRAGS);

```

```
second_datasize= second_offset+second_datasize-(first_offset+
    first_datasize);
cut_second= bf_cut(second, second_hdr_size + first_offset+
    first_datasize-second_offset, second_datasize);
tmp_acc= second->acc_ext_link;
bf_afree(second);
second= tmp_acc;

first_datasize += second_datasize;
first_hdr->ih_length= htons(first_hdr_size + first_datasize);

first= bf_append (first, cut_second);
first->acc_ext_link= second;

assert (first_hdr_size + first_datasize == bf_bufsize(first));

    return first;
}

PRIVATE ip_ass_t *find_ass_ent (ip_port, id, proto, src, dst)
ip_port_t *ip_port;
ul6_t id;
ipproto_t proto;
ipaddr_t src;
ipaddr_t dst;
{
    ip_ass_t *new_ass_ent, *tmp_ass_ent;
    int i;
    acc_t *tmp_acc, *curr_acc;

    new_ass_ent= 0;

    for (i=0, tmp_ass_ent= ip_ass_table; i<IP_ASS_NR; i++,
        tmp_ass_ent++)
    {
        if (!tmp_ass_ent->ia_frags && tmp_ass_ent->ia_first_time)
        {
            DBLOCK(1,
                printf("strange ip_ass entry (can be a race condition)\n"));
            continue;
        }

        if ((tmp_ass_ent->ia_srcaddr == src) &&
            (tmp_ass_ent->ia_dstaddr == dst) &&
            (tmp_ass_ent->ia_proto == proto) &&
            (tmp_ass_ent->ia_id == id) &&
            (tmp_ass_ent->ia_port == ip_port))
        {
            return tmp_ass_ent;
        }
        if (!new_ass_ent || tmp_ass_ent->ia_first_time <
            new_ass_ent->ia_first_time)
        {
            new_ass_ent= tmp_ass_ent;
        }
    }

    if (new_ass_ent->ia_frags)
    {
        DBLOCK(2, printf("old frags id= %u, proto= %u, src= ",
            ntohs(new_ass_ent->ia_id),
            new_ass_ent->ia_proto);
            writeIpAddr(new_ass_ent->ia_srcaddr); printf(" dst= ");
            writeIpAddr(new_ass_ent->ia_dstaddr); printf(": ");
            ip_print_frags(new_ass_ent->ia_frags); printf("\n"));
        curr_acc= new_ass_ent->ia_frags->acc_ext_link;
        while (curr_acc)
        {
            tmp_acc= curr_acc->acc_ext_link;
            bf_afree(curr_acc);
            curr_acc= tmp_acc;
        }
        curr_acc= new_ass_ent->ia_frags;
        new_ass_ent->ia_frags= 0;
    }
}
```



```

        if (broadcast_dst(ip_port, new_ass_ent->ia_dstaddr))
        {
            DBLOCK(1, printf(
"ip_read'find_ass_ent: reassembly timeout for broadcast packet\n"));
            bf_afree(curr_acc); curr_acc= NULL;
        }
        else
        {
            icmp_snd_time_exceeded(ip_port->ip_port,
            curr_acc, ICMP_FRAG_REASSEM);
        }
    }
    new_ass_ent->ia_min_ttl= IP_MAX_TTL;
    new_ass_ent->ia_port= ip_port;
    new_ass_ent->ia_first_time= get_time();
    new_ass_ent->ia_srcaddr= src;
    new_ass_ent->ia_dstaddr= dst;
    new_ass_ent->ia_proto= proto;
    new_ass_ent->ia_id= id;

    return new_ass_ent;
}

PRIVATE int ip_frag_chk(pack)
acc_t *pack;
{
    ip_hdr_t *ip_hdr;
    int hdr_len;

    if (pack->acc_length < sizeof(ip_hdr_t))
    {
        DBLOCK(1, printf("wrong length\n"));
        return FALSE;
    }

    ip_hdr= (ip_hdr_t *)ptr2acc_data(pack);

    hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) * 4;
    if (pack->acc_length < hdr_len)
    {
        DBLOCK(1, printf("wrong length\n"));

        return FALSE;
    }

    if (((ip_hdr->ih_vers_ihl >> 4) & IH_VERSION_MASK) !=
        IP_VERSION)
    {
        DBLOCK(1, printf("wrong version (ih_vers_ihl=0x%x)\n",
            ip_hdr->ih_vers_ihl));
        return FALSE;
    }
    if (ntohs(ip_hdr->ih_length) != bf_bufsize(pack))
    {
        DBLOCK(1, printf("wrong size\n"));

        return FALSE;
    }
    if ((u16_t)~oneC_sum(0, (u16_t *)ip_hdr, hdr_len))
    {
        DBLOCK(1, printf("packet with wrong checksum(= %x)\n",
            (u16_t)~oneC_sum(0, (u16_t *)ip_hdr, hdr_len)));
        return FALSE;
    }
    if (hdr_len>IP_MIN_HDR_SIZE && ip_chk_hdropt((u8_t *)
        (ptr2acc_data(pack) + IP_MIN_HDR_SIZE),
        hdr_len-IP_MIN_HDR_SIZE))
    {
        DBLOCK(1, printf("packet with wrong options\n"));
        return FALSE;
    }
    return TRUE;
}

```

```
PUBLIC void ip_packet2user (ip_fd, pack, exp_time, data_len)
ip_fd_t *ip_fd;
acc_t *pack;
time_t exp_time;
size_t data_len;
{
    acc_t *tmp_pack;
    ip_hdr_t *ip_hdr;
    int result, ip_hdr_len;
    size_t transf_size;

    assert (ip_fd->if_flags & IFF_INUSE);
    if (!(ip_fd->if_flags & IFF_READ_IP))
    {
        if (pack->acc_linkC != 1)
        {
            tmp_pack= bf_dupacc(pack);
            bf_afree(pack);
            pack= tmp_pack;
            tmp_pack= NULL;
        }
        pack->acc_ext_link= NULL;
        if (ip_fd->if_rdbuf_head == NULL)
        {
            ip_fd->if_rdbuf_head= pack;
            ip_fd->if_exp_time= exp_time;
        }
        else
            ip_fd->if_rdbuf_tail->acc_ext_link= pack;
        ip_fd->if_rdbuf_tail= pack;
        return;
    }

    assert (pack->acc_length >= IP_MIN_HDR_SIZE);
    ip_hdr= (ip_hdr_t *)ptr2acc_data(pack);

    if (ip_fd->if_ipopt.nwio_flags & NWIO_RWDATONLY)
    {
        ip_hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) * 4;

        assert (data_len > ip_hdr_len);
        data_len -= ip_hdr_len;
        pack= bf_delhead(pack, ip_hdr_len);
    }

    if (data_len > ip_fd->if_rd_count)
    {
        tmp_pack= bf_cut (pack, 0, ip_fd->if_rd_count);
        bf_afree(pack);
        pack= tmp_pack;
        transf_size= ip_fd->if_rd_count;
    }
    else
        transf_size= data_len;

    if (ip_fd->if_put_pkt)
    {
        (*ip_fd->if_put_pkt)(ip_fd->if_srfd, pack, transf_size);
        return;
    }

    result= (*ip_fd->if_put_userdata)(ip_fd->if_srfd,
        (size_t)0, pack, FALSE);
    if (result >= 0)
    {
        if (data_len > transf_size)
            result= EPACKSIZE;
        else
            result= transf_size;
    }

    ip_fd->if_flags &= ~IFF_READ_IP;
    result= (*ip_fd->if_put_userdata)(ip_fd->if_srfd, result,
        (acc_t *)0, FALSE);
}
```

```

    assert (result >= 0);
}

PUBLIC void ip_port_arrive (ip_port, pack, ip_hdr)
ip_port_t *ip_port;
acc_t *pack;
ip_hdr_t *ip_hdr;
{
    ip_fd_t *ip_fd, *first_fd, *share_fd;
    unsigned long ip_pack_stat;
    unsigned size;
    int i;
    int hash, proto;
    time_t exp_time;

    assert (pack->acc_linkC>0);
    assert (pack->acc_length >= IP_MIN_HDR_SIZE);

    if (ntohs(ip_hdr->ih_flags_fragoff) & (IH_FRAGOFF_MASK|IH_MORE_FRAGS))
    {
        pack= reassemble (ip_port, pack, ip_hdr);
        if (!pack)
            return;
        assert (pack->acc_length >= IP_MIN_HDR_SIZE);
        ip_hdr= (ip_hdr_t *)ptr2acc_data(pack);
        assert (!(ntohs(ip_hdr->ih_flags_fragoff) &
            (IH_FRAGOFF_MASK|IH_MORE_FRAGS)));
    }
    size= ntohs(ip_hdr->ih_length);
    if (size > bf_bufsize(pack))
    {
        /* Should discard packet */
        assert(0);
        bf_afree(pack); pack= NULL;
        return;
    }

    exp_time= get_time() + (ip_hdr->ih_ttl+1) * HZ;

    if (ip_hdr->ih_dst == ip_port->ip_ipaddr)
        ip_pack_stat= NWIO_EN_LOC;
    else
        ip_pack_stat= NWIO_EN_BROAD;

    proto= ip_hdr->ih_proto;
    hash= proto & (IP_PROTO_HASH_NR-1);

    first_fd= NULL;
    for (i= 0; i<2; i++)
    {
        share_fd= NULL;

        ip_fd= (i == 0) ? ip_port->ip_proto_any :
            ip_port->ip_proto[hash];
        for (; ip_fd; ip_fd= ip_fd->if_proto_next)
        {
            if (i && ip_fd->if_ipopt.nwio_proto != proto)
                continue;
            if (!(ip_fd->if_ipopt.nwio_flags & ip_pack_stat))
                continue;
            if ((ip_fd->if_ipopt.nwio_flags & NWIO_REMSPEC) &&
                ip_hdr->ih_src != ip_fd->if_ipopt.nwio_rem)
            {
                continue;
            }
            if ((ip_fd->if_ipopt.nwio_flags & NWIO_ACC_MASK) ==
                NWIO_SHARED)
            {
                if (!share_fd)
                {
                    share_fd= ip_fd;
                    continue;
                }
                if (!ip_fd->if_rdbuf_head)

```

```

        share_fd= ip_fd;
        continue;
    }
    if (!first_fd)
    {
        first_fd= ip_fd;
        continue;
    }
    pack->acc_linkC++;
    ip_packet2user(ip_fd, pack, exp_time, size);
}
if (share_fd)
{
    pack->acc_linkC++;
    ip_packet2user(share_fd, pack, exp_time, size);
}
}
if (first_fd)
{
    if (first_fd->if_put_pkt &&
        (first_fd->if_flags & IFF_READ_IP) &&
        !(first_fd->if_ipopt.nwio_flags & NWIO_RWDATONLY))
    {
        (*first_fd->if_put_pkt)(first_fd->if_srfd, pack,
                                size);
    }
    else
        ip_packet2user(first_fd, pack, exp_time, size);
}
else
{
    if (ip_pack_stat == NWIO_EN_LOC)
    {
        DBLOCK(0x01,
            printf("ip_port arrive: dropping packet for proto %d\n",
                  proto));
    }
    else
    {
        DBLOCK(0x20, printf("dropping packet for proto %d\n",
                            proto));
    }
    bf_afree(pack);
}
}

PUBLIC void ip_arrived(ip_port, pack)
ip_port_t *ip_port;
acc_t *pack;
{
    ip_hdr_t *ip_hdr;
    ipaddr_t dest;
    int ip_frag_len, ip_hdr_len, highbyte;
    size_t pack_size;
    acc_t *tmp_pack, *hdr_pack;
    ev_arg_t ev_arg;

    pack_size= bf_bufsize(pack);

    if (pack_size < IP_MIN_HDR_SIZE)
    {
        DBLOCK(1, printf("wrong acc_length\n"));
        bf_afree(pack);
        return;
    }
    pack= bf_align(pack, IP_MIN_HDR_SIZE, 4);
    pack= bf_packIfLess(pack, IP_MIN_HDR_SIZE);
assert (pack->acc_length >= IP_MIN_HDR_SIZE);

    ip_hdr= (ip_hdr_t *)ptr2acc_data(pack);
    ip_hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
    if (ip_hdr_len>IP_MIN_HDR_SIZE)
    {

```

```

        pack= bf_packIfLess(pack, ip_hdr_len);
        ip_hdr= (ip_hdr_t *)ptr2acc_data(pack);
    }
    ip_frag_len= ntohs(ip_hdr->ih_length);
    if (ip_frag_len != pack_size)
    {
        if (pack_size < ip_frag_len)
        {
            /* Sent ICMP? */
            DBLOCK(1, printf("wrong acc_length\n"));
            bf_afree(pack);
            return;
        }
        assert(ip_frag_len<pack_size);
        tmp_pack= pack;
        pack= bf_cut(tmp_pack, 0, ip_frag_len);
        bf_afree(tmp_pack);
        pack_size= ip_frag_len;
    }

    if (!ip_frag_chk(pack))
    {
        DBLOCK(1, printf("fragment not allright\n"));
        bf_afree(pack);
        return;
    }

    /* Decide about local delivery or routing. Local delivery can happen
     * when the destination is the local ip address, or one of the
     * broadcast addresses and the packet happens to be delivered
     * point-to-point.
     */

    dest= ip_hdr->ih_dst;

    if (dest == ip_port->ip_ipaddr)
    {
        ip_port_arrive (ip_port, pack, ip_hdr);
        return;
    }
    if (broadcast_dst(ip_port, dest))
    {
        ip_port_arrive (ip_port, pack, ip_hdr);
        return;
    }

    if (pack->acc_linkC != 1 || pack->acc_buffer->buf_linkC != 1)
    {
        /* Get a private copy of the IP header */
        hdr_pack= bf_memreq(ip_hdr_len);
        memcpy(ptr2acc_data(hdr_pack), ip_hdr, ip_hdr_len);
        pack= bf_delhead(pack, ip_hdr_len);
        hdr_pack->acc_next= pack;
        pack= hdr_pack; hdr_pack= NULL;
        ip_hdr= (ip_hdr_t *)ptr2acc_data(pack);
    }
    assert(pack->acc_linkC == 1);
    assert(pack->acc_buffer->buf_linkC == 1);

    /* Try to decrement the ttl field with one. */
    if (ip_hdr->ih_ttl < 2)
    {
        icmp_snd_time_exceeded(ip_port->ip_port, pack,
                               ICMP_TTL_EXC);
        return;
    }
    ip_hdr->ih_ttl--;
    ip_hdr_chksum(ip_hdr, ip_hdr_len);

    /* Avoid routing to bad destinations. */
    highbyte= ntohl(dest) >> 24;
    if (highbyte == 0 || highbyte == 127 ||
        (highbyte == 169 && ((ntohl(dest) >> 16) & 0xff) == 254)) ||
        highbyte >= 0xe0)

```

```

    {
        /* Bogus destination address */
        bf_afree(pack);
        return;
    }

    /* Further processing from an event handler */
    if (pack->acc_linkC != 1)
    {
        tmp_pack= bf_dupacc(pack);
        bf_afree(pack);
        pack= tmp_pack;
        tmp_pack= NULL;
    }
    pack->acc_ext_link= NULL;
    if (ip_port->ip_routeq_head)
    {
        ip_port->ip_routeq_tail->acc_ext_link= pack;
        ip_port->ip_routeq_tail= pack;
        return;
    }

    ip_port->ip_routeq_head= pack;
    ip_port->ip_routeq_tail= pack;
    ev_arg.ev_ptr= ip_port;
    ev_enqueue(&ip_port->ip_routeq_event, route_packets, ev_arg);
}

PUBLIC void ip_arrived_broadcast(ip_port, pack)
ip_port_t *ip_port;
acc_t *pack;
{
    ip_hdr_t *ip_hdr;
    int ip_frag_len, ip_hdr_len;
    size_t pack_size;
    acc_t *tmp_pack;

    pack_size= bf_bufsize(pack);

    if (pack_size < IP_MIN_HDR_SIZE)
    {
        DBLOCK(1, printf("wrong acc_length\n"));
        bf_afree(pack);
        return;
    }
    pack= bf_align(pack, IP_MIN_HDR_SIZE, 4);
    pack= bf_packIfLess(pack, IP_MIN_HDR_SIZE);
assert (pack->acc_length >= IP_MIN_HDR_SIZE);

    ip_hdr= (ip_hdr_t *)ptr2acc_data(pack);

    DIFBLOCK(0x20, (ip_hdr->ih_dst & HTONL(0xf0000000)) == HTONL(0xe0000000),
        printf("got multicast packet\n"));

    ip_hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
    if (ip_hdr_len > IP_MIN_HDR_SIZE)
    {
        pack= bf_align(pack, IP_MIN_HDR_SIZE, 4);
        pack= bf_packIfLess(pack, ip_hdr_len);
        ip_hdr= (ip_hdr_t *)ptr2acc_data(pack);
    }
    ip_frag_len= ntohs(ip_hdr->ih_length);
    if (ip_frag_len < pack_size)
    {
        tmp_pack= pack;
        pack= bf_cut(tmp_pack, 0, ip_frag_len);
        bf_afree(tmp_pack);
    }

    if (!ip_frag_chk(pack))
    {
        DBLOCK(1, printf("fragment not allright\n"));
        bf_afree(pack);
        return;
    }

```

```

    }

    if (!broadcast_dst(ip_port, ip_hdr->ih_dst))
    {
#if 0
        printf(
            "ip[%d]: broadcast packet for ip-nonbroadcast addr, src=" ,
                ip_port->ip_port);
        writeIpAddr(ip_hdr->ih_src);
        printf(" dst=");
        writeIpAddr(ip_hdr->ih_dst);
        printf("\n");
#endif
        bf_afree(pack);
        return;
    }

    ip_port_arrive (ip_port, pack, ip_hdr);
}

PRIVATE void route_packets(ev, ev_arg)
event_t *ev;
ev_arg_t ev_arg;
{
    ip_port_t *ip_port;
    ipaddr_t dest;
    acc_t *pack;
    iroute_t *iroute;
    ip_port_t *next_port;
    int r, type;
    ip_hdr_t *ip_hdr;
    size_t req_mtu;

    ip_port= ev_arg.ev_ptr;
    assert(&ip_port->ip_routeq_event == ev);

    while (pack= ip_port->ip_routeq_head, pack != NULL)
    {
        ip_port->ip_routeq_head= pack->acc_ext_link;

        ip_hdr= (ip_hdr_t *)ptr2acc_data(pack);
        dest= ip_hdr->ih_dst;

        iroute= iroute_frag(ip_port->ip_port, dest);
        if (iroute == NULL || iroute->irt_dist == IRTD_UNREACHABLE)
        {
            /* Also unreachable */
            /* Finding out if we send a network unreachable is too
             * much trouble.
             */
            if (iroute == NULL)
            {
                printf("ip[%d]: no route to ",
                    ip_port->ip_port_table);
                writeIpAddr(dest);
                printf("\n");
            }
            icmp_snd_unreachable(ip_port->ip_port, pack,
                ICMP_HOST_UNRCH);
            continue;
        }
        next_port= &ip_port_table[iroute->irt_port];

        if (ip_hdr->ih_flags_fragoff & HTONS(IH_DONT_FRAG))
        {
            req_mtu= bf_bufsize(pack);
            if (req_mtu > next_port->ip_mtu ||
                (iroute->irt_mtu && req_mtu>iroute->irt_mtu))
            {
                icmp_snd_mtu(ip_port->ip_port, pack,
                    next_port->ip_mtu);
                continue;
            }
        }
    }
}

```

```

    if (next_port != ip_port)
    {
        if (iroute->irt_gateway != 0)
        {
            /* Just send the packet to the next gateway */
            pack->acc_linkC++; /* Extra ref for ICMP */
            r= next_port->ip_dev_send(next_port,
                                     iroute->irt_gateway,
                                     pack, IP_LT_NORMAL);
            if (r == EDSTNOTRCH)
            {
                printf("ip[%d]: gw ",
                       ip_port-ip_port_table);
                writeIpAddr(iroute->irt_gateway);
                printf(" on ip[%d] is down for dest ",
                       next_port-ip_port_table);
                writeIpAddr(dest);
                printf("\n");
                icmp_snd_unreachable(next_port-
                                     ip_port_table, pack,
                                     ICMP_HOST_UNRCH);
                pack= NULL;
            }
            else
            {
                assert(r == 0);
                bf_afree(pack); pack= NULL;
            }
            continue;
        }
        /* The packet is for the attached network. Special
         * addresses are the ip address of the interface and
         * net.0 if no IP_42BSD_BCAST.
         */
        if (dest == next_port->ip_ipaddr)
        {
            ip_port_arrive (next_port, pack, ip_hdr);
            continue;
        }
        if (dest == iroute->irt_dest)
        {
            /* Never forward obsolete directed broadcasts */
            type= IP_LT_BROADCAST;

            /* Bogus destination address */
            DBLOCK(1, printf(
                "ip[%d]: dropping old-fashioned directed broadcast ",
                    ip_port-ip_port_table);
                writeIpAddr(dest);
                printf("\n"););
            icmp_snd_unreachable(next_port-ip_port_table,
                                pack, ICMP_HOST_UNRCH);
            continue;
        }
        #endif
        else if (dest == (iroute->irt_dest |
                         ~iroute->irt_subnetmask))
        {
            if (!ip_forward_directed_bcast)
            {
                /* Do not forward directed broadcasts */
                DBLOCK(1, printf(
                    "ip[%d]: dropping directed broadcast ",
                        ip_port-ip_port_table);
                    writeIpAddr(dest);
                    printf("\n"););
                icmp_snd_unreachable(next_port-
                                    ip_port_table, pack,
                                    ICMP_HOST_UNRCH);
                continue;
            }
            else

```



```

        type= IP_LT_BROADCAST;
    }
    else
        type= IP_LT_NORMAL;

    /* Just send the packet to it's destination */
    pack->acc_linkC++; /* Extra ref for ICMP */
    r= next_port->ip_dev_send(next_port, dest, pack, type);
    if (r == EDSTNOTRCH)
    {
        DBLOCK(1, printf("ip[%d]: next hop ",
            ip_port-ip_port_table);
            writeIpAddr(dest);
            printf(" on ip[%d] is down\n",
                next_port-ip_port_table));
        icmp_snd_unreachable(next_port-ip_port_table,
            pack, ICMP_HOST_UNRCH);
        pack= NULL;
    }
    else
    {
        assert(r == 0 || (printf("r=%d\n", r), 0));
        bf_afree(pack); pack= NULL;
    }
    continue;
}

/* Now we know that the packet should be routed over the same
 * network as it came from. If there is a next hop gateway,
 * we can send the packet to that gateway and send a redirect
 * ICMP to the sender if the sender is on the attached
 * network. If there is no gateway complain.
 */
if (iroute->irt_gateway == 0)
{
    printf("ip_arrived: packet should not be here, src=");
    writeIpAddr(ip_hdr->ih_src);
    printf(" dst=");
    writeIpAddr(ip_hdr->ih_dst);
    printf("\n");
    bf_afree(pack);
    continue;
}
if (((ip_hdr->ih_src ^ ip_port->ip_ipaddr) &
    ip_port->ip_subnetmask) == 0)
{
    /* Finding out if we can send a network redirect
     * instead of a host redirect is too much trouble.
     */
    pack->acc_linkC++;
    icmp_snd_redirect(ip_port->ip_port, pack,
        ICMP_REDIRECT_HOST, iroute->irt_gateway);
}
else
{
    printf("ip_arrived: packet is wrongly routed, src=");
    writeIpAddr(ip_hdr->ih_src);
    printf(" dst=");
    writeIpAddr(ip_hdr->ih_dst);
    printf("\n");
    printf("in port %d, output %d, dest net ",
        ip_port->ip_port,
        iroute->irt_port);
    writeIpAddr(iroute->irt_dest);
    printf("/");
    writeIpAddr(iroute->irt_subnetmask);
    printf(" next hop ");
    writeIpAddr(iroute->irt_gateway);
    printf("\n");
    bf_afree(pack);
    continue;
}

/* No code for unreachable ICMPs here. The sender should
 * process the ICMP redirect and figure it out.

```

```

        */
        ip_port->ip_dev_send(ip_port, iroute->irt_gateway, pack,
                             IP_LT_NORMAL);
    }
}

PRIVATE int broadcast_dst(ip_port, dest)
ip_port_t *ip_port;
ipaddr_t dest;
{
    ipaddr_t my_ipaddr, netmask, classmask;

    /* Treat class D (multicast) address as broadcasts. */
    if ((dest & HTONL(0xF0000000)) == HTONL(0xE0000000))
    {
        return 1;
    }

    /* Accept without complaint if netmask not yet configured. */
    if (!(ip_port->ip_flags & IPF_NETMASKSET))
    {
        return 1;
    }

    /* Two possibilities, 0 (iff IP_42BSD_BCAST) and -1 */
    if (dest == HTONL((ipaddr_t)-1))
        return 1;
#if IP_42BSD_BCAST
    if (dest == HTONL((ipaddr_t)0))
        return 1;
#endif

    netmask= ip_port->ip_subnetmask;
    my_ipaddr= ip_port->ip_ipaddr;

    if (((my_ipaddr ^ dest) & netmask) != 0)
    {
        classmask= ip_port->ip_classfulmask;

        /* Not a subnet broadcast, maybe a classful broadcast */
        if (((my_ipaddr ^ dest) & classmask) != 0)
        {
            return 0;
        }

        /* Two possibilities, net.0 (iff IP_42BSD_BCAST) and net.-1 */
        if ((dest & ~classmask) == ~classmask)
        {
            return 1;
        }
    }
#if IP_42BSD_BCAST
    if ((dest & ~classmask) == 0)
        return 1;
#endif

    return 0;
}

    if (!(ip_port->ip_flags & IPF_SUBNET_BCAST))
        return 0; /* No subnet broadcasts on this network */

    /* Two possibilities, subnet.0 (iff IP_42BSD_BCAST) and subnet.-1 */
    if ((dest & ~netmask) == ~netmask)
        return 1;
#if IP_42BSD_BCAST
    if ((dest & ~netmask) == 0)
        return 1;
#endif

    return 0;
}

void ip_process_loopb(ev, arg)
event_t *ev;
ev_arg_t arg;
{
    ip_port_t *ip_port;
    acc_t *pack;

```

```
    ip_port= arg.ev_ptr;
    assert(ev == &ip_port->ip_loopb_event);

    while(pack= ip_port->ip_loopb_head, pack != NULL)
    {
        ip_port->ip_loopb_head= pack->acc_ext_link;
        ip_arrived(ip_port, pack);
    }
}

/*
 * $PchId: ip_read.c,v 1.33 2005/06/28 14:18:50 philip Exp $
 */
```

```
/*
ip_write.c

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "buf.h"
#include "event.h"
#include "type.h"

#include "arp.h"
#include "assert.h"
#include "clock.h"
#include "eth.h"
#include "icmp_lib.h"
#include "io.h"
#include "ip.h"
#include "ip_int.h"
#include "ipr.h"

THIS_FILE

FORWARD void error_reply ARGS(( ip_fd_t *fd, int error ));

PUBLIC int ip_write (fd, count)
int fd;
size_t count;
{
    ip_fd_t *ip_fd;
    acc_t *pack;
    int r;

    ip_fd= &ip_fd_table[fd];
    if (count > IP_MAX_PACKSIZE)
    {
        error_reply (ip_fd, EPACKSIZE);
        return NW_OK;
    }
    pack= (*ip_fd->if_get_userdata)(ip_fd->if_srfd, (size_t)0,
        count, FALSE);
    if (!pack)
        return NW_OK;
    r= ip_send(fd, pack, count);
    assert(r != NW_WOULDBLOCK);

    if (r == NW_OK)
        error_reply (ip_fd, count);
    else
        error_reply (ip_fd, r);
    return NW_OK;
}

PUBLIC int ip_send(fd, data, data_len)
int fd;
acc_t *data;
size_t data_len;
{
    ip_port_t *ip_port;
    ip_fd_t *ip_fd;
    ip_hdr_t *ip_hdr, *tmp_hdr;
    ipaddr_t dstaddr, nexthop, hostrep_dst, my_ipaddr, netmask;
    u8_t *addrInBytes;
    acc_t *tmp_pack, *tmp_pack1;
    int hdr_len, hdr_opt_len, r;
    int type, ttl;
    size_t req_mtu;
    ev_arg_t arg;

    ip_fd= &ip_fd_table[fd];
    ip_port= ip_fd->if_port;

    if (!(ip_fd->if_flags & IFF_OPTSET))
    {
```

```

        bf_afree(data);
        return EBADMODE;
    }

    if (!(ip_fd->if_port->ip_flags & IPF_IPADDRSET))
    {
        /* Interface is down. What kind of error do we want? For
         * the moment, we return OK.
         */
        bf_afree(data);
        return NW_OK;
    }

    data_len= bf_bufsize(data);

    if (ip_fd->if_ipopt.nwio_flags & NWIO_RWDATONLY)
    {
        tmp_pack= bf_memreq(IP_MIN_HDR_SIZE);
        tmp_pack->acc_next= data;
        data= tmp_pack;
        data_len += IP_MIN_HDR_SIZE;
    }
    if (data_len<IP_MIN_HDR_SIZE)
    {
        bf_afree(data);
        return EPACKSIZE;
    }

    data= bf_packIffLess(data, IP_MIN_HDR_SIZE);
    ip_hdr= (ip_hdr_t *)ptr2acc_data(data);
    if (data->acc_linkC != 1 || data->acc_buffer->buf_linkC != 1)
    {
        tmp_pack= bf_memreq(IP_MIN_HDR_SIZE);
        tmp_hdr= (ip_hdr_t *)ptr2acc_data(tmp_pack);
        *tmp_hdr= *ip_hdr;
        tmp_pack->acc_next= bf_cut(data, IP_MIN_HDR_SIZE,
                                   data_len-IP_MIN_HDR_SIZE);
        bf_afree(data);
        ip_hdr= tmp_hdr;
        data= tmp_pack;
        assert (data->acc_length >= IP_MIN_HDR_SIZE);
    }

    if (ip_fd->if_ipopt.nwio_flags & NWIO_HDR_O_SPEC)
    {
        hdr_opt_len= ip_fd->if_ipopt.nwio_hdopt.iho_opt_siz;
        if (hdr_opt_len)
        {
            tmp_pack= bf_cut(data, 0, IP_MIN_HDR_SIZE);
            tmp_pack1= bf_cut (data, IP_MIN_HDR_SIZE,
                               data_len-IP_MIN_HDR_SIZE);
            bf_afree(data);
            data= bf_packIffLess(tmp_pack, IP_MIN_HDR_SIZE);
            ip_hdr= (ip_hdr_t *)ptr2acc_data(data);
            tmp_pack= bf_memreq (hdr_opt_len);
            memcpy (ptr2acc_data(tmp_pack), ip_fd->if_ipopt.
                    nwio_hdopt.iho_data, hdr_opt_len);
            data->acc_next= tmp_pack;
            tmp_pack->acc_next= tmp_pack1;
            hdr_len= IP_MIN_HDR_SIZE+hdr_opt_len;
        }
        else
        {
            hdr_len= IP_MIN_HDR_SIZE;
            ip_hdr->ih_vers_ihl= hdr_len/4;
            ip_hdr->ih_tos= ip_fd->if_ipopt.nwio_tos;
            ip_hdr->ih_flags_fragoff= 0;
            if (ip_fd->if_ipopt.nwio_df)
                ip_hdr->ih_flags_fragoff |= HTONS(IH_DONT_FRAG);
            ip_hdr->ih_ttl= ip_fd->if_ipopt.nwio_ttl;
            ttl= ORTD_UNREACHABLE+1;          /* Don't check TTL */
        }
    }
    else
    {
        hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK)*4;
    }

```

```

    r= NW_OK;
    if (hdr_len<IP_MIN_HDR_SIZE)
        r= EINVAL;
    else if (hdr_len>data_len)
        r= EPACKSIZE;
    else if (!ip_hdr->ih_ttl)
        r= EINVAL;
    if (r != NW_OK)
    {
        bf_afree(data);
        return r;
    }

    data= bf_packIfLess(data, hdr_len);
    ip_hdr= (ip_hdr_t *)ptr2acc_data(data);
    if (hdr_len != IP_MIN_HDR_SIZE)
    {
        r= ip_chk_hdropt((u8_t *) (ptr2acc_data(data) +
            IP_MIN_HDR_SIZE),
            hdr_len-IP_MIN_HDR_SIZE);
        if (r != NW_OK)
        {
            bf_afree(data);
            return r;
        }
    }
    ttl= ip_hdr->ih_ttl;
}

ip_hdr->ih_vers_ihl= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) |
    (IP_VERSION << 4);
ip_hdr->ih_length= htons(data_len);
ip_hdr->ih_flags_fragoff &= ~HTONS(IH_FRAGOFF_MASK |
    IH_FLAGS_UNUSED | IH_MORE_FRAGS);
if (ip_fd->if_ipopt.nwio_flags & NWIO_PROTOSPEC)
    ip_hdr->ih_proto= ip_fd->if_ipopt.nwio_proto;
ip_hdr->ih_id= htons(ip_port->ip_frame_id++);
ip_hdr->ih_src= ip_fd->if_port->ip_ipaddr;
if (ip_fd->if_ipopt.nwio_flags & NWIO_REMSPEC)
    ip_hdr->ih_dst= ip_fd->if_ipopt.nwio_rem;

netmask= ip_port->ip_subnetmask;
my_ipaddr= ip_port->ip_ipaddr;

dstaddr= ip_hdr->ih_dst;
hostrep_dst= ntohl(dstaddr);
r= 0;
if (hostrep_dst == (ipaddr_t)-1)
    ; /* OK, local broadcast */
else if ((hostrep_dst & 0xe0000001) == 0xe0000001)
    ; /* OK, Multicast */
else if ((hostrep_dst & 0xf0000001) == 0xf0000001)
    r= EBADDEST; /* Bad class */
else if ((dstaddr ^ my_ipaddr) & netmask)
    ; /* OK, remote destination */
else if (!(dstaddr & ~netmask) &&
    (ip_port->ip_flags & IPF_SUBNET_BCAST))
{
    r= EBADDEST; /* Zero host part */
}
if (r<0)
{
    DIFBLOCK(1, r == EBADDEST,
        printf("bad destination: ");
        writeIpAddr(ip_hdr->ih_dst);
        printf("\n"));
    bf_afree(data);
    return r;
}
ip_hdr_chksum(ip_hdr, hdr_len);

data= bf_packIfLess(data, IP_MIN_HDR_SIZE);
assert (data->acc_length >= IP_MIN_HDR_SIZE);
ip_hdr= (ip_hdr_t *)ptr2acc_data(data);

```

```
if (ip_hdr->ih_flags_fragoff & HTONS(IH_DONT_FRAG))
{
    req_mtu= bf_bufsize(data);
    if (req_mtu > ip_port->ip_mtu)
    {
        DBLOCK(1, printf(
            "packet is larger than link MTU and DF is set\n" ));
        bf_afree(data);
        return EPACKSIZE;
    }
}
else
    req_mtu= 0;

addrInBytes= (u8_t *)&dstaddr;

if ((addrInBytes[0] & 0xff) == 0x7f) /* local loopback */
{
    assert (data->acc_linkC == 1);
    dstaddr= ip_hdr->ih_dst; /* swap src and dst
                           * addresses */

    ip_hdr->ih_dst= ip_hdr->ih_src;
    ip_hdr->ih_src= dstaddr;
    data->acc_ext_link= NULL;
    if (ip_port->ip_loopb_head == NULL)
    {
        ip_port->ip_loopb_head= data;
        arg.ev_ptr= ip_port;
        ev_enqueue(&ip_port->ip_loopb_event,
            ip_process_loopb, arg);
    }
    else
        ip_port->ip_loopb_tail->acc_ext_link= data;
    ip_port->ip_loopb_tail= data;

    return NW_OK;
}

if ((dstaddr & HTONL(0xe0000000)) == HTONL(0xe0000000))
{
    if (dstaddr == (ipaddr_t)-1)
    {
        r= (*ip_port->ip_dev_send)(ip_port, dstaddr, data,
            IP_LT_BROADCAST);
        return r;
    }
    if (ip_nettype(dstaddr) == IPNT_CLASS_D)
    {
        /* Multicast, what about multicast routing? */
        r= (*ip_port->ip_dev_send)(ip_port, dstaddr, data,
            IP_LT_MULTICAST);
        return r;
    }
}

if (dstaddr == my_ipaddr)
{
    assert (data->acc_linkC == 1);

    data->acc_ext_link= NULL;
    if (ip_port->ip_loopb_head == NULL)
    {
        ip_port->ip_loopb_head= data;
        arg.ev_ptr= ip_port;
        ev_enqueue(&ip_port->ip_loopb_event,
            ip_process_loopb, arg);
    }
    else
        ip_port->ip_loopb_tail->acc_ext_link= data;
    ip_port->ip_loopb_tail= data;

    return NW_OK;
}
```

```

    if (((dstaddr ^ my_ipaddr) & netmask) == 0)
    {
        type= ((dstaddr == (my_ipaddr | ~netmask) &&
            (ip_port->ip_flags & IPF_SUBNET_BCAST)) ?
            IP_LT_BROADCAST : IP_LT_NORMAL);

        r= (*ip_port->ip_dev_send)(ip_port, dstaddr, data, type);
        return r;
    }

    r= oroute_frag (ip_port - ip_port_table, dstaddr, ttl, req_mtu,
        &nexthop);

    if (r == NW_OK)
    {
        if (nexthop == ip_port->ip_ipaddr)
        {
            data->acc_ext_link= NULL;
            if (ip_port->ip_loopb_head == NULL)
            {
                ip_port->ip_loopb_head= data;
                arg.ev_ptr= ip_port;
                ev_enqueue(&ip_port->ip_loopb_event,
                    ip_process_loopb, arg);
            }
            else
                ip_port->ip_loopb_tail->acc_ext_link= data;
            ip_port->ip_loopb_tail= data;
        }
        else
        {
            r= (*ip_port->ip_dev_send)(ip_port,
                nexthop, data, IP_LT_NORMAL);
        }
    }
    else
    {
        DBLOCK(0x10, printf("got error %d\n", r));
        bf_afree(data);
    }
    return r;
}

PUBLIC void ip_hdr_chksum(ip_hdr, ip_hdr_len)
ip_hdr_t *ip_hdr;
int ip_hdr_len;
{
    ip_hdr->ih_hdr_chk= 0;
    ip_hdr->ih_hdr_chk= ~oneC_sum (0, (u16_t *)ip_hdr, ip_hdr_len);
}

PUBLIC acc_t *ip_split_pack (ip_port, ref_last, mtu)
ip_port_t *ip_port;
acc_t **ref_last;
int mtu;
{
    int pack_siz;
    ip_hdr_t *first_hdr, *second_hdr;
    int first_hdr_len, second_hdr_len;
    int first_data_len, second_data_len;
    int data_len, max_data_len, nfrags, new_first_data_len;
    int first_opt_size, second_opt_size;
    acc_t *first_pack, *second_pack, *tmp_pack;
    u8_t *first_optptr, *second_optptr;
    int i, optlen;

    first_pack= *ref_last;
    *ref_last= 0;
    second_pack= 0;

    first_pack= bf_align(first_pack, IP_MIN_HDR_SIZE, 4);
    first_pack= bf_packIfLess(first_pack, IP_MIN_HDR_SIZE);
    assert (first_pack->acc_length >= IP_MIN_HDR_SIZE);

```



```

first_hdr= (ip_hdr_t *)ptr2acc_data(first_pack);
first_hdr_len= (first_hdr->ih_vers_ihl & IH_IHL_MASK) * 4;
if (first_hdr_len>IP_MIN_HDR_SIZE)
{
    first_pack= bf_packIffLess(first_pack, first_hdr_len);
    first_hdr= (ip_hdr_t *)ptr2acc_data(first_pack);
}

pack_siz= bf_bufsize(first_pack);
assert(pack_siz > mtu);

assert (!(first_hdr->ih_flags_fragoff & HTONS(IH_DONT_FRAG)));

if (first_pack->acc_linkC != 1 ||
    first_pack->acc_buffer->buf_linkC != 1)
{
    /* Get a private copy of the IP header */
    tmp_pack= bf_memreq(first_hdr_len);
    memcpy(ptr2acc_data(tmp_pack), first_hdr, first_hdr_len);
    first_pack= bf_delhead(first_pack, first_hdr_len);
    tmp_pack->acc_next= first_pack;
    first_pack= tmp_pack; tmp_pack= NULL;
    first_hdr= (ip_hdr_t *)ptr2acc_data(first_pack);
}

data_len= ntohs(first_hdr->ih_length) - first_hdr_len;

/* Try to split the packet evenly. */
assert(mtu > first_hdr_len);
max_data_len= mtu-first_hdr_len;
nfrags= (data_len/max_data_len)+1;
new_first_data_len= data_len/nfrags;
if (new_first_data_len < 8)
{
    /* Special case for extremely small MTUs */
    new_first_data_len= 8;
}
new_first_data_len &= ~7; /* data goes in 8 byte chunks */

assert(new_first_data_len >= 8);
assert(new_first_data_len+first_hdr_len <= mtu);

second_data_len= data_len-new_first_data_len;
second_pack= bf_cut(first_pack, first_hdr_len+
    new_first_data_len, second_data_len);
tmp_pack= first_pack;
first_data_len= new_first_data_len;
first_pack= bf_cut(tmp_pack, 0, first_hdr_len+first_data_len);
bf_afree(tmp_pack);
tmp_pack= bf_memreq(first_hdr_len);
tmp_pack->acc_next= second_pack;
second_pack= tmp_pack;
second_hdr= (ip_hdr_t *)ptr2acc_data(second_pack);
*second_hdr= *first_hdr;
second_hdr->ih_flags_fragoff= htons(
    ntohs(first_hdr->ih_flags_fragoff)+(first_data_len/8));

first_opt_size= first_hdr_len-IP_MIN_HDR_SIZE;
second_opt_size= 0;
if (first_opt_size)
{
    first_pack= bf_packIffLess (first_pack,
        first_hdr_len);
    first_hdr= (ip_hdr_t *)ptr2acc_data(first_pack);
    assert (first_pack->acc_length>=first_hdr_len);
    first_optptr= (u8_t *)ptr2acc_data(first_pack)+
        IP_MIN_HDR_SIZE;
    second_optptr= (u8_t *)ptr2acc_data(
        second_pack)+IP_MIN_HDR_SIZE;
    i= 0;
    while (i<first_opt_size)
    {
        switch (*first_optptr & IP_OPT_NUMBER)

```

```

        {
            case 0:
            case 1:
                optlen= 1;
                break;
            default:
                optlen= first_optptr[1];
                break;
        }
        assert (i + optlen <= first_opt_size);
        i += optlen;
        if (*first_optptr & IP_OPT_COPIED)
        {
            second_opt_size += optlen;
            while (optlen--)
                *second_optptr++=
                    *first_optptr++;
        }
        else
            first_optptr += optlen;
    }
    while (second_opt_size & 3)
    {
        *second_optptr++= 0;
        second_opt_size++;
    }
}
second_hdr_len= IP_MIN_HDR_SIZE + second_opt_size;

second_hdr->ih_vers_ihl= (second_hdr->ih_vers_ihl & 0xf0)
    + (second_hdr_len/4);
second_hdr->ih_length= htons(second_data_len+
    second_hdr_len);
second_pack->acc_length= second_hdr_len;

assert(first_pack->acc_linkC == 1);
assert(first_pack->acc_buffer->buf_linkC == 1);

first_hdr->ih_flags_fragoff |= HTONS(IH_MORE_FRAGS);
first_hdr->ih_length= htons(first_data_len+
    first_hdr_len);
assert (!(second_hdr->ih_flags_fragoff & HTONS(IH_DONT_FRAG)));

ip_hdr_chksum(first_hdr, first_hdr_len);
if (second_data_len+second_hdr_len <= mtu)
{
    /* second_pack will not be split any further, so we have to
     * calculate the header checksum.
     */
    ip_hdr_chksum(second_hdr, second_hdr_len);
}

*ref_last= second_pack;

return first_pack;
}

PRIVATE void error_reply (ip_fd, error)
ip_fd_t *ip_fd;
int error;
{
    if ((*ip_fd->if_get_userdata)(ip_fd->if_srfd, (size_t)error,
        (size_t)0, FALSE))
    {
        ip_panic(( "can't error_reply" ));
    }
}

/*
 * $PchId: ip_write.c,v 1.22 2004/08/03 11:11:04 philip Exp $
 */

```

```
/*
ipr.c

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "clock.h"

#include "type.h"
#include "assert.h"
#include "buf.h"
#include "event.h"
#include "io.h"
#include "ip_int.h"
#include "ipr.h"

THIS_FILE

#define OROUTE_NR 128
#define OROUTE_STATIC_NR 16
#define OROUTE_HASH_ASS_NR 4
#define OROUTE_HASH_NR 32
#define OROUTE_HASH_MASK (OROUTE_HASH_NR-1)

#define hash_oroute(port_nr, ipaddr, hash_tmp) (hash_tmp= (ipaddr), \
hash_tmp= (hash_tmp >> 20) ^ hash_tmp, \
hash_tmp= (hash_tmp >> 10) ^ hash_tmp, \
hash_tmp= (hash_tmp >> 5) ^ hash_tmp, \
(hash_tmp + (port_nr)) & OROUTE_HASH_MASK)

typedef struct oroute_hash
{
    ipaddr_t orh_addr;
    oroute_t *orh_route;
} oroute_hash_t;

PRIVATE oroute_t oroute_table[OROUTE_NR];
PRIVATE oroute_t *oroute_head;
PRIVATE int static_oroute_nr;
PRIVATE oroute_hash_t oroute_hash_table[OROUTE_HASH_NR][OROUTE_HASH_ASS_NR];

#define IROUTE_NR 512
#define IROUTE_HASH_ASS_NR 4
#define IROUTE_HASH_NR 32
#define IROUTE_HASH_MASK (IROUTE_HASH_NR-1)

#define hash_iroute(port_nr, ipaddr, hash_tmp) (hash_tmp= (ipaddr), \
hash_tmp= (hash_tmp >> 20) ^ hash_tmp, \
hash_tmp= (hash_tmp >> 10) ^ hash_tmp, \
hash_tmp= (hash_tmp >> 5) ^ hash_tmp, \
(hash_tmp + (port_nr)) & IROUTE_HASH_MASK)

typedef struct iroute_hash
{
    ipaddr_t irh_addr;
    iroute_t *irh_route;
} iroute_hash_t;

PRIVATE iroute_t iroute_table[IROUTE_NR];
PRIVATE iroute_hash_t iroute_hash_table[IROUTE_HASH_NR][IROUTE_HASH_ASS_NR];

FORWARD oroute_t *oroute_find_ent ARGS(( int port_nr, ipaddr_t dest ));
FORWARD void oroute_del ARGS(( oroute_t *oroute ));
FORWARD oroute_t *sort_dists ARGS(( oroute_t *oroute ));
FORWARD oroute_t *sort_gws ARGS(( oroute_t *oroute ));
FORWARD void oroute_uncache_nw ARGS(( ipaddr_t dest, ipaddr_t netmask ));
FORWARD void iroute_uncache_nw ARGS(( ipaddr_t dest, ipaddr_t netmask ));

PUBLIC void ipr_init()
{
    int i;
    oroute_t *oroute;
    iroute_t *iroute;
```

```

    for (i= 0, oroute= oroute_table; i<OROUTE_NR; i++, oroute++)
        oroute->ort_flags= ORTF_EMPTY;
    static_oroute_nr= 0;
    assert(OROUTE_HASH_ASS_NR == 4);

    for (i= 0, iroute= iroute_table; i<IROUTE_NR; i++, iroute++)
        iroute->irt_flags= IRTF_EMPTY;
    assert(IROUTE_HASH_ASS_NR == 4);
}

PUBLIC iroute_t *iroute_frag(port_nr, dest)
int port_nr;
ipaddr_t dest;
{
    int hash, i;
    iroute_hash_t *iroute_hash;
    iroute_hash_t tmp_hash;
    iroute_t *iroute, *bestroute;
    unsigned long hash_tmp;
    u32_t tmp_mask;

    hash= hash_iroute(port_nr, dest, hash_tmp);
    iroute_hash= &iroute_hash_table[hash][0];
    if (iroute_hash[0].irh_addr == dest)
        iroute= iroute_hash[0].irh_route;
    else if (iroute_hash[1].irh_addr == dest)
    {
        tmp_hash= iroute_hash[1];
        iroute_hash[1]= iroute_hash[0];
        iroute_hash[0]= tmp_hash;
        iroute= tmp_hash.irh_route;
    }
    else if (iroute_hash[2].irh_addr == dest)
    {
        tmp_hash= iroute_hash[2];
        iroute_hash[2]= iroute_hash[1];
        iroute_hash[1]= iroute_hash[0];
        iroute_hash[0]= tmp_hash;
        iroute= tmp_hash.irh_route;
    }
    else if (iroute_hash[3].irh_addr == dest)
    {
        tmp_hash= iroute_hash[3];
        iroute_hash[3]= iroute_hash[2];
        iroute_hash[2]= iroute_hash[1];
        iroute_hash[1]= iroute_hash[0];
        iroute_hash[0]= tmp_hash;
        iroute= tmp_hash.irh_route;
    }
    else
        iroute= NULL;
    if (iroute)
        return iroute;

    bestroute= NULL;
    for (i= 0, iroute= iroute_table; i < IROUTE_NR; i++, iroute++)
    {
        if (!(iroute->irt_flags & IRTF_INUSE))
            continue;
        if (((dest ^ iroute->irt_dest) & iroute->irt_subnetmask) != 0)
            continue;
        if (!bestroute)
        {
            bestroute= iroute;
            continue;
        }

        /* More specific netmasks are better */
        if (iroute->irt_subnetmask != bestroute->irt_subnetmask)
        {
            /* Using two ntohl macros in one expression
             * is not allowed (tmp_1 is modified twice)

```

```

        */
        tmp_mask= ntohl(iroute->irt_subnetmask);
        if (tmp_mask > ntohl(bestroute->irt_subnetmask))
            bestroute= iroute;
        continue;
    }

    /* Dynamic routes override static routes */
    if ((iroute->irt_flags & IRTF_STATIC) !=
        (bestroute->irt_flags & IRTF_STATIC))
    {
        if (bestroute->irt_flags & IRTF_STATIC)
            bestroute= iroute;
        continue;
    }

    /* A route to the local interface give an opportunity
     * to send redirects.
     */
    if (iroute->irt_port != bestroute->irt_port)
    {
        if (iroute->irt_port == port_nr)
            bestroute= iroute;
        continue;
    }
}
if (bestroute == NULL)
    return NULL;

iroute_hash[3]= iroute_hash[2];
iroute_hash[2]= iroute_hash[1];
iroute_hash[1]= iroute_hash[0];
iroute_hash[0].irh_addr= dest;
iroute_hash[0].irh_route= bestroute;

return bestroute;
}

PUBLIC int oroute_frag(port_nr, dest, ttl, msgsize, nexthop)
int port_nr;
ipaddr_t dest;
int ttl;
size_t msgsize;
ipaddr_t *nexthop;
{
    oroute_t *oroute;

    oroute= oroute_find_ent(port_nr, dest);
    if (!oroute || oroute->ort_dist > ttl)
        return EDSTNOTRCH;
    if (msgsize && oroute->ort_mtu &&
        oroute->ort_mtu < msgsize)
    {
        return EPACKSIZE;
    }

    *nexthop= oroute->ort_gateway;
    return NW_OK;
}

PUBLIC int ipr_add_oroute(port_nr, dest, subnetmask, gateway,
    timeout, dist, mtu, static_route, preference, oroute_p)
int port_nr;
ipaddr_t dest;
ipaddr_t subnetmask;
ipaddr_t gateway;
time_t timeout;
int dist;
int mtu;
int static_route;
i32_t preference;
oroute_t **oroute_p;
{

```

```

int i;
ip_port_t *ip_port;
oroute_t *oroute, *oldest_route, *prev, *nw_route, *gw_route,
        *prev_route;
time_t currtim, exp_tim, exp_tim_orig;

oldest_route= 0;
currtim= get_time();
if (timeout)
    exp_tim= timeout+currtim;
else
    exp_tim= 0;

DBLOCK(0x10,
    printf("ip[%d]: adding oroute to ", port_nr);
    writeIpAddr(dest);
    printf("["); writeIpAddr(subnetmask); printf("] through ");
    writeIpAddr(gateway);
    printf(" timeout: %lds, distance %d, pref %ld, mtu %d\n",
        (long)timeout/HZ, dist, (long)preference, mtu));

ip_port= &ip_port_table[port_nr];

/* Validate gateway */
if (((gateway ^ ip_port->ip_ipaddr) & ip_port->ip_subnetmask) != 0)
{
    DBLOCK(1, printf("ip[%d]: (ipr_add_oroute) invalid gateway: ",
        port_nr); writeIpAddr(gateway); printf("\n"));
    return EINVAL;
}

if (static_route)
{
    if (static_oroute_nr >= OROUTE_STATIC_NR)
        return ENOMEM;
    static_oroute_nr++;
}
else
{
    /* Try to track down any old routes. */
    for(oroute= oroute_head; oroute; oroute= oroute->ort_nextnw)
    {
        if (oroute->ort_port != port_nr)
            continue;
        if (oroute->ort_dest == dest &&
            oroute->ort_subnetmask == subnetmask)
        {
            break;
        }
    }
    for(; oroute; oroute= oroute->ort_nextgw)
    {
        if (oroute->ort_gateway == gateway)
            break;
    }
    for(; oroute; oroute= oroute->ort_nextdist)
    {
        if ((oroute->ort_flags & ORTF_STATIC) != 0)
            continue;
        if (oroute->ort_dist > dist)
            continue;
        break;
    }
    if (oroute)
    {
        assert(oroute->ort_port == port_nr);
        if (dest != 0)
        {
            /* The new expire should not be later
             * than the old expire time. Except for
             * default routes, where the expire time
             * is simple set to the new value.
             */
            exp_tim_orig= oroute->ort_exp_tim;

```

```

        if (!exp_tim)
            exp_tim= exp_tim_orig;
        else if (exp_tim_orig &&
            exp_tim > exp_tim_orig)
        {
            exp_tim= exp_tim_orig;
        }
    }
    oroute_del(oracle);
    oracle->ort_flags= 0;
    oldest_route= oracle;
}

if (oldest_route == NULL)
{
    /* Look for an unused entry, or remove an existing one */
    for (i= 0, oracle= oracle_table; i<OROUTE_NR; i++, oracle++)
    {
        if ((oracle->ort_flags & ORTF_INUSE) == 0)
            break;
        if (oracle->ort_exp_tim && oracle->ort_exp_tim <
            currtim)
        {
            oracle_del(oracle);
            oracle->ort_flags= 0;
            break;
        }
        if (oracle->ort_flags & ORTF_STATIC)
            continue;
        if (oracle->ort_dest == 0)
        {
            /* Never remove default routes. */
            continue;
        }
        if (oldest_route == NULL)
        {
            oldest_route= oracle;
            continue;
        }
        if (oracle->ort_timestamp < oldest_route->ort_timestamp)
        {
            oldest_route= oracle;
        }
    }
    if (i < OROUTE_NR)
        oldest_route= oracle;
    else
    {
        assert(oldest_route);
        oracle_del(oldest_route);
        oldest_route->ort_flags= 0;
    }
}

oldest_route->ort_dest= dest;
oldest_route->ort_gateway= gateway;
oldest_route->ort_subnetmask= subnetmask;
oldest_route->ort_exp_tim= exp_tim;
oldest_route->ort_timestamp= currtim;
oldest_route->ort_dist= dist;
oldest_route->ort_mtu= mtu;
oldest_route->ort_port= port_nr;
oldest_route->ort_flags= ORTF_INUSE;
oldest_route->ort_pref= preference;
if (static_route)
    oldest_route->ort_flags |= ORTF_STATIC;

/* Insert the route by tearing apart the routing table,
 * and insert the entry during the reconstruction.
 */
for (prev= 0, nw_route= oracle_head; nw_route;
    prev= nw_route, nw_route= nw_route->ort_nextnw)
{

```

```

        if (nw_route->ort_port != port_nr)
            continue;
        if (nw_route->ort_dest == dest &&
            nw_route->ort_subnetmask == subnetmask)
        {
            if (prev)
                prev->ort_nextnw= nw_route->ort_nextnw;
            else
                oroute_head= nw_route->ort_nextnw;
            break;
        }
    }
    prev_route= nw_route;
    for(prev= NULL, gw_route= nw_route; gw_route;
        prev= gw_route, gw_route= gw_route->ort_nextgw)
    {
        if (gw_route->ort_gateway == gateway)
        {
            if (prev)
                prev->ort_nextgw= gw_route->ort_nextgw;
            else
                nw_route= gw_route->ort_nextgw;
            break;
        }
    }
    oldest_route->ort_nextdist= gw_route;
    gw_route= oldest_route;
    gw_route= sort_dists(gw_route);
    gw_route->ort_nextgw= nw_route;
    nw_route= gw_route;
    nw_route= sort_gws(nw_route);
    nw_route->ort_nextnw= oroute_head;
    oroute_head= nw_route;
    if (nw_route != prev_route)
        oroute_uncache_nw(nw_route->ort_dest, nw_route->ort_subnetmask);
    if (oroute_p != NULL)
        *oroute_p= oldest_route;
    return NW_OK;
}

```

```

PUBLIC int ipr_del_oroute(port_nr, dest, subnetmask, gateway, static_route)
int port_nr;
ipaddr_t dest;
ipaddr_t subnetmask;
ipaddr_t gateway;
int static_route;
{
    int i;
    oroute_t *oroute;

    for(i= 0, oroute= oroute_table; i<OROUTE_NR; i++, oroute++)
    {
        if ((oroute->ort_flags & ORTF_INUSE) == 0)
            continue;
        if (oroute->ort_port != port_nr ||
            oroute->ort_dest != dest ||
            oroute->ort_subnetmask != subnetmask ||
            oroute->ort_gateway != gateway)
        {
            continue;
        }
        if (!(oroute->ort_flags & ORTF_STATIC) != static_route)
            continue;
        break;
    }

    if (i == OROUTE_NR)
        return ESRCH;

    if (static_route)
        static_oroute_nr--;

    oroute_del(oroute);
    oroute->ort_flags &= ~ORTF_INUSE;
}

```



```
    return NW_OK;
}

PUBLIC void ipr_chk_otab(port_nr, addr, mask)
int port_nr;
ipaddr_t addr;
ipaddr_t mask;
{
    int i;
    oroute_t *oroute;

    DBLOCK(1,
        printf("ip[%d] (ipr_chk_otab): addr ", port_nr);
        writeIpAddr(addr);
        printf(" mask ");
        writeIpAddr(mask);
        printf("\n");
    );

    if (addr == 0)
    {
        /* Special hack to flush entries for an interface that
         * goes down.
         */
        addr= mask= HTONL(0xffffffff);
    }

    for(i= 0, oroute= oroute_table; i<OROUTE_NR; i++, oroute++)
    {
        if ((oroute->ort_flags & ORTF_INUSE) == 0)
            continue;
        if (oroute->ort_port != port_nr ||
            ((oroute->ort_gateway ^ addr) & mask) == 0)
        {
            continue;
        }
        DBLOCK(1, printf("ip[%d] (ipr_chk_otab): deleting route to ",
            port_nr);
            writeIpAddr(oroute->ort_dest);
            printf(" gw ");
            writeIpAddr(oroute->ort_gateway);
            printf("\n"));

        if (oroute->ort_flags & ORTF_STATIC)
            static_oroute_nr--;
        oroute_del(oroute);
        oroute->ort_flags &= ~ORTF_INUSE;
    }
}

PUBLIC void ipr_gateway_down(port_nr, gateway, timeout)
int port_nr;
ipaddr_t gateway;
time_t timeout;
{
    oroute_t *route_ind;
    time_t currtim;
    int i;
    int result;

    currtim= get_time();
    for (i= 0, route_ind= oroute_table; i<OROUTE_NR; i++, route_ind++)
    {
        if (!(route_ind->ort_flags & ORTF_INUSE))
            continue;
        if (route_ind->ort_gateway != gateway)
            continue;
        if (route_ind->ort_exp_tim && route_ind->ort_exp_tim < currtim)
            continue;
        result= ipr_add_oroute(port_nr, route_ind->ort_dest,
            route_ind->ort_subnetmask, gateway,
```

```

        timeout, ORTD_UNREACHABLE, route_ind->ort_mtu,
        FALSE, 0, NULL);
    assert(result == NW_OK);
}
}

PUBLIC void ipr_destunrch(port_nr, dest, netmask, timeout)
int port_nr;
ipaddr_t dest;
ipaddr_t netmask;
time_t timeout;
{
    oroute_t *oroute;
    int result;

    oroute= oroute_find_ent(port_nr, dest);

    if (!oroute)
    {
        DBLOCK(1, printf("ip[%d]: got a dest unreachable for ",
            port_nr);
            writeIpAddr(dest); printf("but no route present\n"));

        return;
    }
    result= ipr_add_oroute(port_nr, dest, netmask, oroute->ort_gateway,
        timeout, ORTD_UNREACHABLE, oroute->ort_mtu, FALSE, 0, NULL);
    assert(result == NW_OK);
}

PUBLIC void ipr_redirect(port_nr, dest, netmask, old_gateway, new_gateway,
    timeout)
int port_nr;
ipaddr_t dest;
ipaddr_t netmask;
ipaddr_t old_gateway;
ipaddr_t new_gateway;
time_t timeout;
{
    oroute_t *oroute;
    ip_port_t *ip_port;
    int result;

    ip_port= &ip_port_table[port_nr];
    oroute= oroute_find_ent(port_nr, dest);

    if (!oroute)
    {
        DBLOCK(1, printf("ip[%d]: got a redirect for ", port_nr);
            writeIpAddr(dest); printf("but no route present\n"));

        return;
    }
    if (oroute->ort_gateway != old_gateway)
    {
        DBLOCK(1, printf("ip[%d]: got a redirect from ", port_nr);
            writeIpAddr(old_gateway); printf(" for ");
            writeIpAddr(dest); printf(" but curr gateway is ");
            writeIpAddr(oroute->ort_gateway); printf("\n"));

        return;
    }
    if ((new_gateway ^ ip_port->ip_ipaddr) & ip_port->ip_subnetmask)
    {
        DBLOCK(1, printf("ip[%d]: redirect from ", port_nr);
            writeIpAddr(old_gateway); printf(" for ");
            writeIpAddr(dest); printf(" but new gateway ");
            writeIpAddr(new_gateway);
            printf(" is not on local subnet\n"));

        return;
    }
    if (oroute->ort_flags & ORTF_STATIC)
    {
        if (oroute->ort_dest == dest)

```

```
        {
            DBLOCK(1, printf("ip[%d]: got a redirect for ",
                             port_nr);
                    writeIpAddr(dest);
                    printf("but route is fixed\n"));
            return;
        }
    }
    else
    {
        result= ipr_add_oroute(port_nr, dest, netmask,
                               oroute->ort_gateway, HZ, ORTD_UNREACHABLE,
                               oroute->ort_mtu, FALSE, 0, NULL);
        assert(result == NW_OK);
    }
    result= ipr_add_oroute(port_nr, dest, netmask, new_gateway,
                           timeout, 1, oroute->ort_mtu, FALSE, 0, NULL);
    assert(result == NW_OK);
}

PUBLIC void ipr_ttl_exc(port_nr, dest, netmask, timeout)
int port_nr;
ipaddr_t dest;
ipaddr_t netmask;
time_t timeout;
{
    oroute_t *oroute;
    int new_dist;
    int result;

    oroute= oroute_find_ent(port_nr, dest);

    if (!oroute)
    {
        DBLOCK(1, printf("ip[%d]: got a ttl exceeded for ",
                         port_nr);
                writeIpAddr(dest); printf("but no route present\n"));
        return;
    }

    new_dist= oroute->ort_dist * 2;
    if (new_dist > IP_DEF_TTL)
    {
        new_dist= oroute->ort_dist+1;
        if (new_dist >= IP_DEF_TTL)
        {
            DBLOCK(1, printf("ip[%d]: got a ttl exceeded for ",
                             port_nr);
                    writeIpAddr(dest);
                    printf("but dist is %d\n",
                           oroute->ort_dist));
            return;
        }
    }

    result= ipr_add_oroute(port_nr, dest, netmask, oroute->ort_gateway,
                           timeout, new_dist, oroute->ort_mtu, FALSE, 0, NULL);
    assert(result == NW_OK);
}

PUBLIC void ipr_mtu(port_nr, dest, mtu, timeout)
int port_nr;
ipaddr_t dest;
ul6_t mtu;
time_t timeout;
{
    oroute_t *oroute;
    int result;

    oroute= oroute_find_ent(port_nr, dest);

    if (!oroute)
    {
```

```

        DBLOCK(1, printf("ip[%d]: got a mtu exceeded for ",
                        port_nr);
                writeIpAddr(dest); printf("but no route present\n"));
        return;
    }

    if (mtu < IP_MIN_MTU)
        return;
    if (oroute->ort_mtu && mtu >= oroute->ort_mtu)
        return; /* Only decrease mtu */

    result= ipr_add_oroute(port_nr, dest, HTONL(0xffffffff),
        oroute->ort_gateway, timeout, oroute->ort_dist, mtu,
        FALSE, 0, NULL);
    assert(result == NW_OK);
}

PUBLIC int ipr_get_oroute(ent_no, route_ent)
int ent_no;
nwio_route_t *route_ent;
{
    oroute_t *oroute;

    if (ent_no<0 || ent_no>= OROUTE_NR)
        return ENOENT;

    oroute= &oroute_table[ent_no];
    if ((oroute->ort_flags & ORTF_INUSE) && oroute->ort_exp_tim &&
        oroute->ort_exp_tim < get_time())
    {
        oroute_del(oroute);
        oroute->ort_flags &= ~ORTF_INUSE;
    }

    route_ent->nwr_ent_no= ent_no;
    route_ent->nwr_ent_count= OROUTE_NR;
    route_ent->nwr_dest= oroute->ort_dest;
    route_ent->nwr_netmask= oroute->ort_subnetmask;
    route_ent->nwr_gateway= oroute->ort_gateway;
    route_ent->nwr_dist= oroute->ort_dist;
    route_ent->nwr_flags= NWRF_EMPTY;
    if (oroute->ort_flags & ORTF_INUSE)
    {
        route_ent->nwr_flags |= NWRF_INUSE;
        if (oroute->ort_flags & ORTF_STATIC)
            route_ent->nwr_flags |= NWRF_STATIC;
    }
    route_ent->nwr_pref= oroute->ort_pref;
    route_ent->nwr_mtu= oroute->ort_mtu;
    route_ent->nwr_ifaddr= ip_get_ifaddr(oroute->ort_port);
    return NW_OK;
}

PRIVATE oroute_t *oroute_find_ent(port_nr, dest)
int port_nr;
ipaddr_t dest;
{
    int hash;
    oroute_hash_t *oroute_hash;
    oroute_hash_t tmp_hash;
    oroute_t *oroute, *bestroute;
    time_t currtim;
    unsigned long hash_tmp;
    u32_t tmp_mask;

    currtim= get_time();

    hash= hash_oroute(port_nr, dest, hash_tmp);
    oroute_hash= &oroute_hash_table[hash][0];
    if (oroute_hash[0].orh_addr == dest)
        oroute= oroute_hash[0].orh_route;
    else if (oroute_hash[1].orh_addr == dest)

```

```

    {
        tmp_hash= oroute_hash[1];
        oroute_hash[1]= oroute_hash[0];
        oroute_hash[0]= tmp_hash;
        oroute= tmp_hash.orh_route;
    }
    else if (oroute_hash[2].orh_addr == dest)
    {
        tmp_hash= oroute_hash[2];
        oroute_hash[2]= oroute_hash[1];
        oroute_hash[1]= oroute_hash[0];
        oroute_hash[0]= tmp_hash;
        oroute= tmp_hash.orh_route;
    }
    else if (oroute_hash[3].orh_addr == dest)
    {
        tmp_hash= oroute_hash[3];
        oroute_hash[3]= oroute_hash[2];
        oroute_hash[2]= oroute_hash[1];
        oroute_hash[1]= oroute_hash[0];
        oroute_hash[0]= tmp_hash;
        oroute= tmp_hash.orh_route;
    }
    else
        oroute= NULL;
    if (oroute)
    {
        assert(oroute->ort_port == port_nr);
        if (oroute->ort_exp_tim && oroute->ort_exp_tim<currtim)
        {
            oroute_del(oroute);
            oroute->ort_flags &= ~ORTF_INUSE;
        }
        else
            return oroute;
    }

    bestroute= NULL;
    for (oroute= oroute_head; oroute; oroute= oroute->ort_nextnw)
    {
        if (((dest ^ oroute->ort_dest) & oroute->ort_subnetmask) != 0)
            continue;
        if (oroute->ort_port != port_nr)
            continue;
        if (!bestroute)
        {
            bestroute= oroute;
            continue;
        }
        assert(oroute->ort_dest != bestroute->ort_dest);
        /* Using two ntohl macros in one expression
         * is not allowed (tmp_l is modified twice)
         */
        tmp_mask= ntohl(oroute->ort_subnetmask);
        if (tmp_mask > ntohl(bestroute->ort_subnetmask))
        {
            bestroute= oroute;
            continue;
        }
    }
    if (bestroute == NULL)
        return NULL;

    oroute_hash[3]= oroute_hash[2];
    oroute_hash[2]= oroute_hash[1];
    oroute_hash[1]= oroute_hash[0];
    oroute_hash[0].orh_addr= dest;
    oroute_hash[0].orh_route= bestroute;

    return bestroute;
}

```

```
PRIVATE void oroute_del(oroute)
```

```

oroute_t *oroute;
{
    oroute_t *prev, *nw_route, *gw_route, *dist_route, *prev_route;

    DBLOCK(0x10,
        printf("ip[%d]: deleting oroute to ", oroute->ort_port);
        writeIpAddr(oroute->ort_dest);
        printf("["); writeIpAddr(oroute->ort_subnetmask);
        printf("] through ");
        writeIpAddr(oroute->ort_gateway);
        printf(
            " timestamp %lds, timeout: %lds, distance %d pref %ld mtu %ld ",
                (long)oroute->ort_timestamp/HZ,
                (long)oroute->ort_exp_tim/HZ, oroute->ort_dist,
                (long)oroute->ort_pref, (long)oroute->ort_mtu);
        printf("flags 0x%x\n", oroute->ort_flags));

    for (prev= NULL, nw_route= oroute_head; nw_route;
        prev= nw_route, nw_route= nw_route->ort_nextnw)
    {
        if (oroute->ort_port == nw_route->ort_port &&
            oroute->ort_dest == nw_route->ort_dest &&
            oroute->ort_subnetmask == nw_route->ort_subnetmask)
        {
            break;
        }
    }
    assert(nw_route);
    if (prev)
        prev->ort_nextnw= nw_route->ort_nextnw;
    else
        oroute_head= nw_route->ort_nextnw;
    prev_route= nw_route;
    for (prev= NULL, gw_route= nw_route; gw_route;
        prev= gw_route, gw_route= gw_route->ort_nextgw)
    {
        if (oroute->ort_gateway == gw_route->ort_gateway)
            break;
    }
    assert(gw_route);
    if (prev)
        prev->ort_nextgw= gw_route->ort_nextgw;
    else
        nw_route= gw_route->ort_nextgw;
    for (prev= NULL, dist_route= gw_route; dist_route;
        prev= dist_route, dist_route= dist_route->ort_nextdist)
    {
        if (oroute == dist_route)
            break;
    }
    assert(dist_route);
    if (prev)
        prev->ort_nextdist= dist_route->ort_nextdist;
    else
        gw_route= dist_route->ort_nextdist;
    gw_route= sort_dists(gw_route);
    if (gw_route != NULL)
    {
        gw_route->ort_nextgw= nw_route;
        nw_route= gw_route;
    }
    nw_route= sort_gws(nw_route);
    if (nw_route != NULL)
    {
        nw_route->ort_nextnw= oroute_head;
        oroute_head= nw_route;
    }
    if (nw_route != prev_route)
    {
        oroute_uncache_nw(prev_route->ort_dest,
            prev_route->ort_subnetmask);
    }
}

```

```
PRIVATE oroute_t *sort_dists(oroute)
oroute_t *oroute;
{
    oroute_t *r, *prev, *best, *best_prev;
    int best_dist, best_pref;

    best= NULL;
    best_dist= best_pref= 0;
    best_prev= NULL;
    for (prev= NULL, r= oroute; r; prev= r, r= r->ort_nextdist)
    {
        if (best == NULL)
            ; /* Force assignment to best */
        else if (r->ort_dist != best_dist)
        {
            if (r->ort_dist > best_dist)
                continue;
        }
        else
        {
            if (r->ort_pref <= best_pref)
                continue;
        }
        best= r;
        best_prev= prev;
        best_dist= r->ort_dist;
        best_pref= r->ort_pref;
    }
    if (!best)
    {
        assert(oroute == NULL);
        return oroute;
    }
    if (!best_prev)
    {
        assert(best == oroute);
        return oroute;
    }
    best_prev->ort_nextdist= best->ort_nextdist;
    best->ort_nextdist= oroute;
    return best;
}
```

```
PRIVATE oroute_t *sort_gws(oroute)
oroute_t *oroute;
{
    oroute_t *r, *prev, *best, *best_prev;
    int best_dist, best_pref;

    best= NULL;
    best_dist= best_pref= 0;
    best_prev= NULL;
    for (prev= NULL, r= oroute; r; prev= r, r= r->ort_nextgw)
    {
        if (best == NULL)
            ; /* Force assignment to best */
        else if (r->ort_dist != best_dist)
        {
            if (r->ort_dist > best_dist)
                continue;
        }
        else
        {
            if (r->ort_pref <= best_pref)
                continue;
        }
        best= r;
        best_prev= prev;
        best_dist= r->ort_dist;
        best_pref= r->ort_pref;
    }
    if (!best)
```

```

    {
        assert(oroute == NULL);
        return oroute;
    }
    if (!best_prev)
    {
        assert(best == oroute);
        return oroute;
    }
    best_prev->ort_nextgw= best->ort_nextgw;
    best->ort_nextgw= oroute;
    return best;
}

PRIVATE void oroute_uncache_nw(dest, netmask)
ipaddr_t dest;
ipaddr_t netmask;
{
    int i, j;
    oroute_hash_t *oroute_hash;

    for (i= 0, oroute_hash= &oroute_hash_table[0][0];
         i<OROUTE_HASH_NR; i++, oroute_hash += OROUTE_HASH_ASS_NR)
    {
        for (j= 0; j<OROUTE_HASH_ASS_NR; j++)
        {
            if (((oroute_hash[j].orh_addr ^ dest) & netmask) == 0)
            {
                oroute_hash[j].orh_addr= 0;
                oroute_hash[j].orh_route= NULL;
            }
        }
    }
}

/*
 * Input routing
 */

PUBLIC int ipr_get_iroute(ent_no, route_ent)
int ent_no;
nwio_route_t *route_ent;
{
    iroute_t *iroute;

    if (ent_no<0 || ent_no>= IROUTE_NR)
        return ENOENT;

    iroute= &iroute_table[ent_no];

    route_ent->nwr_ent_no= ent_no;
    route_ent->nwr_ent_count= IROUTE_NR;
    route_ent->nwr_dest= iroute->irt_dest;
    route_ent->nwr_netmask= iroute->irt_subnetmask;
    route_ent->nwr_gateway= iroute->irt_gateway;
    route_ent->nwr_dist= iroute->irt_dist;
    route_ent->nwr_flags= NWRF_EMPTY;
    if (iroute->irt_flags & IRTF_INUSE)
    {
        route_ent->nwr_flags |= NWRF_INUSE;
        if (iroute->irt_flags & IRTF_STATIC)
            route_ent->nwr_flags |= NWRF_STATIC;
        if (iroute->irt_dist == IRTD_UNREACHABLE)
            route_ent->nwr_flags |= NWRF_UNREACHABLE;
    }
    route_ent->nwr_pref= 0;
    route_ent->nwr_mtu= iroute->irt_mtu;
    route_ent->nwr_ifaddr= ip_get_ifaddr(iroute->irt_port);
    return NW_OK;
}

```



```

PUBLIC int ipr_add_iroute(port_nr, dest, subnetmask, gateway,
                          dist, mtu, static_route, iroute_p)
int port_nr;
ipaddr_t dest;
ipaddr_t subnetmask;
ipaddr_t gateway;
int dist;
int mtu;
int static_route;
iroute_t **iroute_p;
{
    int i;
    iroute_t *iroute, *unused_route;
    ip_port_t *ip_port;

    ip_port = &ip_port_table[port_nr];

    /* Check gateway */
    if (((gateway ^ ip_port->ip_ipaddr) & ip_port->ip_subnetmask) != 0 &&
        gateway != 0)
    {
        DBLOCK(1, printf("ip[%d] (ipr_add_iroute): invalid gateway: ",
                        port_nr);
                writeIpAddr(gateway); printf("\n"));
        return EINVAL;
    }

    unused_route = NULL;
    if (static_route)
    {
        /* Static routes are not reused automatically, so we look
         * for an unused entry.
         */
        for(i = 0, iroute = iroute_table; i < IROUTE_NR; i++, iroute++)
        {
            if ((iroute->irt_flags & IRTF_INUSE) == 0)
                break;
        }
        if (i != IROUTE_NR)
            unused_route = iroute;
    }
    else
    {
        /* Try to track down any old routes, and look for an
         * unused one.
         */
        for(i = 0, iroute = iroute_table; i < IROUTE_NR; i++, iroute++)
        {
            if ((iroute->irt_flags & IRTF_INUSE) == 0)
            {
                unused_route = iroute;
                continue;
            }
            if ((iroute->irt_flags & IRTF_STATIC) != 0)
                continue;
            if (iroute->irt_port != port_nr ||
                iroute->irt_dest != dest ||
                iroute->irt_subnetmask != subnetmask ||
                iroute->irt_gateway != gateway)
            {
                continue;
            }
            break;
        }
        if (i != IROUTE_NR)
            unused_route = iroute;
    }

    if (unused_route == NULL)
        return ENOMEM;
    iroute = unused_route;

    iroute->irt_port = port_nr;
    iroute->irt_dest = dest;

```

```

    iroute->irt_subnetmask= subnetmask;
    iroute->irt_gateway= gateway;
    iroute->irt_dist= dist;
    iroute->irt_mtu= mtu;
    iroute->irt_flags= IRTF_INUSE;
    if (static_route)
        iroute->irt_flags |= IRTF_STATIC;

    iroute_uncache_nw(iroute->irt_dest, iroute->irt_subnetmask);
    if (iroute_p != NULL)
        *iroute_p= iroute;
    return NW_OK;
}

PUBLIC int ipr_del_iroute(port_nr, dest, subnetmask, gateway, static_route)
int port_nr;
ipaddr_t dest;
ipaddr_t subnetmask;
ipaddr_t gateway;
int static_route;
{
    int i;
    iroute_t *iroute;

    /* Try to track down any old routes, and look for an
     * unused one.
     */
    for(i= 0, iroute= iroute_table; i<IROUTE_NR; i++, iroute++)
    {
        if ((iroute->irt_flags & IRTF_INUSE) == 0)
            continue;
        if (iroute->irt_port != port_nr ||
            iroute->irt_dest != dest ||
            iroute->irt_subnetmask != subnetmask ||
            iroute->irt_gateway != gateway)
        {
            continue;
        }
        if (!(iroute->irt_flags & IRTF_STATIC) != static_route)
            continue;
        break;
    }

    if (i == IROUTE_NR)
        return ESRCH;

    iroute_uncache_nw(iroute->irt_dest, iroute->irt_subnetmask);
    iroute->irt_flags= IRTF_EMPTY;
    return NW_OK;
}

PUBLIC void ipr_chk_itab(port_nr, addr, mask)
int port_nr;
ipaddr_t addr;
ipaddr_t mask;
{
    int i;
    iroute_t *iroute;

    DBLOCK(1,
        printf("ip[%d] (ipr_chk_itab): addr ", port_nr);
        writeIpAddr(addr);
        printf(" mask ");
        writeIpAddr(mask);
        printf("\n");
    );

    if (addr == 0)
    {
        /* Special hack to flush entries for an interface that
         * goes down.
         */

```

```

        addr= mask= HTONL(0xffffffff);
    }

    for(i= 0, iroute= iroute_table; i<IROUTE_NR; i++, iroute++)
    {
        if ((iroute->irt_flags & IRTF_INUSE) == 0)
            continue;
        if (iroute->irt_port != port_nr)
            continue;
        if (iroute->irt_gateway == 0)
        {
            /* Special case: attached network. */
            if (iroute->irt_subnetmask == mask &&
                iroute->irt_dest == (addr & mask))
            {
                /* Nothing changed. */
                continue;
            }
        }
        if (((iroute->irt_gateway ^ addr) & mask) == 0)
            continue;

        DBLOCK(1, printf("ip[%d] (ipr_chk_itab): deleting route to ",
                        port_nr);
        writeIpAddr(iroute->irt_dest);
        printf(" gw ");
        writeIpAddr(iroute->irt_gateway);
        printf("\n"));

        iroute_uncache_nw(iroute->irt_dest, iroute->irt_subnetmask);
        iroute->irt_flags &= ~IRTF_INUSE;
    }
}

```

```

PRIVATE void iroute_uncache_nw(dest, netmask)
ipaddr_t dest;
ipaddr_t netmask;
{
    int i, j;
    iroute_hash_t *iroute_hash;

    for (i= 0, iroute_hash= &iroute_hash_table[0][0];
        i<IROUTE_HASH_NR; i++, iroute_hash += IROUTE_HASH_ASS_NR)
    {
        for (j= 0; j<IROUTE_HASH_ASS_NR; j++)
        {
            if (((iroute_hash[j].irh_addr ^ dest) &
                netmask) == 0)
            {
                iroute_hash[j].irh_addr= 0;
                iroute_hash[j].irh_route= NULL;
            }
        }
    }
}

```

```

/*
 * $PchId: ipr.c,v 1.23 2003/01/22 11:49:58 philip Exp $
 */

```

```

/*
ipr.h

Copyright 1995 Philip Homburg
*/

#ifndef IPR_H
#define IPR_H

typedef struct oroute
{
    int ort_port;
    ipaddr_t ort_dest;
    ipaddr_t ort_subnetmask;
    int ort_dist;
    i32_t ort_pref;
    u32_t ort_mtu;
    ipaddr_t ort_gateway;
    time_t ort_exp_tim;
    time_t ort_timestamp;
    int ort_flags;

    struct oroute *ort_nextnw;
    struct oroute *ort_nextgw;
    struct oroute *ort_nextdist;
} oroute_t;

#define ORTD_UNREACHABLE          512

#define ORTF_EMPTY                0
#define ORTF_INUSE                1
#define ORTF_STATIC              2

typedef struct iroute
{
    ipaddr_t irt_dest;
    ipaddr_t irt_gateway;
    ipaddr_t irt_subnetmask;
    int irt_dist;
    u32_t irt_mtu;
    int irt_port;
    int irt_flags;
} iroute_t;

#define IRTD_UNREACHABLE          512

#define IRTF_EMPTY                0
#define IRTF_INUSE                1
#define IRTF_STATIC              2

#define IPR_UNRCH_TIMEOUT        (60L * HZ)
#define IPR_TTL_TIMEOUT          (60L * HZ)
#define IPR_REDIRECT_TIMEOUT     (20 * 60L * HZ)
#define IPR_GW_DOWN_TIMEOUT      (60L * HZ)
#define IPR_MTU_TIMEOUT          (10*60L * HZ) /* RFC-1191 */

/* Prototypes */

iroute_t *iroute_frag ARGS(( int port_nr, ipaddr_t dest ));
int oroute_frag ARGS(( int port_nr, ipaddr_t dest, int ttl, size_t msgsize,
                      ipaddr_t *nexthop ));

void ipr_init ARGS(( void ));
int ipr_get_iroute ARGS(( int ent_no, nwio_route_t *route_ent ));
int ipr_add_iroute ARGS(( int port_nr, ipaddr_t dest, ipaddr_t subnetmask,
                        ipaddr_t gateway, int dist, int mtu, int static_route,
                        iroute_t **route_p ));
int ipr_del_iroute ARGS(( int port_nr, ipaddr_t dest, ipaddr_t subnetmask,
                        ipaddr_t gateway, int static_route ));
void ipr_chk_itab ARGS(( int port_nr, ipaddr_t addr, ipaddr_t mask ));
int ipr_get_oroute ARGS(( int ent_no, nwio_route_t *route_ent ));
int ipr_add_oroute ARGS(( int port_nr, ipaddr_t dest, ipaddr_t subnetmask,
                        ipaddr_t gateway, time_t timeout, int dist, int mtu, int static_route,
                        i32_t preference, oroute_t **route_p ));
int ipr_del_oroute ARGS(( int port_nr, ipaddr_t dest, ipaddr_t subnetmask,

```

```
        ipaddr_t gateway, int static_route ));
void ipr_chk_otab ARGS(( int port_nr, ipaddr_t addr, ipaddr_t mask ));
void ipr_gateway_down ARGS(( int port_nr, ipaddr_t gateway, time_t timeout ));
void ipr_redirect ARGS(( int port_nr, ipaddr_t dest, ipaddr_t subnetmask,
        ipaddr_t old_gateway, ipaddr_t new_gateway, time_t timeout ));
void ipr_destunrch ARGS(( int port_nr, ipaddr_t dest, ipaddr_t subnetmask,
        time_t timeout ));
void ipr_ttl_exc ARGS(( int port_nr, ipaddr_t dest, ipaddr_t subnetmask,
        time_t timeout ));
void ipr_mtu ARGS(( int port_nr, ipaddr_t dest, U16_t mtu, time_t timeout ));

#endif /* IPR_H */

/*
 * $PchId: ipr.h,v 1.8 2002/06/09 07:48:11 philip Exp $
 */
```

```
/*
generic/psip.c

Implementation of a pseudo IP device.

Created:      Apr 22, 1993 by Philip Homburg

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "assert.h"
#include "buf.h"
#include "event.h"
#include "type.h"
#include "ip_int.h"
#include "psip.h"
#include "sr.h"

THIS_FILE

typedef struct psip_port
{
    int pp_flags;
    int pp_ipdev;
    int pp_opencnt;
    struct psip_fd *pp_rd_head;
    struct psip_fd *pp_rd_tail;
    acc_t *pp_promisc_head;
    acc_t *pp_promisc_tail;
} psip_port_t;

#define PPF_EMPTY      0
#define PPF_CONFIGURED 1
#define PPF_ENABLED    2
#define PPF_PROMISC    4

#define PSIP_FD_NR      (1*IP_PORT_MAX)

typedef struct psip_fd
{
    int pf_flags;
    int pf_srfd;
    psip_port_t *pf_port;
    get_userdata_t pf_get_userdata;
    put_userdata_t pf_put_userdata;
    struct psip_fd *pf_rd_next;
    size_t pf_rd_count;
    nwio_psipt_t pf_psipt;
} psip_fd_t;

#define PPF_EMPTY      0
#define PPF_INUSE      1
#define PPF_READ_IP    2
#define PPF_PROMISC    4
#define PPF_NEXTHOP    8

PRIVATE psip_port_t *psip_port_table;
PRIVATE psip_fd_t psip_fd_table[PSIP_FD_NR];

FORWARD int psip_open ARGS(( int port, int srfd,
    get_userdata_t get_userdata, put_userdata_t put_userdata,
    put_pkt_t pkt_pkt, select_res_t select_res ));
FORWARD int psip_ioctl ARGS(( int fd, ioreq_t req ));
FORWARD int psip_read ARGS(( int fd, size_t count ));
FORWARD int psip_write ARGS(( int fd, size_t count ));
FORWARD int psip_select ARGS(( int port_nr, unsigned operations ));
FORWARD void psip_close ARGS(( int fd ));
FORWARD int psip_cancel ARGS(( int fd, int which_operation ));
FORWARD void psip_restart_read ARGS(( psip_port_t *psip_port ));
FORWARD int psip_setopt ARGS(( psip_fd_t *psip_fd, nwio_psipt_t *newopt ));
FORWARD void psip_buffree ARGS(( int priority ));
FORWARD void check_promisc ARGS(( psip_port_t *psip_port ));
#ifdef BUF_CONSISTENCY_CHECK
```

```

FORWARD void psip_bufcheck ARGS(( void ));
#endif
FORWARD void reply_thr_put ARGS(( psip_fd_t *psip_fd, int reply,
    int for_ioctl ));
FORWARD void reply_thr_get ARGS(( psip_fd_t *psip_fd, int reply,
    int for_ioctl ));

PUBLIC void psip_prep()
{
    psip_port_table= alloc(psip_conf_nr * sizeof(psip_port_table[0]));
}

PUBLIC void psip_init()
{
    int i;
    psip_port_t *psip_port;
    psip_fd_t *psip_fd;

    for (i=0, psip_port= psip_port_table; i<psip_conf_nr; i++, psip_port++)
        psip_port->pp_flags= PPF_EMPTY;

    for (i=0, psip_fd= psip_fd_table; i<PSIP_FD_NR; i++, psip_fd++)
        psip_fd->pf_flags= PPF_EMPTY;

    for (i=0, psip_port= psip_port_table; i<psip_conf_nr; i++, psip_port++)
    {
        psip_port->pp_flags |= PPF_CONFIGURED;
        psip_port->pp_opencnt= 0;
        psip_port->pp_rd_head= NULL;
        psip_port->pp_promisc_head= NULL;
    }

#ifndef BUF_CONSISTENCY_CHECK
    bf_logon(psip_buffree);
#else
    bf_logon(psip_buffree, psip_bufcheck);
#endif
}

PUBLIC int psip_enable(port_nr, ip_port_nr)
int port_nr;
int ip_port_nr;
{
    psip_port_t *psip_port;

    assert(port_nr >= 0);
    if (port_nr >= psip_conf_nr)
        return -1;

    psip_port= &psip_port_table[port_nr];
    if (!(psip_port->pp_flags &PPF_CONFIGURED))
        return -1;

    psip_port->pp_ipdev= ip_port_nr;
    psip_port->pp_flags |= PPF_ENABLED;

    sr_add_minor(if2minor(psip_conf[port_nr].pc_ifno, PSIP_DEV_OFF),
        port_nr, psip_open, psip_close, psip_read,
        psip_write, psip_ioctl, psip_cancel, psip_select);

    return NW_OK;
}

PUBLIC int psip_send(port_nr, dest, pack)
int port_nr;
ipaddr_t dest;
acc_t *pack;
{
    psip_port_t *psip_port;
    psip_fd_t *psip_fd, *mark_fd;
    int i, result, result1;
    size_t buf_size, extrasize;
    acc_t *hdr_pack, *acc;
    psip_io_hdr_t *hdr;

```

```

assert(port_nr >= 0 && port_nr < psip_conf_nr);
psip_port= &psip_port_table[port_nr];

if (psip_port->pp_opencnt == 0)
{
    bf_afree(pack);
    return NW_OK;
}

for(;;)
{
    mark_fd= psip_port->pp_rd_tail;

    for(i= 0; i<PSIP_FD_NR; i++)
    {
        psip_fd= psip_port->pp_rd_head;
        if (!psip_fd)
            return NW_SUSPEND;
        psip_port->pp_rd_head= psip_fd->pf_rd_next;
        if (!(psip_fd->pf_flags & PFF_PROMISC))
            break;
        psip_fd->pf_rd_next= NULL;
        if (psip_port->pp_rd_head == NULL)
            psip_port->pp_rd_head= psip_fd;
        else
            psip_port->pp_rd_tail->pf_rd_next= psip_fd;
        psip_port->pp_rd_tail= psip_fd;
        if (psip_fd == mark_fd)
            return NW_SUSPEND;
    }
    if (i == PSIP_FD_NR)
        ip_panic(( "psip_send: loop" ));

    assert(psip_fd->pf_flags & PFF_READ_IP);
    psip_fd->pf_flags &= ~PFF_READ_IP;

    if (psip_fd->pf_flags & PFF_NEXTHOP)
        extrasize= sizeof(dest);
    else
        extrasize= 0;

    buf_size= bf_bufsize(pack);
    if (buf_size+extrasize <= psip_fd->pf_rd_count)
    {
        if (psip_port->pp_flags & PPF_PROMISC)
        {
            /* Deal with promiscuous mode. */
            hdr_pack= bf_memreq(sizeof(*hdr));
            hdr= (psip_io_hdr_t *)ptr2acc_data(hdr_pack);
            memset(hdr, '\0', sizeof(*hdr));
            hdr->pih_flags |= PF_LOC2REM;
            hdr->pih_nexthop= dest;

            pack->acc_linkC++;
            hdr_pack->acc_next= pack;
            hdr_pack->acc_ext_link= NULL;
            if (psip_port->pp_promisc_head)
            {
                /* Append at the end. */
                psip_port->pp_promisc_tail->
                    acc_ext_link= hdr_pack;
                psip_port->pp_promisc_tail= hdr_pack;
            }
            else
            {
                /* First packet. */
                psip_port->pp_promisc_head= hdr_pack;
                psip_port->pp_promisc_tail= hdr_pack;
                if (psip_port->pp_rd_head)
                    promisc_restart_read(psip_port);
            }
        }
    }
}

```



```

        if (extrasize)
        {
            /* Prepend nexthop address */
            acc= bf_memreq(sizeof(dest));
            *(ipaddr_t *) (ptr2acc_data(acc))= dest;
            acc->acc_next= pack;
            pack= acc; acc= NULL;
            buf_size += extrasize;
        }

        result= (*psip_fd->pf_put_userdata)(psip_fd->pf_srfd,
            (size_t)0, pack, FALSE);
        if (result == NW_OK)
            result= buf_size;
    }
    else
        result= EPACKSIZE;

    result1= (*psip_fd->pf_put_userdata)(psip_fd->pf_srfd,
        (size_t)result, NULL, FALSE);
    assert(result1 == NW_OK);
    if (result == EPACKSIZE)
        continue;
    return NW_OK;
}
return NW_SUSPEND;
}

PRIVATE int psip_open(port, srfd, get_userdata, put_userdata, put_pkt,
    select_res)
int port;
int srfd;
get_userdata_t get_userdata;
put_userdata_t put_userdata;
put_pkt_t put_pkt;
select_res_t select_res;
{
    psip_port_t *psip_port;
    psip_fd_t *psip_fd;
    int i;

    assert(port >= 0 && port < psip_conf_nr);
    psip_port= &psip_port_table[port];

    if (!(psip_port->pp_flags & PPF_ENABLED))
        return ENXIO;

    for (i= 0, psip_fd= psip_fd_table; i<PSIP_FD_NR; i++, psip_fd++)
    {
        if (psip_fd->pf_flags & PFF_INUSE)
            continue;
        break;
    }
    if (i == PSIP_FD_NR)
        return ENFILE;
    psip_fd->pf_flags |= PFF_INUSE;
    psip_fd->pf_srfd= srfd;
    psip_fd->pf_port= psip_port;
    psip_fd->pf_get_userdata= get_userdata;
    psip_fd->pf_put_userdata= put_userdata;
    psip_port->pp_opencnt++;

    return i;
}

PRIVATE int psip_ioctl(fd, req)
int fd;
ioreq_t req;
{
    int result;
    psip_fd_t *psip_fd;
    acc_t *data;
    nwio_ipconf_t *ipconfp;
    nwio_psipopt_t *psip_opt, *newoptp;

```

```
assert(fd >= 0 && fd < PSIP_FD_NR);
psip_fd= &psip_fd_table[fd];

switch(req)
{
case NWIOSIPCONF:
    data= (*psip_fd->pf_get_userdata)(psip_fd->pf_srfd, 0,
                                     sizeof(*ipconfp), TRUE);
    if (!data)
    {
        result= EFAULT;
        break;
    }
    data= bf_packIffLess(data, sizeof(*ipconfp));
    assert (data->acc_length == sizeof(*ipconfp));

    ipconfp= (nwio_ipconf_t *)ptr2acc_data(data);
    result= ip_setconf(psip_fd->pf_port->pp_ipdev, ipconfp);
    bf_afree(data);
    reply_thr_get(psip_fd, result, TRUE);
    break;
case NWIOSPSILOPT:
    data= (*psip_fd->pf_get_userdata)(psip_fd->pf_srfd, 0,
                                     sizeof(*psip_opt), TRUE);
    if (!data)
    {
        result= EFAULT;
        break;
    }
    data= bf_packIffLess(data, sizeof(*psip_opt));
    assert (data->acc_length == sizeof(*psip_opt));

    newoptp= (nwio_psipt_t *)ptr2acc_data(data);
    result= psip_setopt(psip_fd, newoptp);
    bf_afree(data);
    if (result == NW_OK)
    {
        if (psip_fd->pf_psipt.nwpo_flags & NWPO_EN_PROMISC)
        {
            psip_fd->pf_flags |= PFF_PROMISC;
            psip_fd->pf_port->pp_flags |= PFF_PROMISC;
        }
        else
        {
            psip_fd->pf_flags &= ~PFF_PROMISC;
            check_promisc(psip_fd->pf_port);
        }
        if (psip_fd->pf_psipt.nwpo_flags & NWPO_EN_NEXTHOP)
        {
            psip_fd->pf_flags |= PFF_NEXTHOP;
        }
        else
        {
            psip_fd->pf_flags &= ~PFF_NEXTHOP;
        }
    }
    reply_thr_get(psip_fd, result, TRUE);
    break;
case NWIOGPSILOPT:
    data= bf_memreq(sizeof(*psip_opt));
    psip_opt= (nwio_psipt_t *)ptr2acc_data(data);

    *psip_opt= psip_fd->pf_psipt;
    result= (*psip_fd->pf_put_userdata)(psip_fd->pf_srfd, 0,
                                     data, TRUE);
    if (result == NW_OK)
        reply_thr_put(psip_fd, NW_OK, TRUE);
    break;
default:
    reply_thr_put(psip_fd, ENOTTY, TRUE);
    break;
}
return NW_OK;
```

```

}

PRIVATE int psip_read(fd, count)
int fd;
size_t count;
{
    psip_port_t *psip_port;
    psip_fd_t *psip_fd;
    acc_t *pack;
    size_t buf_size;
    int result, result1;

    assert(fd >= 0 && fd < PSIP_FD_NR);
    psip_fd= &psip_fd_table[fd];
    psip_port= psip_fd->pf_port;

    if ((psip_fd->pf_flags & PFF_PROMISC) && psip_port->pp_promisc_head)
    {
        /* Deliver a queued packet. */
        pack= psip_port->pp_promisc_head;
        buf_size= bf_bufsize(pack);
        if (buf_size <= count)
        {
            psip_port->pp_promisc_head= pack->acc_ext_link;
            result= (*psip_fd->pf_put_userdata)(psip_fd->pf_srfd,
                (size_t)0, pack, FALSE);
            if (result == NW_OK)
                result= buf_size;
        }
        else
            result= EPACKSIZE;

        result1= (*psip_fd->pf_put_userdata)(psip_fd->pf_srfd,
            (size_t)result, NULL, FALSE);
        assert(result1 == NW_OK);
        return NW_OK;
    }

    psip_fd->pf_rd_count= count;
    if (psip_port->pp_rd_head == NULL)
        psip_port->pp_rd_head= psip_fd;
    else
        psip_port->pp_rd_tail->pf_rd_next= psip_fd;
    psip_fd->pf_rd_next= NULL;
    psip_port->pp_rd_tail= psip_fd;

    psip_fd->pf_flags |= PFF_READ_IP;
    if (!(psip_fd->pf_flags & PFF_PROMISC))
        ipps_get(psip_port->pp_ipdev);
    if (psip_fd->pf_flags & PFF_READ_IP)
        return NW_SUSPEND;
    return NW_OK;
}

PRIVATE int psip_write(fd, count)
int fd;
size_t count;
{
    psip_port_t *psip_port;
    psip_fd_t *psip_fd;
    acc_t *pack, *hdr_pack;
    psip_io_hdr_t *hdr;
    size_t pack_len;
    ipaddr_t nexthop;

    assert(fd >= 0 && fd < PSIP_FD_NR);
    psip_fd= &psip_fd_table[fd];
    psip_port= psip_fd->pf_port;

    pack= (*psip_fd->pf_get_userdata)(psip_fd->pf_srfd, (size_t)0,
        count, FALSE);
    if (pack == NULL)
    {
        pack= (*psip_fd->pf_get_userdata)(psip_fd->pf_srfd,

```

```

        (size_t)EFAULT, (size_t)0, FALSE);
    assert(pack == NULL);
    return NW_OK;
}

if (psip_fd->pf_flags & PFF_NEXTHOP)
{
    pack_len= bf_bufsize(pack);
    if (pack_len <= sizeof(nexthop))
    {
        /* Something strange */
        bf_afree(pack); pack= NULL;
        pack= (*psip_fd->pf_get_userdata)(psip_fd->pf_srfd,
            (size_t)EPACKSIZE, (size_t)0, FALSE);
        assert(pack == NULL);
        return NW_OK;
    }
    pack= bf_packIfLess(pack, sizeof(nexthop));
    nexthop= *(ipaddr_t *)ptr2acc_data(pack);
    pack= bf_delhead(pack, sizeof(nexthop));

    /* Map multicast to broadcast */
    if ((nexthop & HTONL(0xE0000000)) == HTONL(0xE0000000))
        nexthop= HTONL(0xffffffff);
}
else
{
    /* Assume point to point */
    nexthop= HTONL(0x00000000);
}

if (psip_port->pp_flags & PPF_PROMISC)
{
    /* Deal with promiscuous mode. */
    hdr_pack= bf_memreq(sizeof(*hdr));
    hdr= (psip_io_hdr_t *)ptr2acc_data(hdr_pack);
    memset(hdr, '\0', sizeof(*hdr));
    hdr->pih_flags |= PF_REM2LOC;
    hdr->pih_nexthop= nexthop;

    pack->acc_linkC++;
    hdr_pack->acc_next= pack;
    hdr_pack->acc_ext_link= NULL;
    if (psip_port->pp_promisc_head)
    {
        /* Append at the end. */
        psip_port->pp_promisc_tail->acc_ext_link= hdr_pack;
        psip_port->pp_promisc_tail= hdr_pack;
    }
    else
    {
        /* First packet. */
        psip_port->pp_promisc_head= hdr_pack;
        psip_port->pp_promisc_tail= hdr_pack;
        if (psip_port->pp_rd_head)
            promisc_restart_read(psip_port);
    }
}
ipps_put(psip_port->pp_ipdev, nexthop, pack);
pack= (*psip_fd->pf_get_userdata)(psip_fd->pf_srfd, (size_t)count,
    (size_t)0, FALSE);
assert(pack == NULL);
return NW_OK;
}

PRIVATE int psip_select(fd, operations)
int fd;
unsigned operations;
{
    printf("psip_select: not implemented\n");
    return 0;
}

PRIVATE void psip_close(fd)

```

```

int fd;
{
    psip_port_t *psip_port;
    psip_fd_t *psip_fd;

    assert(fd >= 0 && fd < PSIP_FD_NR);
    psip_fd= &psip_fd_table[fd];
    psip_port= psip_fd->pf_port;

    if (psip_fd->pf_flags & PFF_PROMISC)
    {
        /* Check if the port should still be in promiscuous mode.
        */
        psip_fd->pf_flags &= ~PFF_PROMISC;
        check_promisc(psip_fd->pf_port);
    }

    assert(psip_port->pp_opencnt >0);
    psip_port->pp_opencnt--;
    psip_fd->pf_flags= PFF_EMPTY;
    ipps_get(psip_port->pp_ipdev);
}

PRIVATE int psip_cancel(fd, which_operation)
int fd;
int which_operation;
{
    psip_port_t *psip_port;
    psip_fd_t *psip_fd, *prev_fd, *tmp_fd;
    int result;

    DBLOCK(1, printf("psip_cancel(%d,%d)\n", fd, which_operation));

    assert(fd >= 0 && fd < PSIP_FD_NR);
    psip_fd= &psip_fd_table[fd];
    psip_port= psip_fd->pf_port;

    switch(which_operation)
    {
    case SR_CANCEL_IOCTL:
        ip_panic(( "should not be here" ));
    case SR_CANCEL_READ:
        assert(psip_fd->pf_flags & PFF_READ_IP);
        for (prev_fd= NULL, tmp_fd= psip_port->pp_rd_head; tmp_fd;
             prev_fd= tmp_fd, tmp_fd= tmp_fd->pf_rd_next)
        {
            if (tmp_fd == psip_fd)
                break;
        }
        if (tmp_fd == NULL)
            ip_panic(( "unable to find to request to cancel" ));
        if (prev_fd == NULL)
            psip_port->pp_rd_head= psip_fd->pf_rd_next;
        else
            prev_fd->pf_rd_next= psip_fd->pf_rd_next;
        if (psip_fd->pf_rd_next == NULL)
            psip_port->pp_rd_tail= prev_fd;
        psip_fd->pf_flags &= ~PFF_READ_IP;
        result= (*psip_fd->pf_put_userdata)(psip_fd->pf_srfd,
                                           (size_t)EINTR, NULL, FALSE);
        assert(result == NW_OK);
        break;
    case SR_CANCEL_WRITE:
        ip_panic(( "should not be here" ));
    default:
        ip_panic(( "invalid operation for cancel" ));
    }
    return NW_OK;
}

PRIVATE void promisc_restart_read(psip_port)
psip_port_t *psip_port;
{

```

```

psip_fd_t *psip_fd, *prev, *next;
acc_t *pack;
size_t buf_size;
int result, result1;

/* Overkill at the moment: just one reader in promiscuous mode is
 * allowed.
 */
pack= psip_port->pp_promisc_head;
if (!pack)
    return;
assert(pack->acc_ext_link == NULL);

for(psip_fd= psip_port->pp_rd_head, prev= NULL; psip_fd;
    prev= psip_fd, psip_fd= psip_fd->pf_rd_next)
{
again:
    if (!(psip_fd->pf_flags & PFF_PROMISC))
        continue;
    next= psip_fd->pf_rd_next;
    if (prev)
        prev->pf_rd_next= next;
    else
        psip_port->pp_rd_head= next;
    if (!next)
        psip_port->pp_rd_tail= prev;

    assert(psip_fd->pf_flags & PFF_READ_IP);
    psip_fd->pf_flags &= ~PFF_READ_IP;

    buf_size= bf_bufsize(pack);
    if (buf_size <= psip_fd->pf_rd_count)
    {
        psip_port->pp_promisc_head= pack->acc_ext_link;
        result= (*psip_fd->pf_put_userdata)(psip_fd->pf_srfd,
            (size_t)0, pack, FALSE);
        if (result == NW_OK)
            result= buf_size;
    }
    else
        result= EPACKSIZE;

    result1= (*psip_fd->pf_put_userdata)(psip_fd->pf_srfd,
        (size_t)result, NULL, FALSE);
    assert(result1 == NW_OK);

    if (psip_port->pp_promisc_head)
    {
        /* Restart from the beginning */
        assert(result == EPACKSIZE);
        psip_fd= psip_port->pp_rd_head;
        prev= NULL;
        goto again;
    }
    break;
}
}

PRIVATE int psip_setopt(psip_fd, newoptp)
psip_fd_t *psip_fd;
nwio_psipt_t *newoptp;
{
    nwio_psipt_t oldopt;
    unsigned int new_en_flags, new_di_flags, old_en_flags, old_di_flags;
    unsigned long new_flags;

    oldopt= psip_fd->pf_psipt;

    old_en_flags= oldopt.nwpo_flags & 0xffff;
    old_di_flags= (oldopt.nwpo_flags >> 16) & 0xffff;

    new_en_flags= newoptp->nwpo_flags & 0xffff;
    new_di_flags= (newoptp->nwpo_flags >> 16) & 0xffff;

```

```

    if (new_en_flags & new_di_flags)
        return EBADMODE;

    /* NWUO_LOCADDR_MASK */
    if (!(new_en_flags | new_di_flags) & NWPO_PROMISC_MASK)
    {
        new_en_flags |= (old_en_flags & NWPO_PROMISC_MASK);
        new_di_flags |= (old_di_flags & NWPO_PROMISC_MASK);
    }

    new_flags = ((unsigned long)new_di_flags << 16) | new_en_flags;
    if ((new_flags & NWPO_EN_PROMISC) &&
        (psip_fd->pf_port->pp_flags & PPF_PROMISC))
    {
        printf("psip_setopt: EBUSY for port %d, flags 0x%x\n",
            psip_fd->pf_port - psip_port_table,
            psip_fd->pf_port->pp_flags);
        /* We can support only one at a time. */
        return EBUSY;
    }

    psip_fd->pf_psipopt = *newoptp;
    psip_fd->pf_psipopt.nwpo_flags = new_flags;

    return NW_OK;
}

PRIVATE void check_promisc(psip_port)
psip_port_t *psip_port;
{
    int i;
    psip_fd_t *psip_fd;
    acc_t *acc, *acc_next;

    /* Check if the port should still be in promiscuous mode. Overkill
     * at the moment.
     */
    if (!(psip_port->pp_flags & PPF_PROMISC))
        return;

    psip_port->pp_flags &= ~PPF_PROMISC;
    for (i = 0, psip_fd = psip_fd_table; i < PSIP_FD_NR; i++, psip_fd++)
    {
        if ((psip_fd->pf_flags & (PPF_INUSE | PPF_PROMISC)) !=
            (PPF_INUSE | PPF_PROMISC))
        {
            continue;
        }
        if (psip_fd->pf_port != psip_port)
            continue;
        printf("check_promisc: setting PROMISC for port %d\n",
            psip_port - psip_port_table);
        psip_port->pp_flags |= PPF_PROMISC;
        break;
    }
    if (!(psip_port->pp_flags & PPF_PROMISC))
    {
        /* Delete queued packets. */
        acc = psip_port->pp_promisc_head;
        psip_port->pp_promisc_head = NULL;
        while (acc)
        {
            acc_next = acc->acc_ext_link;
            bf_afree(acc);
            acc = acc_next;
        }
    }
}

PRIVATE void psip_buffree (priority)
int priority;
{
    int i;
    psip_port_t *psip_port;

```

```

    acc_t *tmp_acc, *next_acc;

    if (priority == PSIP_PRI_EXP_PROMISC)
    {
        for (i=0, psip_port= psip_port_table; i<psip_conf_nr;
             i++, psip_port++)
        {
            if (!(psip_port->pp_flags & PPF_CONFIGURED) )
                continue;
            if (psip_port->pp_promisc_head)
            {
                tmp_acc= psip_port->pp_promisc_head;
                while(tmp_acc)
                {
                    next_acc= tmp_acc->acc_ext_link;
                    bf_afree(tmp_acc);
                    tmp_acc= next_acc;
                }
                psip_port->pp_promisc_head= NULL;
            }
        }
    }
}

#ifdef BUF_CONSISTENCY_CHECK
PRIVATE void psip_bufcheck()
{
    int i;
    psip_port_t *psip_port;
    acc_t *tmp_acc;

    for (i= 0, psip_port= psip_port_table; i<psip_conf_nr;
         i++, psip_port++)
    {
        for (tmp_acc= psip_port->pp_promisc_head; tmp_acc;
             tmp_acc= tmp_acc->acc_ext_link)
        {
            bf_check_acc(tmp_acc);
        }
    }
}
#endif

/*
reply_thr_put
*/

PRIVATE void reply_thr_put(psip_fd, reply, for_ioctl)
psip_fd_t *psip_fd;
int reply;
int for_ioctl;
{
    int result;

    result= (*psip_fd->pf_put_userdata)(psip_fd->pf_srfd, reply,
                                       (acc_t *)0, for_ioctl);
    assert(result == NW_OK);
}

/*
reply_thr_get
*/

PRIVATE void reply_thr_get(psip_fd, reply, for_ioctl)
psip_fd_t *psip_fd;
int reply;
int for_ioctl;
{
    acc_t *result;
    result= (*psip_fd->pf_get_userdata)(psip_fd->pf_srfd, reply,
                                       (size_t)0, for_ioctl);
    assert (!result);
}

```



```
/*  
 * $PchId: psip.c,v 1.15 2005/06/28 14:19:29 philip Exp $  
 */
```

```
/*
generic/psip.h

Public interface to the pseudo IP module

Created:      Apr 22, 1993 by Philip Homburg

Copyright 1995 Philip Homburg
*/

#ifndef PSIP_H
#define PSIP_H

void psip_prep ARGS(( void ));
void psip_init ARGS(( void ));
int psip_enable ARGS(( int port_nr, int ip_port_nr ));
int psip_send ARGS(( int port_nr, ipaddr_t dest, acc_t *pack ));

#endif /* PSIP_H */

/*
 * $PchId: psip.h,v 1.6 2001/04/19 21:16:22 philip Exp $
 */
```

```
/*
rand256.c

Created:      Oct 2000 by Philip Homburg <philip@f-mnx.phicoh.com>

Generate 256-bit random numbers
*/

#include <sha2.h>
#include "inet.h"
#include "rand256.h"

PRIVATE u32_t base_bits[8];

PUBLIC void init_rand256(bits)
u8_t bits[32];
{
    memcpy(base_bits, bits, sizeof(base_bits));
}

PUBLIC void rand256(bits)
u8_t bits[32];
{
    u32_t a;
    SHA256_CTX ctx;

    a= ++base_bits[0];
    if (a == 0)
        base_bits[1]++;
    SHA256_Init(&ctx);
    SHA256_Update(&ctx, (unsigned char *)base_bits, sizeof(base_bits));
    SHA256_Final(bits, &ctx);
}

/*
 * $PchId: rand256.c,v 1.1 2005/06/28 14:13:43 philip Exp $
 */
```

```
/*  
rand256.h
```

```
Created:      Oct 2000 by Philip Homburg <philip@f-mnx.phicoh.com>
```

```
Provide 256-bit random numbers  
*/
```

```
#define RAND256_BUFSIZE 32
```

```
void init_rand256 ARGS(( u8_t bits[RAND256_BUFSIZE] ));  
void rand256 ARGS(( u8_t bits[RAND256_BUFSIZE] ));
```

```
/*  
 * $PchId: rand256.h,v 1.1 2005/06/28 14:14:05 philip Exp $  
 */
```

```
/*
sr.h

Copyright 1995 Philip Homburg
*/

#ifndef SR_H
#define SR_H

#define MAX_IOCTL_S      512

#define SR_CANCEL_IOCTL  1
#define SR_CANCEL_READ   2
#define SR_CANCEL_WRITE  3

#define SR_SELECT_READ    0x01
#define SR_SELECT_WRITE   0x02
#define SR_SELECT_EXCEPTION 0x04
#define SR_SELECT_POLL    0x10

/* Forward struct declarations */

struct acc;

/* prototypes */

typedef int  (*sr_open_t) ARGS(( int port, int srfd,
    get_userdata_t get_userdata, put_userdata_t put_userdata,
    put_pkt_t put_pkt, select_res_t select_res ));
typedef void (*sr_close_t) ARGS(( int fd ));
typedef int  (*sr_read_t)  ARGS(( int fd, size_t count ));
typedef int  (*sr_write_t) ARGS(( int fd, size_t count ));
typedef int  (*sr_ioctl_t) ARGS(( int fd, ioreq_t req ));
typedef int  (*sr_cancel_t) ARGS(( int fd, int which_operation ));
typedef int  (*sr_select_t) ARGS(( int fd, unsigned operations ));

void sr_init ARGS(( void ));
void sr_add_minor ARGS(( int minor, int port, sr_open_t openf,
    sr_close_t closef, sr_read_t sr_read, sr_write_t sr_write,
    sr_ioctl_t ioctlf, sr_cancel_t cancelf, sr_select_t selectf ));

#endif /* SR_H */

/* Track TCP connections back into sr (for lsof, identd, etc.) */
EXTERN sr_cancel_t tcp_cancel_f;

/*
 * $PchId: sr.h,v 1.9 2005/06/28 14:19:51 philip Exp $
 */
```

```
/*
tcp.c

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "buf.h"
#include "clock.h"
#include "event.h"
#include "type.h"

#include "io.h"
#include "ip.h"
#include "sr.h"
#include "assert.h"
#include "rand256.h"
#include "tcp.h"
#include "tcp_int.h"

THIS_FILE

#define NOT_IMPLEMENTED 0

PUBLIC tcp_port_t *tcp_port_table;
PUBLIC tcp_fd_t tcp_fd_table[TCP_FD_NR];
PUBLIC tcp_conn_t tcp_conn_table[TCP_CONN_NR];
PUBLIC sr_cancel_t tcp_cancel_f;

FORWARD void tcp_main ARGS(( tcp_port_t *port ));
FORWARD int tcp_select ARGS(( int fd, unsigned operations ));
FORWARD acc_t *tcp_get_data ARGS(( int fd, size_t offset,
size_t count, int for_ioctl ));
FORWARD int tcp_put_data ARGS(( int fd, size_t offset,
acc_t *data, int for_ioctl ));
FORWARD void tcp_put_pkt ARGS(( int fd, acc_t *data, size_t datalen ));
FORWARD void read_ip_packets ARGS(( tcp_port_t *port ));
FORWARD int tcp_setconf ARGS(( tcp_fd_t *tcp_fd ));
FORWARD int tcp_setopt ARGS(( tcp_fd_t *tcp_fd ));
FORWARD int tcp_connect ARGS(( tcp_fd_t *tcp_fd ));
FORWARD int tcp_listen ARGS(( tcp_fd_t *tcp_fd, int do_listenq ));
FORWARD int tcp_acceptto ARGS(( tcp_fd_t *tcp_fd ));
FORWARD tcpport_t find_unused_port ARGS(( int fd ));
FORWARD int is_unused_port ARGS(( Tcpport_t port ));
FORWARD int reply_thr_put ARGS(( tcp_fd_t *tcp_fd, int reply,
int for_ioctl ));
FORWARD void reply_thr_get ARGS(( tcp_fd_t *tcp_fd, int reply,
int for_ioctl ));
FORWARD tcp_conn_t *find_conn_entry ARGS(( Tcpport_t locport,
ipaddr_t locaddr, Tcpport_t remport, ipaddr_t readaddr ));
FORWARD tcp_conn_t *find_empty_conn ARGS(( void ));
FORWARD tcp_conn_t *find_best_conn ARGS(( ip_hdr_t *ip_hdr,
tcp_hdr_t *tcp_hdr ));
FORWARD tcp_conn_t *new_conn_for_queue ARGS(( tcp_fd_t *tcp_fd ));
FORWARD int maybe_listen ARGS(( ipaddr_t locaddr, Tcpport_t locport,
ipaddr_t remaddr, Tcpport_t remport ));
FORWARD int tcp_su4connect ARGS(( tcp_fd_t *tcp_fd ));
FORWARD void tcp_buffree ARGS(( int priority ));
#ifdef BUF_CONSISTENCY_CHECK
FORWARD void tcp_bufcheck ARGS(( void ));
#endif
FORWARD void tcp_setup_conn ARGS(( tcp_port_t *tcp_port,
tcp_conn_t *tcp_conn ));
FORWARD u32_t tcp_rand32 ARGS(( void ));

PUBLIC void tcp_prep()
{
    tcp_port_table= alloc(tcp_conf_nr * sizeof(tcp_port_table[0]));
}

PUBLIC void tcp_init()
{
    int i, j, k, ifno;
    tcp_fd_t *tcp_fd;
```

```

    tcp_port_t *tcp_port;
    tcp_conn_t *tcp_conn;

    assert (BUF_S >= sizeof(struct nwio_ipopt));
    assert (BUF_S >= sizeof(struct nwio_ipconf));
    assert (BUF_S >= sizeof(struct nwio_tcpconf));
    assert (BUF_S >= IP_MAX_HDR_SIZE + TCP_MAX_HDR_SIZE);

    for (i=0, tcp_fd= tcp_fd_table; i<TCP_FD_NR; i++, tcp_fd++)
    {
        tcp_fd->tf_flags= TFF_EMPTY;
    }

    for (i=0, tcp_conn= tcp_conn_table; i<TCP_CONN_NR; i++,
        tcp_fd++)
    {
        tcp_conn->tc_flags= TCF_EMPTY;
        tcp_conn->tc_busy= 0;
    }

#ifdef BUF_CONSISTENCY_CHECK
    bf_logon(tcp_buffree);
#else
    bf_logon(tcp_buffree, tcp_bufcheck);
#endif

    for (i=0, tcp_port= tcp_port_table; i<tcp_conf_nr; i++, tcp_port++)
    {
        tcp_port->tp_ipdev= tcp_conf[i].tc_port;

        tcp_port->tp_flags= TPF_EMPTY;
        tcp_port->tp_state= TPS_EMPTY;
        tcp_port->tp_snd_head= NULL;
        tcp_port->tp_snd_tail= NULL;
        ev_init(&tcp_port->tp_snd_event);
        for (j= 0; j<TCP_CONN_HASH_NR; j++)
        {
            for (k= 0; k<4; k++)
            {
                tcp_port->tp_conn_hash[j][k]=
                    &tcp_conn_table[0];
            }
        }

        ifno= ip_conf[tcp_port->tp_ipdev].ic_ifno;
        sr_add_minor(if2minor(ifno, TCP_DEV_OFF),
            i, tcp_open, tcp_close, tcp_read,
            tcp_write, tcp_ioctl, tcp_cancel, tcp_select);

        tcp_main(tcp_port);
    }
    tcp_cancel_f= tcp_cancel;
}

PRIVATE void tcp_main(tcp_port)
tcp_port_t *tcp_port;
{
    int result, i;
    tcp_conn_t *tcp_conn;
    tcp_fd_t *tcp_fd;

    switch (tcp_port->tp_state)
    {
    case TPS_EMPTY:
        tcp_port->tp_state= TPS_SETPROTO;
        tcp_port->tp_ipfd= ip_open(tcp_port->tp_ipdev,
            tcp_port->tp_ipdev, tcp_get_data,
            tcp_put_data, tcp_put_pkt, 0 /* no select_res */);
        if (tcp_port->tp_ipfd < 0)
        {
            tcp_port->tp_state= TPS_ERROR;
            DBLOCK(1, printf( "%s, %d: unable to open ip port\n",
                __FILE__, __LINE__));
            return;
        }
    }
}

```

```

    }

    result= ip_ioctl(tcp_port->tp_ipfd, NWIOSIPOPT);
    if (result == NW_SUSPEND)
        tcp_port->tp_flags |= TPF_SUSPEND;
    if (result < 0)
    {
        return;
    }
    if (tcp_port->tp_state != TPS_GETCONF)
        return;
    /* drops through */
case TPS_GETCONF:
    tcp_port->tp_flags &= ~TPF_SUSPEND;

    result= ip_ioctl(tcp_port->tp_ipfd, NWIOGIPCONF);
    if (result == NW_SUSPEND)
        tcp_port->tp_flags |= TPF_SUSPEND;
    if (result < 0)
    {
        return;
    }
    if (tcp_port->tp_state != TPS_MAIN)
        return;
    /* drops through */
case TPS_MAIN:
    tcp_port->tp_flags &= ~TPF_SUSPEND;
    tcp_port->tp_pack= 0;

    tcp_conn= &tcp_conn_table[tcp_port->tp_ipdev];
    tcp_conn->tc_flags= TCF_INUSE;
    assert(!tcp_conn->tc_busy);
    tcp_conn->tc_locport= 0;
    tcp_conn->tc_locaddr= tcp_port->tp_ipaddr;
    tcp_conn->tc_rempport= 0;
    tcp_conn->tc_remaddr= 0;
    tcp_conn->tc_state= TCS_CLOSED;
    tcp_conn->tc_fd= 0;
    tcp_conn->tc_connInprogress= 0;
    tcp_conn->tc_orglisten= FALSE;
    tcp_conn->tc_senddis= 0;
    tcp_conn->tc_ISS= 0;
    tcp_conn->tc_SND_UNA= tcp_conn->tc_ISS;
    tcp_conn->tc_SND_TRM= tcp_conn->tc_ISS;
    tcp_conn->tc_SND_NXT= tcp_conn->tc_ISS;
    tcp_conn->tc_SND_UP= tcp_conn->tc_ISS;
    tcp_conn->tc_IRS= 0;
    tcp_conn->tc_RCV_LO= tcp_conn->tc_IRS;
    tcp_conn->tc_RCV_NXT= tcp_conn->tc_IRS;
    tcp_conn->tc_RCV_HI= tcp_conn->tc_IRS;
    tcp_conn->tc_RCV_UP= tcp_conn->tc_IRS;
    tcp_conn->tc_port= tcp_port;
    tcp_conn->tc_rcvd_data= NULL;
    tcp_conn->tc_adv_data= NULL;
    tcp_conn->tc_send_data= 0;
    tcp_conn->tc_remipopt= NULL;
    tcp_conn->tc_tcpopt= NULL;
    tcp_conn->tc_frag2send= 0;
    tcp_conn->tc_tos= TCP_DEF_TOS;
    tcp_conn->tc_ttl= IP_MAX_TTL;
    tcp_conn->tc_rcv_wnd= TCP_MAX_RCV_WND_SIZE;
    tcp_conn->tc_rt_dead= TCP_DEF_RT_DEAD;
    tcp_conn->tc_stt= 0;
    tcp_conn->tc_0wnd_to= 0;
    tcp_conn->tc_artt= TCP_DEF_RTT*TCP_RTT_SCALE;
    tcp_conn->tc_drtdt= 0;
    tcp_conn->tc_rtt= TCP_DEF_RTT;
    tcp_conn->tc_max_mtu= tcp_port->tp_mtu;
    tcp_conn->tc_mtu= tcp_conn->tc_max_mtu;
    tcp_conn->tc_mtutim= 0;
    tcp_conn->tc_error= NW_OK;
    tcp_conn->tc_snd_wnd= TCP_MAX_SND_WND_SIZE;
    tcp_conn->tc_snd_cinc=
        (long)TCP_DEF_MSS*TCP_DEF_MSS/TCP_MAX_SND_WND_SIZE+1;

```



```

    tcp_conn->tc_rt_time= 0;
    tcp_conn->tc_rt_seq= 0;
    tcp_conn->tc_rt_threshold= tcp_conn->tc_ISS;

    for (i=0, tcp_fd= tcp_fd_table; i<TCP_FD_NR; i++,
        tcp_fd++)
    {
        if (!(tcp_fd->tf_flags & TFF_INUSE))
            continue;
        if (tcp_fd->tf_port != tcp_port)
            continue;
        if (tcp_fd->tf_flags & TFF_IOC_INIT_SP)
        {
            tcp_fd->tf_flags &= ~TFF_IOC_INIT_SP;
            tcp_ioctl(i, tcp_fd->tf_ioreq);
        }
    }
    read_ip_packets(tcp_port);
    return;

default:
    ip_panic(( "unknown state" ));
    break;
}
}

PRIVATE int tcp_select(fd, operations)
int fd;
unsigned operations;
{
    int i;
    unsigned resops;
    tcp_fd_t *tcp_fd;
    tcp_conn_t *tcp_conn;

    tcp_fd= &tcp_fd_table[fd];
    assert (tcp_fd->tf_flags & TFF_INUSE);

    resops= 0;
    if (tcp_fd->tf_flags & TFF_LISTENQ)
    {
        /* Special case for LISTENQ */
        if (operations & SR_SELECT_READ)
        {
            for (i= 0; i<TFL_LISTEN_MAX; i++)
            {
                if (tcp_fd->tf_listenq[i] == NULL)
                    continue;
                if (tcp_fd->tf_listenq[i]->tc_connInProgress
                    == 0)
                {
                    break;
                }
            }
            if (i >= TFL_LISTEN_MAX)
                tcp_fd->tf_flags |= TFF_SEL_READ;
            else
                resops |= SR_SELECT_READ;
        }
        if (operations & SR_SELECT_WRITE)
        {
            /* We can't handles writes. Just return the error
             * when the user tries to write.
             */
            resops |= SR_SELECT_WRITE;
        }
        return resops;
    }
    if (tcp_fd->tf_flags & TFF_CONNECTING)
    {
        /* Special case for CONNECTING */
        if (operations & SR_SELECT_WRITE)
            tcp_fd->tf_flags |= TFF_SEL_WRITE;
    }
}

```

```

        return 0;
    }
    if (operations & SR_SELECT_READ)
    {
        tcp_conn= tcp_fd->tf_conn;

        if (!(tcp_fd->tf_flags & TFF_CONNECTED))
        {
            /* We can't handle reads until a connection has been
             * established. Return the error when the user tries
             * to read.
             */
            resops |= SR_SELECT_READ;
        }
        else if (tcp_conn->tc_state == TCS_CLOSED ||
                 tcp_sel_read(tcp_conn))
        {
            resops |= SR_SELECT_READ;
        }
        else if (!(operations & SR_SELECT_POLL))
            tcp_fd->tf_flags |= TFF_SEL_READ;
    }
    if (operations & SR_SELECT_WRITE)
    {
        tcp_conn= tcp_fd->tf_conn;
        if (!(tcp_fd->tf_flags & TFF_CONNECTED))
        {
            /* We can't handle writes until a connection has been
             * established. Return the error when the user tries
             * to write.
             */
            resops |= SR_SELECT_WRITE;
        }
        else if (tcp_conn->tc_state == TCS_CLOSED ||
                 tcp_conn->tc_flags & TCF_FIN_SENT ||
                 tcp_sel_write(tcp_conn))
        {
            resops |= SR_SELECT_WRITE;
        }
        else if (!(operations & SR_SELECT_POLL))
            tcp_fd->tf_flags |= TFF_SEL_WRITE;
    }
    if (operations & SR_SELECT_EXCEPTION)
    {
        /* Should add code for exceptions */
    }
    return resops;
}

```

```

PRIVATE acc_t *tcp_get_data (port, offset, count, for_ioctl)
int port;
size_t offset;
size_t count;
int for_ioctl;
{
    tcp_port_t *tcp_port;
    int result;

    tcp_port= &tcp_port_table[port];

    switch (tcp_port->tp_state)
    {
    case TPS_SETPROTO:
        if (!count)
        {
            result= (int)offset;
            if (result<0)
            {
                tcp_port->tp_state= TPS_ERROR;
                break;
            }
            tcp_port->tp_state= TPS_GETCONF;
            if (tcp_port->tp_flags & TPF_SUSPEND)
                tcp_main(tcp_port);
        }
    }
}

```

```

        return NW_OK;
    }
    assert (!offset);
    assert (count == sizeof(struct nwio_ipopt));
    {
        struct nwio_ipopt *ipopt;
        acc_t *acc;

        acc= bf_memreq(sizeof(*ipopt));
        ipopt= (struct nwio_ipopt *)ptr2acc_data(acc);
        ipopt->nwio_flags= NWIO_COPY |
            NWIO_EN_LOC | NWIO_DI_BROAD |
            NWIO_REMANY | NWIO_PROTOSPEC |
            NWIO_HDR_O_ANY | NWIO_RWDATALL;
        ipopt->nwio_proto= IPPROTO_TCP;
        return acc;
    }
    case TPS_MAIN:
        assert(tcp_port->tp_flags & TPF_WRITE_IP);
        if (!count)
        {
            result= (int)offset;
            if (result<0)
            {
                if (result == EDSTNOTRCH)
                {
                    if (tcp_port->tp_snd_head)
                    {
                        tcp_notreach(tcp_port->
                                    tp_snd_head);
                    }
                }
                else
                {
                    ip_warning((
                        "ip_write failed with error: %d\n",
                        result ));
                }
            }
            assert (tcp_port->tp_pack);
            bf_afree (tcp_port->tp_pack);
            tcp_port->tp_pack= 0;

            if (tcp_port->tp_flags & TPF_WRITE_SP)
            {
                tcp_port->tp_flags &= ~(TPF_WRITE_SP|
                    TPF_WRITE_IP);
                if (tcp_port->tp_snd_head)
                    tcp_port_write(tcp_port);
            }
            else
                tcp_port->tp_flags &= ~TPF_WRITE_IP;
        }
        else
        {
            return bf_cut (tcp_port->tp_pack, offset,
                           count);
        }
        break;
    default:
        printf("tcp_get_data(%d, 0x%x, 0x%x) called but tp_state= 0x%x\n",
               port, offset, count, tcp_port->tp_state);
        break;
    }
    return NW_OK;
}

PRIVATE int tcp_put_data (fd, offset, data, for_ioctl)
int fd;
size_t offset;
acc_t *data;
int for_ioctl;
{
    tcp_port_t *tcp_port;

```

```

    int result;

    tcp_port= &tcp_port_table[fd];

    switch (tcp_port->tp_state)
    {
    case TPS_GETCONF:
        if (!data)
        {
            result= (int)offset;
            if (result<0)
            {
                tcp_port->tp_state= TPS_ERROR;
                return NW_OK;
            }
            tcp_port->tp_state= TPS_MAIN;
            if (tcp_port->tp_flags & TPF_SUSPEND)
                tcp_main(tcp_port);
        }
        else
        {
            struct nwio_ipconf *ipconf;

            data= bf_packIffLess(data, sizeof(*ipconf));
            ipconf= (struct nwio_ipconf *)ptr2acc_data(data);
assert (ipconf->nwic_flags & NWIC_IPADDR_SET);
            tcp_port->tp_ipaddr= ipconf->nwic_ipaddr;
            tcp_port->tp_subnetmask= ipconf->nwic_netmask;
            tcp_port->tp_mtu= ipconf->nwic_mtu;
            bf_afree(data);
        }
        break;
    case TPS_MAIN:
        assert(tcp_port->tp_flags & TPF_READ_IP);
        if (!data)
        {
            result= (int)offset;
            if (result<0)
                ip_panic(( "ip_read() failed" ));

            if (tcp_port->tp_flags & TPF_READ_SP)
            {
                tcp_port->tp_flags &= ~(TPF_READ_SP|
                    TPF_READ_IP);
                read_ip_packets(tcp_port);
            }
            else
                tcp_port->tp_flags &= ~TPF_READ_IP;
        }
        else
        {
            assert(!offset);
            /* this is an invalid assertion but ip sends
             * only whole datagrams up */
            tcp_put_pkt(fd, data, bf_bufsize(data));
        }
        break;
    default:
        printf(
            "tcp_put_data(%d, 0x%x, %p) called but tp_state= 0x%x\n",
            fd, offset, data, tcp_port->tp_state);
        break;
    }
    return NW_OK;
}

/*
tcp_put_pkt
*/

PRIVATE void tcp_put_pkt(fd, data, datalen)
int fd;
acc_t *data;
size_t datalen;

```

```

{
    tcp_port_t *tcp_port;
    tcp_conn_t *tcp_conn, **conn_p;
    ip_hdr_t *ip_hdr;
    tcp_hdr_t *tcp_hdr;
    acc_t *ip_pack, *tcp_pack;
    size_t ip_datalen, tcp_datalen, ip_hdr_len, tcp_hdr_len;
    u16_t sum, mtu;
    u32_t bits;
    int i, hash;
    ipaddr_t srcaddr, dstaddr, ipaddr, mask;
    tcpport_t srcport, dstport;

    tcp_port = &tcp_port_table[fd];

    /* Extract the IP header. */
    ip_hdr = (ip_hdr_t *)ptr2acc_data(data);
    ip_hdr_len = (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
    ip_datalen = datalen - ip_hdr_len;
    if (ip_datalen == 0)
    {
        if (ip_hdr->ih_proto == 0)
        {
            /* IP layer reports new IP address */
            ipaddr = ip_hdr->ih_src;
            mask = ip_hdr->ih_dst;
            mtu = ntohs(ip_hdr->ih_length);
            tcp_port->tp_ipaddr = ipaddr;
            tcp_port->tp_subnetmask = mask;
            tcp_port->tp_mtu = mtu;
            DBLOCK(1, printf("tcp_put_pkt: using address ");
                    writeIpAddr(ipaddr);
                    printf(", netmask ");
                    writeIpAddr(mask);
                    printf(", mtu %u\n", mtu));
            for (i = 0, tcp_conn = tcp_conn_table + i;
                 i < TCP_CONN_NR; i++, tcp_conn++)
            {
                if (!(tcp_conn->tc_flags & TCF_INUSE))
                    continue;
                if (tcp_conn->tc_port != tcp_port)
                    continue;
                tcp_conn->tc_locaddr = ipaddr;
            }
        }
        else
            DBLOCK(1, printf("tcp_put_pkt: no TCP header\n"));
        bf_afree(data);
        return;
    }
    data->acc_linkC++;
    ip_pack = data;
    ip_pack = bf_align(ip_pack, ip_hdr_len, 4);
    ip_hdr = (ip_hdr_t *)ptr2acc_data(ip_pack);
    data = bf_delhead(data, ip_hdr_len);

    /* Compute the checksum */
    sum = tcp_pack_oneCsum(ip_hdr, data);

    /* Extract the TCP header */
    if (ip_datalen < TCP_MIN_HDR_SIZE)
    {
        DBLOCK(1, printf("truncated TCP header\n"));
        bf_afree(ip_pack);
        bf_afree(data);
        return;
    }
    data = bf_packIfLess(data, TCP_MIN_HDR_SIZE);
    tcp_hdr = (tcp_hdr_t *)ptr2acc_data(data);
    tcp_hdr_len = (tcp_hdr->th_data_off & TH_DO_MASK) >> 2;
    /* actually (>> 4) << 2 */
    if (ip_datalen < tcp_hdr_len || tcp_hdr_len < TCP_MIN_HDR_SIZE)
    {
        if (tcp_hdr_len < TCP_MIN_HDR_SIZE)

```

```

        {
            DBLOCK(1, printf("strange tcp header length %d\n",
                             tcp_hdr_len));
        }
        else
        {
            DBLOCK(1, printf("truncated TCP header\n"));
        }
        bf_afree(ip_pack);
        bf_afree(data);
        return;
    }
    data->acc_linkC++;
    tcp_pack= data;
    tcp_pack= bf_align(tcp_pack, tcp_hdr_len, 4);
    tcp_hdr= (tcp_hdr_t *)ptr2acc_data(tcp_pack);
    if (ip_datalen == tcp_hdr_len)
    {
        bf_afree(data);
        data= NULL;
    }
    else
        data= bf_delhead(data, tcp_hdr_len);
    tcp_datalen= ip_datalen-tcp_hdr_len;

    if ((u16_t)~sum)
    {
        DBLOCK(1, printf("checksum error in tcp packet\n");
                printf("tcp_pack_oneCsum(...)= 0x%x length= %d\n",
                        (u16_t)~sum, tcp_datalen);
                printf("src ip_addr= "); writeIpAddr(ip_hdr->ih_src);
                printf("\n"));
        bf_afree(ip_pack);
        bf_afree(tcp_pack);
        bf_afree(data);
        return;
    }

    srcaddr= ip_hdr->ih_src;
    dstaddr= ip_hdr->ih_dst;
    srcport= tcp_hdr->th_srcport;
    dstport= tcp_hdr->th_dstport;
    bits= srcaddr ^ dstaddr ^ srcport ^ dstport;
    bits= (bits >> 16) ^ bits;
    bits= (bits >> 8) ^ bits;
    hash= ((bits >> TCP_CONN_HASH_SHIFT) ^ bits) & (TCP_CONN_HASH_NR-1);
    conn_p= tcp_port->tp_conn_hash[hash];
    if (conn_p[0]->tc_locport == dstport &&
        conn_p[0]->tc_remport == srcport &&
        conn_p[0]->tc_remaddr == srcaddr &&
        conn_p[0]->tc_locaddr == dstaddr)
    {
        tcp_conn= conn_p[0];
    }
    else if (conn_p[1]->tc_locport == dstport &&
        conn_p[1]->tc_remport == srcport &&
        conn_p[1]->tc_remaddr == srcaddr &&
        conn_p[1]->tc_locaddr == dstaddr)
    {
        tcp_conn= conn_p[1];
        conn_p[1]= conn_p[0];
        conn_p[0]= tcp_conn;
    }
    else if (conn_p[2]->tc_locport == dstport &&
        conn_p[2]->tc_remport == srcport &&
        conn_p[2]->tc_remaddr == srcaddr &&
        conn_p[2]->tc_locaddr == dstaddr)
    {
        tcp_conn= conn_p[2];
        conn_p[2]= conn_p[1];
        conn_p[1]= conn_p[0];
        conn_p[0]= tcp_conn;
    }
    else if (conn_p[3]->tc_locport == dstport &&

```

```

        conn_p[3]->tc_rempport == srcport &&
        conn_p[3]->tc_remaddr == srcaddr &&
        conn_p[3]->tc_locaddr == dstaddr)
    {
        tcp_conn= conn_p[3];
        conn_p[3]= conn_p[2];
        conn_p[2]= conn_p[1];
        conn_p[1]= conn_p[0];
        conn_p[0]= tcp_conn;
    }
    else
        tcp_conn= NULL;
    if ((tcp_conn != NULL && tcp_conn->tc_state == TCS_CLOSED) ||
        (tcp_hdr->th_flags & THF_SYN))
    {
        tcp_conn= NULL;
    }

    if (tcp_conn == NULL)
    {
        tcp_conn= find_best_conn(ip_hdr, tcp_hdr);
        if (!tcp_conn)
        {
            /* listen backlog hack */
            bf_afree(ip_pack);
            bf_afree(tcp_pack);
            bf_afree(data);
            return;
        }
        if (tcp_conn->tc_state != TCS_CLOSED)
        {
            conn_p[3]= conn_p[2];
            conn_p[2]= conn_p[1];
            conn_p[1]= conn_p[0];
            conn_p[0]= tcp_conn;
        }
    }
    assert(tcp_conn->tc_busy == 0);
    tcp_conn->tc_busy++;
    tcp_frag2conn(tcp_conn, ip_hdr, tcp_hdr, data, tcp_datalen);
    tcp_conn->tc_busy--;
    bf_afree(ip_pack);
    bf_afree(tcp_pack);
}

PUBLIC int tcp_open (port, srfd, get_userdata, put_userdata, put_pkt,
    select_res)
int port;
int srfd;
get_userdata_t get_userdata;
put_userdata_t put_userdata;
put_pkt_t put_pkt;
select_res_t select_res;
{
    int i, j;

    tcp_fd_t *tcp_fd;

    for (i=0; i<TCP_FD_NR && (tcp_fd_table[i].tf_flags & TFF_INUSE);
        i++);
    if (i>=TCP_FD_NR)
    {
        return EAGAIN;
    }

    tcp_fd= &tcp_fd_table[i];

    tcp_fd->tf_flags= TFF_INUSE;
    tcp_fd->tf_flags |= TFF_PUSH_DATA;

    tcp_fd->tf_port= &tcp_port_table[port];
    tcp_fd->tf_srfd= srfd;
    tcp_fd->tf_tcpconf.nwtc_flags= TCP_DEF_CONF;

```

```

    tcp_fd->tf_tcpconf.nwtc_remaddr= 0;
    tcp_fd->tf_tcpconf.nwtc_rempport= 0;
    tcp_fd->tf_tcpopt.nwto_flags= TCP_DEF_OPT;
    tcp_fd->tf_get_userdata= get_userdata;
    tcp_fd->tf_put_userdata= put_userdata;
    tcp_fd->tf_select_res= select_res;
    tcp_fd->tf_conn= 0;
    tcp_fd->tf_error= 0;
    for (j= 0; j<TFL_LISTEN_MAX; j++)
        tcp_fd->tf_listenq[j]= NULL;
    return i;
}

/*
tcp_ioctl
*/
PUBLIC int tcp_ioctl (fd, req)
int fd;
ioreq_t req;
{
    tcp_fd_t *tcp_fd;
    tcp_port_t *tcp_port;
    tcp_conn_t *tcp_conn;
    nwio_tcpconf_t *tcp_conf;
    nwio_tcpopt_t *tcp_opt;
    tcp_cookie_t *cookiep;
    acc_t *acc, *conf_acc, *opt_acc;
    int result, *bytesp;
    u8_t rndbits[RAND256_BUFSIZE];

    tcp_fd= &tcp_fd_table[fd];

    assert (tcp_fd->tf_flags & TFF_INUSE);

    tcp_port= tcp_fd->tf_port;
    tcp_fd->tf_flags |= TFF_IOCTL_IP;
    tcp_fd->tf_ioreq= req;

    if (tcp_port->tp_state != TPS_MAIN)
    {
        tcp_fd->tf_flags |= TFF_IOC_INIT_SP;
        return NW_SUSPEND;
    }

    switch (req)
    {
    case NWIOSTCPCONF:
        if ((tcp_fd->tf_flags & TFF_CONNECTED) ||
            (tcp_fd->tf_flags & TFF_CONNECTING) ||
            (tcp_fd->tf_flags & TFF_LISTENQ))
        {
            tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
            reply_thr_get (tcp_fd, EISCONN, TRUE);
            result= NW_OK;
            break;
        }
        result= tcp_setconf(tcp_fd);
        break;
    case NWIOGTCPCONF:
        conf_acc= bf_memreq(sizeof(*tcp_conf));
        assert (conf_acc->acc_length == sizeof(*tcp_conf));
        tcp_conf= (nwio_tcpconf_t *)ptr2acc_data(conf_acc);

        *tcp_conf= tcp_fd->tf_tcpconf;
        if (tcp_fd->tf_flags & TFF_CONNECTED)
        {
            tcp_conn= tcp_fd->tf_conn;
            tcp_conf->nwtc_locport= tcp_conn->tc_locport;
            tcp_conf->nwtc_remaddr= tcp_conn->tc_remaddr;
            tcp_conf->nwtc_rempport= tcp_conn->tc_rempport;
        }
        tcp_conf->nwtc_locaddr= tcp_fd->tf_port->tp_ipaddr;
        result= (*tcp_fd->tf_put_userdata)(tcp_fd->tf_srfd,
            0, conf_acc, TRUE);
    }
}

```



```

        tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
        reply_thr_put(tcp_fd, result, TRUE);
        result= NW_OK;
        break;
    case NWIOSTCPOPT:
        result= tcp_setopt(tcp_fd);
        break;
    case NWIOGTCPOPT:
        opt_acc= bf_memreq(sizeof(*tcp_opt));
        assert (opt_acc->acc_length == sizeof(*tcp_opt));
        tcp_opt= (nwio_tcptopt_t *)ptr2acc_data(opt_acc);

        *tcp_opt= tcp_fd->tf_tcptopt;
        result= (*tcp_fd->tf_put_userdata)(tcp_fd->tf_srfd,
            0, opt_acc, TRUE);
        tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
        reply_thr_put(tcp_fd, result, TRUE);
        result= NW_OK;
        break;
    case NWIOTCPCONN:
        if (tcp_fd->tf_flags & TFF_CONNECTING)
        {
            tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
            reply_thr_get (tcp_fd, EALREADY, TRUE);
            result= NW_OK;
            break;
        }
        if (tcp_fd->tf_flags & TFF_CONNECTED)
        {
            tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
            reply_thr_get (tcp_fd, EISCONN, TRUE);
            result= NW_OK;
            break;
        }
        result= tcp_connect(tcp_fd);
        if (result == NW_OK)
            tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
        break;
    case NWIOTCPLISTEN:
    case NWIOTCPLISTENQ:
        if ((tcp_fd->tf_flags & TFF_CONNECTED) ||
            (tcp_fd->tf_flags & TFF_LISTENQ) ||
            (tcp_fd->tf_flags & TFF_CONNECTING))
        {
            tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
            reply_thr_get (tcp_fd, EISCONN, TRUE);
            result= NW_OK;
            break;
        }
        result= tcp_listen(tcp_fd, (req == NWIOTCPLISTENQ));
        break;
    case NWIOTCPSHUTDOWN:
        if (!(tcp_fd->tf_flags & TFF_CONNECTED))
        {
            tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
            reply_thr_get (tcp_fd, ENOTCONN, TRUE);
            result= NW_OK;
            break;
        }
        tcp_fd->tf_flags |= TFF_IOCTL_IP;
        tcp_fd->tf_ioreq= req;
        tcp_conn= tcp_fd->tf_conn;

        tcp_conn->tc_busy++;
        tcp_fd_write(tcp_conn);
        tcp_conn->tc_busy--;
        tcp_conn_write(tcp_conn, 0);
        if (!(tcp_fd->tf_flags & TFF_IOCTL_IP))
            result= NW_OK;
        else
            result= NW_SUSPEND;
        break;
    case NWIOTCPPUSH:
        if (!(tcp_fd->tf_flags & TFF_CONNECTED))

```

```

    {
        tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
        reply_thr_get(tcp_fd, ENOTCONN, TRUE);
        result= NW_OK;
        break;
    }
    tcp_conn= tcp_fd->tf_conn;
    tcp_conn->tc_SND_PSH= tcp_conn->tc_SND_NXT;
    tcp_conn->tc_flags &= ~TCF_NO_PUSH;
    tcp_conn->tc_flags |= TCF_PUSH_NOW;

    /* Start the timer (if necessary) */
    if (tcp_conn->tc_SND_TRM == tcp_conn->tc_SND_UNA)
        tcp_set_send_timer(tcp_conn);

    tcp_conn_write(tcp_conn, 0);
    tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
    reply_thr_get(tcp_fd, NW_OK, TRUE);
    result= NW_OK;
    break;
case NWIOGTCPCOOKIE:
    if (!(tcp_fd->tf_flags & TFF_COOKIE))
    {
        tcp_fd->tf_cookie.tc_ref= fd;
        rand256(rndbits);
        assert(sizeof(tcp_fd->tf_cookie.tc_secret) <=
                RAND256_BUFSIZE);
        memcpy(tcp_fd->tf_cookie.tc_secret,
                rndbits, sizeof(tcp_fd->tf_cookie.tc_secret));
        tcp_fd->tf_flags |= TFF_COOKIE;
    }
    acc= bf_memreq(sizeof(*cookiep));
    cookiep= (tcp_cookie_t *)ptr2acc_data(acc);

    *cookiep= tcp_fd->tf_cookie;
    result= (*tcp_fd->tf_put_userdata)(tcp_fd->tf_srfd,
        0, acc, TRUE);
    tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
    reply_thr_put(tcp_fd, result, TRUE);
    result= NW_OK;
    break;
case NWIOTCPACCEPTTO:
    result= tcp_acceptto(tcp_fd);
    break;
case FIONREAD:
    acc= bf_memreq(sizeof(*bytesp));
    bytesp= (int *)ptr2acc_data(acc);
    tcp_bytesavailable(tcp_fd, bytesp);
    result= (*tcp_fd->tf_put_userdata)(tcp_fd->tf_srfd,
        0, acc, TRUE);
    tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
    reply_thr_put(tcp_fd, result, TRUE);
    result= NW_OK;
    break;
case NWIOTCPGERROR:
    acc= bf_memreq(sizeof(*bytesp));
    bytesp= (int *)ptr2acc_data(acc);
    *bytesp= -tcp_fd->tf_error;    /* Errors are positive in
                                * user space.
                                */

    tcp_fd->tf_error= 0;
    result= (*tcp_fd->tf_put_userdata)(tcp_fd->tf_srfd,
        0, acc, TRUE);
    tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
    reply_thr_put(tcp_fd, result, TRUE);
    result= NW_OK;
    break;
default:
    tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
    reply_thr_get(tcp_fd, EBADIOCTL, TRUE);
    result= NW_OK;
    break;

```

```

    }
    return result;
}

/*
tcp_setconf
*/

PRIVATE int tcp_setconf(tcp_fd)
tcp_fd_t *tcp_fd;
{
    nwio_tcpconf_t *tcpconf;
    nwio_tcpconf_t oldconf, newconf;
    acc_t *data;
    tcp_fd_t *fd_ptr;
    unsigned int new_en_flags, new_di_flags,
               old_en_flags, old_di_flags, all_flags, flags;
    int i;

    data= (*tcp_fd->tf_get_userdata)
        (tcp_fd->tf_srfd, 0,
         sizeof(nwio_tcpconf_t), TRUE);

    if (!data)
        return EFAULT;

    data= bf_packIfLess(data, sizeof(nwio_tcpconf_t));
assert (data->acc_length == sizeof(nwio_tcpconf_t));

    tcpconf= (nwio_tcpconf_t *)ptr2acc_data(data);
    oldconf= tcp_fd->tf_tcpconf;
    newconf= *tcpconf;

    old_en_flags= oldconf.nwtc_flags & 0xffff;
    old_di_flags= (oldconf.nwtc_flags >> 16) &
                  0xffff;
    new_en_flags= newconf.nwtc_flags & 0xffff;
    new_di_flags= (newconf.nwtc_flags >> 16) &
                  0xffff;
    if (new_en_flags & new_di_flags)
    {
        tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
        reply_thr_get(tcp_fd, EBADMODE, TRUE);
        bf_afree(data);
        return NW_OK;
    }

    /* NWTC_ACC_MASK */
    if (new_di_flags & NWTC_ACC_MASK)
    {
        tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
        reply_thr_get(tcp_fd, EBADMODE, TRUE);
        bf_afree(data);
        return NW_OK;
        /* access modes can't be disabled */
    }

    if (!(new_en_flags & NWTC_ACC_MASK))
        new_en_flags |= (old_en_flags & NWTC_ACC_MASK);

    /* NWTC_LOCPORT_MASK */
    if (new_di_flags & NWTC_LOCPORT_MASK)
    {
        tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
        reply_thr_get(tcp_fd, EBADMODE, TRUE);
        bf_afree(data);
        return NW_OK;
        /* the loc ports can't be disabled */
    }
    if (!(new_en_flags & NWTC_LOCPORT_MASK))
    {
        new_en_flags |= (old_en_flags &
                         NWTC_LOCPORT_MASK);
    }
}

```

```

        newconf.nwtc_locport= oldconf.nwtc_locport;
    }
    else if ((new_en_flags & NWTC_LOCPORT_MASK) == NWTC_LP_SEL)
    {
        newconf.nwtc_locport= find_unused_port(tcp_fd-
            tcp_fd_table);
    }
    else if ((new_en_flags & NWTC_LOCPORT_MASK) == NWTC_LP_SET)
    {
        if (!newconf.nwtc_locport)
        {
            tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
            reply_thr_get(tcp_fd, EBADMODE, TRUE);
            bf_afree(data);
            return NW_OK;
        }
    }

    /* NWTC_REMADDR_MASK */
    if (!(new_en_flags | new_di_flags) &
        NWTC_REMADDR_MASK)
    {
        new_en_flags |= (old_en_flags &
            NWTC_REMADDR_MASK);
        new_di_flags |= (old_di_flags &
            NWTC_REMADDR_MASK);
        newconf.nwtc_remaddr= oldconf.nwtc_remaddr;
    }
    else if (new_en_flags & NWTC_SET_RA)
    {
        if (!newconf.nwtc_remaddr)
        {
            tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
            reply_thr_get(tcp_fd, EBADMODE, TRUE);
            bf_afree(data);
            return NW_OK;
        }
    }
    else
    {
assert (new_di_flags & NWTC_REMADDR_MASK);
        newconf.nwtc_remaddr= 0;
    }

    /* NWTC_REMPORT_MASK */
    if (!(new_en_flags | new_di_flags) & NWTC_REMPORT_MASK)
    {
        new_en_flags |= (old_en_flags &
            NWTC_REMPORT_MASK);
        new_di_flags |= (old_di_flags &
            NWTC_REMPORT_MASK);
        newconf.nwtc_rempport=
            oldconf.nwtc_rempport;
    }
    else if (new_en_flags & NWTC_SET_RP)
    {
        if (!newconf.nwtc_rempport)
        {
            tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
            reply_thr_get(tcp_fd, EBADMODE, TRUE);
            bf_afree(data);
            return NW_OK;
        }
    }
    else
    {
assert (new_di_flags & NWTC_REMPORT_MASK);
        newconf.nwtc_rempport= 0;
    }

    newconf.nwtc_flags= ((unsigned long)new_di_flags
        << 16) | new_en_flags;
    all_flags= new_en_flags | new_di_flags;

```

```

/* check the access modes */
if ((all_flags & NWTC_LOCPORT_MASK) != NWTC_LP_UNSET)
{
    for (i=0, fd_ptr= tcp_fd_table; i<TCP_FD_NR; i++, fd_ptr++)
    {
        if (fd_ptr == tcp_fd)
            continue;
        if (!(fd_ptr->tf_flags & TFF_INUSE))
            continue;
        if (fd_ptr->tf_port != tcp_fd->tf_port)
            continue;
        flags= fd_ptr->tf_tcpconf.nwtc_flags;
        if ((flags & NWTC_LOCPORT_MASK) == NWTC_LP_UNSET)
            continue;
        if (fd_ptr->tf_tcpconf.nwtc_locport !=
            newconf.nwtc_locport)
            continue;
        if ((flags & NWTC_ACC_MASK) != (all_flags &
            NWTC_ACC_MASK) ||
            (all_flags & NWTC_ACC_MASK) == NWTC_EXCL)
        {
            tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
            reply_thr_get(tcp_fd, EADDRINUSE, TRUE);
            bf_afree(data);
            return NW_OK;
        }
    }
}

tcp_fd->tf_tcpconf= newconf;

if ((all_flags & NWTC_ACC_MASK) &&
    ((all_flags & NWTC_LOCPORT_MASK) == NWTC_LP_SET ||
    (all_flags & NWTC_LOCPORT_MASK) == NWTC_LP_SEL) &&
    (all_flags & NWTC_REMADDR_MASK) &&
    (all_flags & NWTC_REMPORT_MASK))
    tcp_fd->tf_flags |= TFF_CONF_SET;
else
{
    tcp_fd->tf_flags &= ~TFF_CONF_SET;
}
bf_afree(data);
tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
reply_thr_get(tcp_fd, NW_OK, TRUE);
return NW_OK;
}

/*
tcp_setopt
*/

PRIVATE int tcp_setopt(tcp_fd)
tcp_fd_t *tcp_fd;
{
    nwio_tcpopt_t *tcpopt;
    nwio_tcpopt_t oldopt, newopt;
    acc_t *data;
    unsigned int new_en_flags, new_di_flags,
        old_en_flags, old_di_flags;

    data= (*tcp_fd->tf_get_userdata) (tcp_fd->tf_srfd, 0,
        sizeof(nwio_tcpopt_t), TRUE);

    if (!data)
        return EFAULT;

    data= bf_packIfLess(data, sizeof(nwio_tcpopt_t));
    assert (data->acc_length == sizeof(nwio_tcpopt_t));

    tcpopt= (nwio_tcpopt_t *)ptr2acc_data(data);
    oldopt= tcp_fd->tf_tcpopt;
    newopt= *tcpopt;

```

```
old_en_flags= oldopt.nwto_flags & 0xffff;
old_di_flags= (oldopt.nwto_flags >> 16) & 0xffff;
new_en_flags= newopt.nwto_flags & 0xffff;
new_di_flags= (newopt.nwto_flags >> 16) & 0xffff;
if (new_en_flags & new_di_flags)
{
    tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
    reply_thr_get(tcp_fd, EBADMODE, TRUE);
    return NW_OK;
}

/* NWTO_SND_URG_MASK */
if (!(new_en_flags | new_di_flags) & NWTO_SND_URG_MASK)
{
    new_en_flags |= (old_en_flags & NWTO_SND_URG_MASK);
    new_di_flags |= (old_di_flags & NWTO_SND_URG_MASK);
}

/* NWTO_RCV_URG_MASK */
if (!(new_en_flags | new_di_flags) & NWTO_RCV_URG_MASK)
{
    new_en_flags |= (old_en_flags & NWTO_RCV_URG_MASK);
    new_di_flags |= (old_di_flags & NWTO_RCV_URG_MASK);
}

/* NWTO_BSD_URG_MASK */
if (!(new_en_flags | new_di_flags) & NWTO_BSD_URG_MASK)
{
    new_en_flags |= (old_en_flags & NWTO_BSD_URG_MASK);
    new_di_flags |= (old_di_flags & NWTO_BSD_URG_MASK);
}
else
{
    if (tcp_fd->tf_conn == NULL)
    {
        bf_afree(data);
        tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
        reply_thr_get(tcp_fd, EINVAL, TRUE);
        return NW_OK;
    }
}

/* NWTO_DEL_RST_MASK */
if (!(new_en_flags | new_di_flags) & NWTO_DEL_RST_MASK)
{
    new_en_flags |= (old_en_flags & NWTO_DEL_RST_MASK);
    new_di_flags |= (old_di_flags & NWTO_DEL_RST_MASK);
}

/* NWTO_BULK_MASK */
if (!(new_en_flags | new_di_flags) & NWTO_BULK_MASK)
{
    new_en_flags |= (old_en_flags & NWTO_BULK_MASK);
    new_di_flags |= (old_di_flags & NWTO_BULK_MASK);
}

newopt.nwto_flags= ((unsigned long)new_di_flags << 16) |
    new_en_flags;
tcp_fd->tf_tcpopt= newopt;
if (newopt.nwto_flags & NWTO_SND_URG)
    tcp_fd->tf_flags |= TFF_WR_URG;
else
    tcp_fd->tf_flags &= ~TFF_WR_URG;

if (newopt.nwto_flags & NWTO_RCV_URG)
    tcp_fd->tf_flags |= TFF_RECV_URG;
else
    tcp_fd->tf_flags &= ~TFF_RECV_URG;

if (tcp_fd->tf_conn)
{
    if (newopt.nwto_flags & NWTO_BSD_URG)
        tcp_fd->tf_conn->tc_flags |= TCF_BSD_URG;
    else
```

```

        tcp_fd->tf_conn->tc_flags &= ~TCF_BSD_URG;
    }

    if (newopt.nwto_flags & NWTO_DEL_RST)
        tcp_fd->tf_flags |= TFF_DEL_RST;
    else
        tcp_fd->tf_flags &= ~TFF_DEL_RST;

    if (newopt.nwto_flags & NWTO_BULK)
        tcp_fd->tf_flags &= ~TFF_PUSH_DATA;
    else
        tcp_fd->tf_flags |= TFF_PUSH_DATA;

    bf_afree(data);
    tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
    reply_thr_get(tcp_fd, NW_OK, TRUE);
    return NW_OK;
}

PRIVATE tcpport_t find_unused_port(fd)
int fd;
{
    tcpport_t port, nw_port;

    for (port= 0x8000+fd; port < 0xffff-TCP_FD_NR; port+= TCP_FD_NR)
    {
        nw_port= htons(port);
        if (is_unused_port(nw_port))
            return nw_port;
    }
    for (port= 0x8000; port < 0xffff; port++)
    {
        nw_port= htons(port);
        if (is_unused_port(nw_port))
            return nw_port;
    }
    ip_panic(( "unable to find unused port (shouldn't occur)" ));
    return 0;
}

PRIVATE int is_unused_port(port)
tcpport_t port;
{
    int i;
    tcp_fd_t *tcp_fd;
    tcp_conn_t *tcp_conn;

    for (i= 0, tcp_fd= tcp_fd_table; i<TCP_FD_NR; i++,
        tcp_fd++)
    {
        if (!(tcp_fd->tf_flags & TFF_CONF_SET))
            continue;
        if (tcp_fd->tf_tcpconf.nwtc_locport == port)
            return FALSE;
    }
    for (i= tcp_conf_nr, tcp_conn= tcp_conn_table+i;
        i<TCP_CONN_NR; i++, tcp_conn++)
        /* the first tcp_conf_nr ports are special */
    {
        if (!(tcp_conn->tc_flags & TCF_INUSE))
            continue;
        if (tcp_conn->tc_locport == port)
            return FALSE;
    }
    return TRUE;
}

PRIVATE int reply_thr_put(tcp_fd, reply, for_ioctl)
tcp_fd_t *tcp_fd;
int reply;
int for_ioctl;
{
    assert (tcp_fd);

```

```

    return (*tcp_fd->tf_put_userdata)(tcp_fd->tf_srfd, reply,
        (acc_t *)0, for_ioctl);
}

PRIVATE void reply_thr_get(tcp_fd, reply, for_ioctl)
tcp_fd_t *tcp_fd;
int reply;
int for_ioctl;
{
    acc_t *result;

    result= (*tcp_fd->tf_get_userdata)(tcp_fd->tf_srfd, reply,
        (size_t)0, for_ioctl);
    assert (!result);
}

PUBLIC int tcp_su4listen(tcp_fd, tcp_conn, do_listenq)
tcp_fd_t *tcp_fd;
tcp_conn_t *tcp_conn;
int do_listenq;
{
    tcp_conn->tc_locport= tcp_fd->tf_tcpconf.nwtc_locport;
    tcp_conn->tc_locaddr= tcp_fd->tf_port->tp_ipaddr;
    if (tcp_fd->tf_tcpconf.nwtc_flags & NWTC_SET_RP)
        tcp_conn->tc_rempport= tcp_fd->tf_tcpconf.nwtc_rempport;
    else
        tcp_conn->tc_rempport= 0;
    if (tcp_fd->tf_tcpconf.nwtc_flags & NWTC_SET_RA)
        tcp_conn->tc_remaddr= tcp_fd->tf_tcpconf.nwtc_remaddr;
    else
        tcp_conn->tc_remaddr= 0;

    tcp_setup_conn(tcp_fd->tf_port, tcp_conn);
    tcp_conn->tc_fd= tcp_fd;
    tcp_conn->tc_connInprogress= 1;
    tcp_conn->tc_orglisten= TRUE;
    tcp_conn->tc_state= TCS_LISTEN;
    tcp_conn->tc_rt_dead= TCP_DEF_RT_MAX_LISTEN;
    if (do_listenq)
    {
        tcp_fd->tf_flags |= TFF_LISTENQ;
        tcp_reply_ioctl(tcp_fd, NW_OK);
        return NW_OK;
    }
    return NW_SUSPEND;
}

/*
find_empty_conn

This function returns a connection that is not inuse.
This includes connections that are never used, and connections without a
user that are not used for a while.
*/

PRIVATE tcp_conn_t *find_empty_conn()
{
    int i;
    tcp_conn_t *tcp_conn;

    for (i=tcp_conf_nr, tcp_conn= tcp_conn_table+i;
        i<TCP_CONN_NR; i++, tcp_conn++)
        /* the first tcp_conf_nr connections are reserved for
         * RSTs
         */
        {
            if (tcp_conn->tc_flags == TCF_EMPTY)
            {
                tcp_conn->tc_connInprogress= 0;
                tcp_conn->tc_fd= NULL;
                return tcp_conn;
            }
            if (tcp_conn->tc_fd)

```



```

        continue;
    if (tcp_conn->tc_senddis > get_time())
        continue;
    if (tcp_conn->tc_state != TCS_CLOSED)
    {
        tcp_close_connection (tcp_conn, ENOCONN);
    }
    tcp_conn->tc_flags= 0;
    return tcp_conn;
}
return NULL;
}

/*
find_conn_entry

This function return a connection matching locport, locaddr, remport, remaddr.
If no such connection exists NULL is returned.
If a connection exists without mainuser it is closed.
*/

PRIVATE tcp_conn_t *find_conn_entry(locport, locaddr, remport, remaddr)
tcpport_t locport;
ipaddr_t locaddr;
tcpport_t remport;
ipaddr_t remaddr;
{
    tcp_conn_t *tcp_conn;
    int i, state;

    assert(remport);
    assert(remaddr);
    for (i=tcp_conf_nr, tcp_conn= tcp_conn_table+i; i<TCP_CONN_NR;
        i++, tcp_conn++)
        /* the first tcp_conf_nr connections are reserved for
           RSTs */
        {
            if (tcp_conn->tc_flags == TCF_EMPTY)
                continue;
            if (tcp_conn->tc_locport != locport ||
                tcp_conn->tc_locaddr != locaddr ||
                tcp_conn->tc_remport != remport ||
                tcp_conn->tc_remaddr != remaddr)
                continue;
            if (tcp_conn->tc_fd)
                return tcp_conn;
            state= tcp_conn->tc_state;
            if (state != TCS_CLOSED)
            {
                tcp_close_connection(tcp_conn, ENOCONN);
            }
            return tcp_conn;
        }
    return NULL;
}

PRIVATE void read_ip_packets(tcp_port)
tcp_port_t *tcp_port;
{
    int result;

    do
    {
        tcp_port->tp_flags |= TPF_READ_IP;
        result= ip_read(tcp_port->tp_ipfd, TCP_MAX_DATAGRAM);
        if (result == NW_SUSPEND)
        {
            tcp_port->tp_flags |= TPF_READ_SP;
            return;
        }
        assert(result == NW_OK);
        tcp_port->tp_flags &= ~TPF_READ_IP;
    } while(!(tcp_port->tp_flags & TPF_READ_IP));
}

```

```
}

/*
find_best_conn
*/

PRIVATE tcp_conn_t *find_best_conn(ip_hdr, tcp_hdr)
ip_hdr_t *ip_hdr;
tcp_hdr_t *tcp_hdr;
{
    int best_level, new_level;
    tcp_conn_t *best_conn, *listen_conn, *tcp_conn;
    tcp_fd_t *tcp_fd;
    int i;
    ipaddr_t locaddr;
    ipaddr_t remaddr;
    tcpport_t locport;
    tcpport_t remport;

    locaddr= ip_hdr->ih_dst;
    remaddr= ip_hdr->ih_src;
    locport= tcp_hdr->th_dstport;
    remport= tcp_hdr->th_srcport;
    if (!remport) /* This can interfere with a listen, so we reject it
                  * by clearing the requested port
                  */
        locport= 0;

    best_level= 0;
    best_conn= NULL;
    listen_conn= NULL;
    for (i= tcp_conf_nr, tcp_conn= tcp_conn_table+i;
        i<TCP_CONN_NR; i++, tcp_conn++)
        /* the first tcp_conf_nr connections are reserved for
           RSTs */
    {
        if (!(tcp_conn->tc_flags & TCF_INUSE))
            continue;
        /* First fast check for open connections. */
        if (tcp_conn->tc_locaddr == locaddr &&
            tcp_conn->tc_locport == locport &&
            tcp_conn->tc_remport == remport &&
            tcp_conn->tc_remaddr == remaddr &&
            tcp_conn->tc_fd)
        {
            return tcp_conn;
        }

        /* Now check for listens and abandoned connections. */
        if (tcp_conn->tc_locaddr != locaddr)
        {
            continue;
        }
        new_level= 0;
        if (tcp_conn->tc_locport)
        {
            if (tcp_conn->tc_locport != locport)
            {
                continue;
            }
            new_level += 4;
        }
        if (tcp_conn->tc_remport)
        {
            if (tcp_conn->tc_remport != remport)
            {
                continue;
            }
            new_level += 1;
        }
        if (tcp_conn->tc_remaddr)
        {
            if (tcp_conn->tc_remaddr != remaddr)

```

```

        {
            continue;
        }
        new_level += 2;
    }
    if (new_level < best_level)
        continue;
    if (new_level != 7 && tcp_conn->tc_state != TCS_LISTEN)
        continue;
    if (new_level == 7)
        /* We found an abandoned connection */
        {
            assert(!tcp_conn->tc_fd);
            if (best_conn && tcp_conn->tc_ISS <
                best_conn->tc_ISS)
            {
                continue;
            }
            best_conn = tcp_conn;
            continue;
        }
    if (!(tcp_hdr->th_flags & THF_SYN))
        continue;
    best_level = new_level;
    listen_conn = tcp_conn;
    assert(listen_conn->tc_fd != NULL);
}

if (listen_conn && listen_conn->tc_fd->tf_flags & TFF_LISTENQ &&
    listen_conn->tc_fd->tf_conn == listen_conn)
{
    /* Special processing for listen queues. Only accept the
     * connection if there is empty space in the queue and
     * there are empty connections as well.
     */
    listen_conn = new_conn_for_queue(listen_conn->tc_fd);
}

if (!best_conn && !listen_conn)
{
    if ((tcp_hdr->th_flags & THF_SYN) &&
        maybe_listen(locaddr, locport, remaddr, remport))
    {
        /* Quick hack to implement listen back logs:
         * if a SYN arrives and there is no listen waiting
         * for that packet, then no reply is sent.
         */
        return NULL;
    }

    for (i=0, tcp_conn = tcp_conn_table; i < tcp_conf_nr;
        i++, tcp_conn++)
    {
        /* find valid port to send RST */
        if ((tcp_conn->tc_flags & TCF_INUSE) &&
            tcp_conn->tc_locaddr == locaddr)
        {
            break;
        }
    }
    assert(tcp_conn);
    assert(tcp_conn->tc_state == TCS_CLOSED);

    tcp_conn->tc_locport = locport;
    tcp_conn->tc_locaddr = locaddr;
    tcp_conn->tc_remport = remport;
    tcp_conn->tc_remaddr = remaddr;
    assert(!tcp_conn->tc_fd);
    return tcp_conn;
}

if (best_conn)
{
    if (!listen_conn)

```

```

        {
            assert(!best_conn->tc_fd);
            return best_conn;
        }

    assert(listen_conn->tc_connInProgress);
    tcp_fd= listen_conn->tc_fd;
    assert(tcp_fd);
    assert((tcp_fd->tf_flags & TFF_LISTENQ) ||
            tcp_fd->tf_conn == listen_conn);

    if (best_conn->tc_state != TCS_CLOSED)
        tcp_close_connection(best_conn, ENOCONN);

    listen_conn->tc_ISS= best_conn->tc_ISS;
    if (best_conn->tc_senddis > listen_conn->tc_senddis)
        listen_conn->tc_senddis= best_conn->tc_senddis;
    return listen_conn;
}
assert (listen_conn);
return listen_conn;
}

/*
new_conn_for_queue
*/
PRIVATE tcp_conn_t *new_conn_for_queue(tcp_fd)
tcp_fd_t *tcp_fd;
{
    int i;
    tcp_conn_t *tcp_conn;

    assert(tcp_fd->tf_flags & TFF_LISTENQ);

    for (i= 0; i<TFL_LISTEN_MAX; i++)
    {
        if (tcp_fd->tf_listenq[i] == NULL)
            break;
    }
    if (i >= TFL_LISTEN_MAX)
        return NULL;

    tcp_conn= find_empty_conn();
    if (!tcp_conn)
        return NULL;
    tcp_fd->tf_listenq[i]= tcp_conn;
    (void)tcp_su4listen(tcp_fd, tcp_conn, 0 /* !do_listenq */);
    return tcp_conn;
}

/*
maybe_listen
*/
PRIVATE int maybe_listen(locaddr, locport, remaddr, remport)
ipaddr_t locaddr;
tcpport_t locport;
ipaddr_t remaddr;
tcpport_t remport;
{
    int i;
    tcp_conn_t *tcp_conn;
    tcp_fd_t *fd;

    for (i= tcp_conf_nr, tcp_conn= tcp_conn_table+i;
         i<TCP_CONN_NR; i++, tcp_conn++)
    {
        if (!(tcp_conn->tc_flags & TCF_INUSE))
            continue;

        if (tcp_conn->tc_locaddr != locaddr)
        {
            continue;
        }
        if (tcp_conn->tc_locport != locport )

```

```

        {
            continue;
        }
        if (!tcp_conn->tc_orglisten)
            continue;
        fd= tcp_conn->tc_fd;
        if (!fd)
            continue;
        if ((fd->tf_tcpconf.nwtc_flags & NWTC_SET_RP) &&
            tcp_conn->tc_rempport != remport)
        {
            continue;
        }
        if ((fd->tf_tcpconf.nwtc_flags & NWTC_SET_RA) &&
            tcp_conn->tc_remaddr != remaddr)
        {
            continue;
        }
        if (!(fd->tf_flags & TFF_DEL_RST))
            continue;
        return 1;
    }
    return 0;
}

PUBLIC void tcp_reply_ioctl(tcp_fd, reply)
tcp_fd_t *tcp_fd;
int reply;
{
    assert (tcp_fd->tf_flags & TFF_IOCTL_IP);
    assert (tcp_fd->tf_ioreq == NWIOTCPSHUTDOWN ||
            tcp_fd->tf_ioreq == NWIOTCPLISTEN ||
            tcp_fd->tf_ioreq == NWIOTCPLISTENQ ||
            tcp_fd->tf_ioreq == NWIOTCPACCEPTTO ||
            tcp_fd->tf_ioreq == NWIOTCPCONN);

    tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
    reply_thr_get (tcp_fd, reply, TRUE);
}

PUBLIC void tcp_reply_write(tcp_fd, reply)
tcp_fd_t *tcp_fd;
size_t reply;
{
    assert (tcp_fd->tf_flags & TFF_WRITE_IP);

    tcp_fd->tf_flags &= ~TFF_WRITE_IP;
    reply_thr_get (tcp_fd, reply, FALSE);
}

PUBLIC void tcp_reply_read(tcp_fd, reply)
tcp_fd_t *tcp_fd;
size_t reply;
{
    assert (tcp_fd->tf_flags & TFF_READ_IP);

    tcp_fd->tf_flags &= ~TFF_READ_IP;
    reply_thr_put (tcp_fd, reply, FALSE);
}

PUBLIC int tcp_write(fd, count)
int fd;
size_t count;
{
    tcp_fd_t *tcp_fd;
    tcp_conn_t *tcp_conn;

    tcp_fd= &tcp_fd_table[fd];

    assert (tcp_fd->tf_flags & TFF_INUSE);

    if (!(tcp_fd->tf_flags & TFF_CONNECTED))

```

```

    {
        reply_thr_get (tcp_fd, ENOTCONN, FALSE);
        return NW_OK;
    }
    tcp_conn= tcp_fd->tf_conn;
    if (tcp_conn->tc_state == TCS_CLOSED)
    {
        reply_thr_get(tcp_fd, tcp_conn->tc_error, FALSE);
        return NW_OK;
    }
    if (tcp_conn->tc_flags & TCF_FIN_SENT)
    {
        reply_thr_get (tcp_fd, ESHUTDOWN, FALSE);
        return NW_OK;
    }

    tcp_fd->tf_flags |= TFF_WRITE_IP;
    tcp_fd->tf_write_offset= 0;
    tcp_fd->tf_write_count= count;

    /* New data may cause a segment to be sent. Clear PUSH_NOW
     * from last NWIOTCPPUSH ioctl.
     */
    tcp_conn->tc_flags &= ~(TCF_NO_PUSH|TCF_PUSH_NOW);

    /* Start the timer (if necessary) */
    if (tcp_conn->tc_SND_TRM == tcp_conn->tc_SND_UNA)
        tcp_set_send_timer(tcp_conn);

    assert(tcp_conn->tc_busy == 0);
    tcp_conn->tc_busy++;
    tcp_fd_write(tcp_conn);
    tcp_conn->tc_busy--;
    tcp_conn_write(tcp_conn, 0);

    if (!(tcp_fd->tf_flags & TFF_WRITE_IP))
        return NW_OK;
    else
        return NW_SUSPEND;
}

PUBLIC int
tcp_read(fd, count)
int fd;
size_t count;
{
    tcp_fd_t *tcp_fd;
    tcp_conn_t *tcp_conn;

    tcp_fd= &tcp_fd_table[fd];

    assert (tcp_fd->tf_flags & TFF_INUSE);

    if (!(tcp_fd->tf_flags & TFF_CONNECTED))
    {
        reply_thr_put (tcp_fd, ENOTCONN, FALSE);
        return NW_OK;
    }
    tcp_conn= tcp_fd->tf_conn;

    tcp_fd->tf_flags |= TFF_READ_IP;
    tcp_fd->tf_read_offset= 0;
    tcp_fd->tf_read_count= count;

    assert(tcp_conn->tc_busy == 0);
    tcp_conn->tc_busy++;
    tcp_fd_read(tcp_conn, 0);
    tcp_conn->tc_busy--;
    if (!(tcp_fd->tf_flags & TFF_READ_IP))
        return NW_OK;
    else
        return NW_SUSPEND;
}

```

```

/*
tcp_restart_connect

reply the success or failure of a connect to the user.
*/

PUBLIC void tcp_restart_connect(tcp_conn)
tcp_conn_t *tcp_conn;
{
    tcp_fd_t *tcp_fd;
    int reply;

    assert(tcp_conn->tc_connInprogress);
    tcp_conn->tc_connInprogress = 0;

    tcp_fd = tcp_conn->tc_fd;
    assert(tcp_fd);
    if (tcp_fd->tf_flags & TFF_LISTENQ)
    {
        /* Special code for listen queues */
        assert(tcp_conn->tc_state != TCS_CLOSED);

        /* Reply for select */
        if ((tcp_fd->tf_flags & TFF_SEL_READ) &&
            tcp_fd->tf_select_res)
        {
            tcp_fd->tf_flags &= ~TFF_SEL_READ;
            tcp_fd->tf_select_res(tcp_fd->tf_srfd,
                                SR_SELECT_READ);
        }

        /* Reply for acceptto */
        if (tcp_fd->tf_flags & TFF_IOCTL_IP)
            (void) tcp_acceptto(tcp_fd);

        return;
    }

    if (tcp_conn->tc_state == TCS_CLOSED)
    {
        reply = tcp_conn->tc_error;
        assert(tcp_conn->tc_fd == tcp_fd);
        tcp_fd->tf_conn = NULL;
        tcp_conn->tc_fd = NULL;
        tcp_fd->tf_error = reply;
    }
    else
    {
        tcp_fd->tf_flags |= TFF_CONNECTED;
        reply = NW_OK;
    }

    if (tcp_fd->tf_flags & TFF_CONNECTING)
    {
        /* Special code for asynchronous connects */
        tcp_fd->tf_flags &= ~TFF_CONNECTING;

        /* Reply for select */
        if ((tcp_fd->tf_flags & TFF_SEL_WRITE) &&
            tcp_fd->tf_select_res)
        {
            tcp_fd->tf_flags &= ~TFF_SEL_WRITE;
            tcp_fd->tf_select_res(tcp_fd->tf_srfd,
                                SR_SELECT_WRITE);
        }

        return;
    }

    assert(tcp_fd->tf_flags & TFF_IOCTL_IP);
    assert(tcp_fd->tf_ioreq == NWIOTCPLISTEN ||
           tcp_fd->tf_ioreq == NWIOTCPCONN);
}

```

```

    tcp_reply_ioctl (tcp_fd, reply);
}

/*
tcp_close
*/
PUBLIC void tcp_close(fd)
int fd;
{
    int i;
    tcp_fd_t *tcp_fd;
    tcp_conn_t *tcp_conn;

    tcp_fd= &tcp_fd_table[fd];

    assert (tcp_fd->tf_flags & TFF_INUSE);
    assert (!(tcp_fd->tf_flags &
        (TFF_IOCTL_IP|TFF_READ_IP|TFF_WRITE_IP)));

    if (tcp_fd->tf_flags & TFF_LISTENQ)
    {
        /* Special code for listen queues */
        for (i= 0; i<TFL_LISTEN_MAX; i++)
        {
            tcp_conn= tcp_fd->tf_listenq[i];
            if (!tcp_conn)
                continue;

            tcp_fd->tf_listenq[i]= NULL;
            assert(tcp_conn->tc_fd == tcp_fd);
            tcp_conn->tc_fd= NULL;

            if (tcp_conn->tc_connInProgress)
            {
                tcp_conn->tc_connInProgress= 0;
                tcp_close_connection(tcp_conn, ENOCONN);
                continue;
            }

            tcp_shutdown (tcp_conn);
            if (tcp_conn->tc_state == TCS_ESTABLISHED)
                tcp_conn->tc_state= TCS_CLOSING;

            /* Set the retransmission timeout a bit smaller. */
            tcp_conn->tc_rt_dead= TCP_DEF_RT_MAX_CLOSING;

            /* If all data has been acknowledged, close the connection. */
            if (tcp_conn->tc_SND_UNA == tcp_conn->tc_SND_NXT)
                tcp_close_connection(tcp_conn, ENOTCONN);
        }

        tcp_conn= tcp_fd->tf_conn;
        assert(tcp_conn->tc_fd == tcp_fd);
        assert (tcp_conn->tc_connInProgress);
        tcp_conn->tc_connInProgress= 0;
        tcp_conn->tc_fd= NULL;
        tcp_fd->tf_conn= NULL;
        tcp_close_connection(tcp_conn, ENOCONN);
    }
    for (i= 0; i<TFL_LISTEN_MAX; i++)
    {
        assert(tcp_fd->tf_listenq[i] == NULL);
    }

    if (tcp_fd->tf_flags & TFF_CONNECTING)
    {
        tcp_conn= tcp_fd->tf_conn;
        assert(tcp_conn != NULL);

        assert (tcp_conn->tc_connInProgress);
        tcp_conn->tc_connInProgress= 0;
        tcp_conn->tc_fd= NULL;
        tcp_fd->tf_conn= NULL;
    }
}

```



```

        tcp_close_connection(tcp_conn, ENOCONN);

        tcp_fd->tf_flags &= ~TFF_CONNECTING;
    }

    tcp_fd->tf_flags &= ~TFF_INUSE;
    if (!tcp_fd->tf_conn)
        return;

    tcp_conn= tcp_fd->tf_conn;
    assert(tcp_conn->tc_fd == tcp_fd);
    tcp_conn->tc_fd= NULL;

    assert (!tcp_conn->tc_connInProgress);

    tcp_shutdown (tcp_conn);
    if (tcp_conn->tc_state == TCS_ESTABLISHED)
    {
        tcp_conn->tc_state= TCS_CLOSING;
    }

    /* Set the retransmission timeout a bit smaller. */
    tcp_conn->tc_rt_dead= TCP_DEF_RT_MAX_CLOSING;

    /* If all data has been acknowledged, close the connection. */
    if (tcp_conn->tc_SND_UNA == tcp_conn->tc_SND_NXT)
        tcp_close_connection(tcp_conn, ENOTCONN);
}

PUBLIC int tcp_cancel(fd, which_operation)
int fd;
int which_operation;
{
    tcp_fd_t *tcp_fd;
    tcp_conn_t *tcp_conn;

    tcp_fd= &tcp_fd_table[fd];

    assert (tcp_fd->tf_flags & TFF_INUSE);

    tcp_conn= tcp_fd->tf_conn;

    switch (which_operation)
    {
    case SR_CANCEL_WRITE:
        assert (tcp_fd->tf_flags & TFF_WRITE_IP);
        tcp_fd->tf_flags &= ~TFF_WRITE_IP;

        if (tcp_fd->tf_write_offset)
            reply_thr_get (tcp_fd, tcp_fd->tf_write_offset, FALSE);
        else
            reply_thr_get (tcp_fd, EINTR, FALSE);
        break;
    case SR_CANCEL_READ:
        assert (tcp_fd->tf_flags & TFF_READ_IP);
        tcp_fd->tf_flags &= ~TFF_READ_IP;
        if (tcp_fd->tf_read_offset)
            reply_thr_put (tcp_fd, tcp_fd->tf_read_offset, FALSE);
        else
            reply_thr_put (tcp_fd, EINTR, FALSE);
        break;
    case SR_CANCEL_IOCTL:
        assert (tcp_fd->tf_flags & TFF_IOCTL_IP);
        tcp_fd->tf_flags &= ~TFF_IOCTL_IP;

        if (tcp_fd->tf_flags & TFF_IOC_INIT_SP)
        {
            tcp_fd->tf_flags &= ~TFF_IOC_INIT_SP;
            reply_thr_put (tcp_fd, EINTR, TRUE);
            break;
        }

        switch (tcp_fd->tf_ioreq)

```

```

        {
            case NWIOGTCPCONF:
                reply_thr_put (tcp_fd, EINTR, TRUE);
                break;
            case NWIOSTCPCONF:
            case NWIOTCPSHUTDOWN:
                reply_thr_get (tcp_fd, EINTR, TRUE);
                break;
            case NWIOTCPCONN:
            case NWIOTCPLISTEN:
                assert (tcp_conn->tc_connInprogress);
                tcp_conn->tc_connInprogress= 0;
                tcp_conn->tc_fd= NULL;
                tcp_fd->tf_conn= NULL;
                tcp_close_connection(tcp_conn, ENOCONN);
                reply_thr_get (tcp_fd, EINTR, TRUE);
                break;
            default:
                ip_warning(( "unknown ioctl inprogress: 0x%x",
                            tcp_fd->tf_ioreq ));
                reply_thr_get (tcp_fd, EINTR, TRUE);
                break;
        }
        break;
    default:
        ip_panic(( "unknown cancel request" ));
        break;
    }
    return NW_OK;
}

/*
tcp_connect
*/

PRIVATE int tcp_connect(tcp_fd)
tcp_fd_t *tcp_fd;
{
    tcp_conn_t *tcp_conn;
    nwio_tcpcl_t *tcpcl;
    long nwtcl_flags;
    int r, do_asynch;
    acc_t *data;

    if (!(tcp_fd->tf_flags & TFF_CONF_SET))
    {
        tcp_reply_ioctl(tcp_fd, EBADMODE);
        return NW_OK;
    }
    assert (!(tcp_fd->tf_flags & TFF_CONNECTED) &&
            !(tcp_fd->tf_flags & TFF_CONNECTING) &&
            !(tcp_fd->tf_flags & TFF_LISTENQ));
    if ((tcp_fd->tf_tcpconf.nwtc_flags & (NWTC_SET_RA|NWTC_SET_RP))
        != (NWTC_SET_RA|NWTC_SET_RP))
    {
        tcp_reply_ioctl(tcp_fd, EBADMODE);
        return NW_OK;
    }

    data= (*tcp_fd->tf_get_userdata) (tcp_fd->tf_srfd, 0,
                                     sizeof(*tcpcl), TRUE);
    if (!data)
        return EFAULT;

    data= bf_packIfLess(data, sizeof(*tcpcl));
    assert (data->acc_length == sizeof(*tcpcl));
    tcpcl= (nwio_tcpcl_t *)ptr2acc_data(data);

    nwtcl_flags= tcpcl->nwtcl_flags;
    bf_afree(data); data= NULL; tcpcl= NULL;

    if (nwtcl_flags == TCF_ASYNC)
        do_asynch= 1;
    else if (nwtcl_flags == TCF_DEFAULT)

```

```

        do_asynch= 0;
    else
    {
        tcp_reply_ioctl(tcp_fd, EINVAL);
        return NW_OK;
    }

    assert(!tcp_fd->tf_conn);
    tcp_conn= find_conn_entry(tcp_fd->tf_tcpconf.nwtc_locport,
        tcp_fd->tf_port->tp_ipaddr,
        tcp_fd->tf_tcpconf.nwtc_rempport,
        tcp_fd->tf_tcpconf.nwtc_remaddr);
    if (tcp_conn)
    {
        if (tcp_conn->tc_fd)
        {
            tcp_reply_ioctl(tcp_fd, EADDRINUSE);
            return NW_OK;
        }
    }
    else
    {
        tcp_conn= find_empty_conn();
        if (!tcp_conn)
        {
            tcp_reply_ioctl(tcp_fd, EAGAIN);
            return NW_OK;
        }
    }
    tcp_fd->tf_conn= tcp_conn;

    r= tcp_su4connect(tcp_fd);
    if (r == NW_SUSPEND && do_asynch)
    {
        tcp_fd->tf_flags |= TFF_CONNECTING;
        tcp_reply_ioctl(tcp_fd, EINPROGRESS);
        r= NW_OK;
    }
    return r;
}

/*
tcp_su4connect
*/

PRIVATE int tcp_su4connect(tcp_fd)
tcp_fd_t *tcp_fd;
{
    tcp_conn_t *tcp_conn;

    tcp_conn= tcp_fd->tf_conn;

    tcp_conn->tc_locport= tcp_fd->tf_tcpconf.nwtc_locport;
    tcp_conn->tc_locaddr= tcp_fd->tf_port->tp_ipaddr;

    assert (tcp_fd->tf_tcpconf.nwtc_flags & NWTc_SET_RP);
    assert (tcp_fd->tf_tcpconf.nwtc_flags & NWTc_SET_RA);
    tcp_conn->tc_rempport= tcp_fd->tf_tcpconf.nwtc_rempport;
    tcp_conn->tc_remaddr= tcp_fd->tf_tcpconf.nwtc_remaddr;

    tcp_setup_conn(tcp_fd->tf_port, tcp_conn);

    tcp_conn->tc_fd= tcp_fd;
    tcp_conn->tc_connInProgress= 1;
    tcp_conn->tc_orglisten= FALSE;
    tcp_conn->tc_state= TCS_SYN_SENT;
    tcp_conn->tc_rt_dead= TCP_DEF_RT_MAX_CONNECT;

    /* Start the timer (if necessary) */
    tcp_set_send_timer(tcp_conn);

    tcp_conn_write(tcp_conn, 0);

    if (tcp_conn->tc_connInProgress)

```

```

        return NW_SUSPEND;
    else
        return NW_OK;
}

/*
tcp_listen
*/
PRIVATE int tcp_listen(tcp_fd, do_listenq)
tcp_fd_t *tcp_fd;
int do_listenq;
{
    tcp_conn_t *tcp_conn;

    if (!(tcp_fd->tf_flags & TFF_CONF_SET))
    {
        tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
        reply_thr_get(tcp_fd, EBADMODE, TRUE);
        return NW_OK;
    }
    assert (!(tcp_fd->tf_flags & TFF_CONNECTED) &&
            !(tcp_fd->tf_flags & TFF_CONNECTING) &&
            !(tcp_fd->tf_flags & TFF_LISTENQ));
    tcp_conn= tcp_fd->tf_conn;
    assert(!tcp_conn);

    if ((tcp_fd->tf_tcpconf.nwtc_flags & (NWTC_SET_RA|NWTC_SET_RP))
        == (NWTC_SET_RA|NWTC_SET_RP))
    {
        tcp_conn= find_conn_entry(
            tcp_fd->tf_tcpconf.nwtc_locport,
            tcp_fd->tf_port->tp_ipaddr,
            tcp_fd->tf_tcpconf.nwtc_rempport,
            tcp_fd->tf_tcpconf.nwtc_remaddr);
        if (tcp_conn)
        {
            if (tcp_conn->tc_fd)
            {
                tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
                reply_thr_get (tcp_fd, EADDRINUSE, TRUE);
                return NW_OK;
            }
            tcp_fd->tf_conn= tcp_conn;
            return tcp_su4listen(tcp_fd, tcp_conn, do_listenq);
        }
    }
    tcp_conn= find_empty_conn();
    if (!tcp_conn)
    {
        tcp_fd->tf_flags &= ~TFF_IOCTL_IP;
        reply_thr_get (tcp_fd, EAGAIN, TRUE);
        return NW_OK;
    }
    tcp_fd->tf_conn= tcp_conn;
    return tcp_su4listen(tcp_fd, tcp_conn, do_listenq);
}

/*
tcp_acceptto
*/
PRIVATE int tcp_acceptto(tcp_fd)
tcp_fd_t *tcp_fd;
{
    int i, dst_nr;
    tcp_fd_t *dst_fd;
    tcp_conn_t *tcp_conn;
    tcp_cookie_t *cookiep;
    acc_t *data;
    tcp_cookie_t cookie;

    if (!(tcp_fd->tf_flags & TFF_LISTENQ))

```

```

{
    tcp_reply_ioctl(tcp_fd, EINVAL);
    return NW_OK;
}
for (i= 0; i<TFL_LISTEN_MAX; i++)
{
    tcp_conn= tcp_fd->tf_listenq[i];
    if (tcp_conn && !tcp_conn->tc_connInProgress)
        break;
}
if (i >= TFL_LISTEN_MAX)
{
    /* Nothing, suspend caller */
    return NW_SUSPEND;
}

data= (*tcp_fd->tf_get_userdata) (tcp_fd->tf_srfd, 0,
    sizeof(*cookiep), TRUE);
if (!data)
    return EFAULT;

data= bf_packIfLess(data, sizeof(*cookiep));
cookiep= (tcp_cookie_t *)ptr2acc_data(data);
cookie= *cookiep;

bf_afree(data); data= NULL;

dst_nr= cookie.tc_ref;
if (dst_nr < 0 || dst_nr >= TCP_FD_NR)
{
    printf("tcp_acceptto: bad fd %d\n", dst_nr);
    tcp_reply_ioctl(tcp_fd, EINVAL);
    return NW_OK;
}
dst_fd= &tcp_fd_table[dst_nr];
if (!(dst_fd->tf_flags & TFF_INUSE) ||
    (dst_fd->tf_flags & (TFF_READ_IP|TFF_WRITE_IP|TFF_IOCTL_IP)) ||
    dst_fd->tf_conn != NULL ||
    !(dst_fd->tf_flags & TFF_COOKIE))
{
    printf("tcp_acceptto: bad flags 0x%x or conn %p for fd %d\n",
        dst_fd->tf_flags, dst_fd->tf_conn, dst_nr);
    tcp_reply_ioctl(tcp_fd, EINVAL);
    return NW_OK;
}
if (memcmp(&cookie, &dst_fd->tf_cookie, sizeof(cookie)) != 0)
{
    printf("tcp_acceptto: bad cookie\n");
    return NW_OK;
}

/* Move connection */
tcp_fd->tf_listenq[i]= NULL;
tcp_conn->tc_fd= dst_fd;
dst_fd->tf_conn= tcp_conn;
dst_fd->tf_flags |= TFF_CONNECTED;

tcp_reply_ioctl(tcp_fd, NW_OK);
return NW_OK;
}

```

```

PRIVATE void tcp_buffree (priority)
int priority;
{
    int i;
    tcp_conn_t *tcp_conn;

    if (priority == TCP_PRI_FRAG2SEND)
    {
        for (i=0, tcp_conn= tcp_conn_table; i<TCP_CONN_NR; i++,
            tcp_conn++)
        {
            if (!(tcp_conn->tc_flags & TCF_INUSE))

```

```

        continue;
    if (!tcp_conn->tc_frag2send)
        continue;
    if (tcp_conn->tc_busy)
        continue;
    bf_afree(tcp_conn->tc_frag2send);
    tcp_conn->tc_frag2send= 0;
}

}

if (priority == TCP_PRI_CONN_EXTRA)
{
    for (i=0, tcp_conn= tcp_conn_table; i<TCP_CONN_NR; i++,
        tcp_conn++)
    {
        if (!(tcp_conn->tc_flags & TCF_INUSE))
            continue;
        if (tcp_conn->tc_busy)
            continue;
        if (tcp_conn->tc_adv_data)
        {
            bf_afree(tcp_conn->tc_adv_data);
            tcp_conn->tc_adv_data= NULL;
        }
    }
}

if (priority == TCP_PRI_CONNwoUSER)
{
    for (i=0, tcp_conn= tcp_conn_table; i<TCP_CONN_NR; i++,
        tcp_conn++)
    {
        if (!(tcp_conn->tc_flags & TCF_INUSE))
            continue;
        if (tcp_conn->tc_busy)
            continue;
        if (tcp_conn->tc_fd)
            continue;
        if (tcp_conn->tc_state == TCS_CLOSED)
            continue;
        if (tcp_conn->tc_rcvd_data == NULL &&
            tcp_conn->tc_send_data == NULL)
        {
            continue;
        }
        tcp_close_connection (tcp_conn, EOUTOFBUFS);
    }
}

if (priority == TCP_PRI_CONN_INUSE)
{
    for (i=0, tcp_conn= tcp_conn_table; i<TCP_CONN_NR; i++,
        tcp_conn++)
    {
        if (!(tcp_conn->tc_flags & TCF_INUSE))
            continue;
        if (tcp_conn->tc_busy)
            continue;
        if (tcp_conn->tc_state == TCS_CLOSED)
            continue;
        if (tcp_conn->tc_rcvd_data == NULL &&
            tcp_conn->tc_send_data == NULL)
        {
            continue;
        }
        tcp_close_connection (tcp_conn, EOUTOFBUFS);
    }
}

}

#ifdef BUF_CONSISTENCY_CHECK
PRIVATE void tcp_bufcheck()
{
    int i;

```

```

tcp_conn_t *tcp_conn;
tcp_port_t *tcp_port;

for (i= 0, tcp_port= tcp_port_table; i<tcp_conf_nr; i++, tcp_port++)
{
    if (tcp_port->tp_pack)
        bf_check_acc(tcp_port->tp_pack);
}
for (i= 0, tcp_conn= tcp_conn_table; i<TCP_CONN_NR; i++, tcp_conn++)
{
    assert(!tcp_conn->tc_busy);
    if (tcp_conn->tc_rcvd_data)
        bf_check_acc(tcp_conn->tc_rcvd_data);
    if (tcp_conn->tc_adv_data)
        bf_check_acc(tcp_conn->tc_adv_data);
    if (tcp_conn->tc_send_data)
        bf_check_acc(tcp_conn->tc_send_data);
    if (tcp_conn->tc_remipopt)
        bf_check_acc(tcp_conn->tc_remipopt);
    if (tcp_conn->tc_tcpopt)
        bf_check_acc(tcp_conn->tc_tcpopt);
    if (tcp_conn->tc_frag2send)
        bf_check_acc(tcp_conn->tc_frag2send);
}
}
#endif

```

```

PUBLIC void tcp_notreach(tcp_conn)
tcp_conn_t *tcp_conn;
{
    int new_ttl;

    new_ttl= tcp_conn->tc_ttl;
    if (new_ttl == IP_MAX_TTL)
    {
        if (tcp_conn->tc_state == TCS_SYN_SENT)
            tcp_close_connection(tcp_conn, EDSTNOTRCH);
        return;
    }
    else if (new_ttl < TCP_DEF_TTL_NEXT)
        new_ttl= TCP_DEF_TTL_NEXT;
    else
    {
        new_ttl *= 2;
        if (new_ttl > IP_MAX_TTL)
            new_ttl= IP_MAX_TTL;
    }
    tcp_conn->tc_ttl= new_ttl;
    tcp_conn->tc_stt= 0;
    tcp_conn->tc_SND_TRM= tcp_conn->tc_SND_UNA;
    tcp_conn_write(tcp_conn, 1);
}

```

```

FORWARD u32_t mtu_table[]=
{
    /* From RFC-1191 */
    /* Plateau    MTU    Comments                Reference    */
    /* -----    ---    -----                - */
    /*          65535  Official maximum MTU        RFC 791      */
    /*          65535  Hyperchannel                RFC 1044     */
    /*          65535,
    /*          32000,    /* Just in case                */
    /*          17914    16Mb IBM Token Ring            ref. [6]     */
    /*          17914,
    /*          8166     IEEE 802.4                    RFC 1042     */
    /*          8166,
    /*          4464     IEEE 802.5 (4Mb max)           RFC 1042     */
    /*          4352     FDDI (Revised)                RFC 1188     */
    /*          4352, /* (1%) */
    /*          2048     Wideband Network              RFC 907      */
    /*          2002     IEEE 802.5 (4Mb recommended)  RFC 1042     */
    /*          2002, /* (2%) */
    /*          1536     Exp. Ethernet Nets            RFC 895      */
    /*          1500     Ethernet Networks            RFC 894      */
    /*          1500     Point-to-Point (default)      RFC 1134     */
}

```

```

/*      1492      IEEE 802.3      RFC 1042      */
      1492, /* (3%) */
/*      1006      SLIP      RFC 1055      */
/*      1006      ARPANET      BBN 1822      */
      1006,
/*      576      X.25 Networks      RFC 877      */
/*      544      DEC IP Portal      ref. [10]      */
/*      512      NETBIOS      RFC 1088      */
/*      508      IEEE 802/Source-Rt Bridge      RFC 1042      */
/*      508      ARCNET      RFC 1051      */
      508, /* (13%) */
/*      296      Point-to-Point (low delay)      RFC 1144      */
      296,
      68,      /* Official minimum MTU      RFC 791      */
      0,      /* End of list      */
};

```

```

PUBLIC void tcp_mtu_exceeded(tcp_conn)
tcp_conn_t *tcp_conn;
{
    u16_t mtu;
    int i;
    clock_t curr_time;

    if (!(tcp_conn->tc_flags & TCF_PMTU))
    {
        /* Strange, got MTU exceeded but DF is not set. Ignore
         * the error. If the problem persists, the connection will
         * time-out.
         */
        return;
    }
    curr_time= get_time();

    /* We get here in cases. Either were are trying to find an MTU
     * that works at all, or we are trying see how far we can increase
     * the current MTU. If the last change to the MTU was a long time
     * ago, we assume the second case.
     */
    if (curr_time >= tcp_conn->tc_mtutim + TCP_PMTU_INCR_IV)
    {
        mtu= tcp_conn->tc_mtu;
        mtu -= mtu/TCP_PMTU_INCR_FRAC;
        tcp_conn->tc_mtu= mtu;
        tcp_conn->tc_mtutim= curr_time;
        DBLOCK(1, printf(
            "tcp_mtu_exceeded: new (lowered) mtu %d for conn %d\n",
            mtu, tcp_conn-tcp_conn_table));
        tcp_conn->tc_stt= 0;
        tcp_conn->tc_SND_TRM= tcp_conn->tc_SND_UNA;
        tcp_conn_write(tcp_conn, 1);
        return;
    }

    tcp_conn->tc_mtutim= curr_time;
    mtu= tcp_conn->tc_mtu;
    for (i= 0; mtu_table[i] >= mtu; i++)
        ; /* Nothing to do */
    mtu= mtu_table[i];
    if (mtu >= TCP_MIN_PATH_MTU)
    {
        tcp_conn->tc_mtu= mtu;
    }
    else
    {
        /* Small MTUs can be used for denial-of-service attacks.
         * Switch-off PMTU if the MTU becomes too small.
         */
        tcp_conn->tc_flags &= ~TCF_PMTU;
        tcp_conn->tc_mtu= TCP_MIN_PATH_MTU;
        DBLOCK(1, printf(
            "tcp_mtu_exceeded: clearing TCF_PMTU for conn %d\n",
            tcp_conn-tcp_conn_table));
    }
}

```



```

    }
    DBLOCK(1, printf("tcp_mtu_exceeded: new mtu %d for conn %d\n",
        mtu, tcp_conn-tcp_conn_table));
    tcp_conn->tc_stt= 0;
    tcp_conn->tc_SND_TRM= tcp_conn->tc_SND_UNA;
    tcp_conn_write(tcp_conn, 1);
}

PUBLIC void tcp_mtu_incr(tcp_conn)
tcp_conn_t *tcp_conn;
{
    clock_t curr_time;
    u32_t mtu;

    assert(tcp_conn->tc_mtu < tcp_conn->tc_max_mtu);
    if (!(tcp_conn->tc_flags & TCF_PMTU))
    {
        /* Use a much longer time-out for retrying PMTU discovery
         * after is has been disabled. Note that PMTU discovery
         * can be disabled during a short loss of connectivity.
         */
        curr_time= get_time();
        if (curr_time > tcp_conn->tc_mtutim+TCP_PMTU_EN_IV)
        {
            tcp_conn->tc_flags |= TCF_PMTU;
            DBLOCK(1, printf(
                "tcp_mtu_incr: setting TCF_PMTU for conn %d\n",
                tcp_conn-tcp_conn_table));
        }
        return;
    }

    mtu= tcp_conn->tc_mtu;
    mtu += mtu/TCP_PMTU_INCR_FRAC;
    if (mtu > tcp_conn->tc_max_mtu)
        mtu= tcp_conn->tc_max_mtu;
    tcp_conn->tc_mtu= mtu;
    DBLOCK(0x1, printf("tcp_mtu_incr: new mtu %ld for conn %d\n",
        mtu, tcp_conn-tcp_conn_table));
}

/*
tcp_setup_conn
*/

PRIVATE void tcp_setup_conn(tcp_port, tcp_conn)
tcp_port_t *tcp_port;
tcp_conn_t *tcp_conn;
{
    ul6_t mss;

    assert(!tcp_conn->tc_connInprogress);
    tcp_conn->tc_port= tcp_port;
    if (tcp_conn->tc_flags & TCF_INUSE)
    {
        assert (tcp_conn->tc_state == TCS_CLOSED);
        assert (!tcp_conn->tc_send_data);
        if (tcp_conn->tc_senddis < get_time())
            tcp_conn->tc_ISS= 0;
    }
    else
    {
        assert(!tcp_conn->tc_busy);
        tcp_conn->tc_senddis= 0;
        tcp_conn->tc_ISS= 0;
        tcp_conn->tc_tos= TCP_DEF_TOS;
        tcp_conn->tc_ttl= TCP_DEF_TTL;
        tcp_conn->tc_rcv_wnd= TCP_MAX_RCV_WND_SIZE;
        tcp_conn->tc_fd= NULL;
    }
    if (!tcp_conn->tc_ISS)
    {
        tcp_conn->tc_ISS= tcp_rand32();
    }
}

```

```
tcp_conn->tc_SND_UNA= tcp_conn->tc_ISS;
tcp_conn->tc_SND_TRM= tcp_conn->tc_ISS;
tcp_conn->tc_SND_NXT= tcp_conn->tc_ISS+1;
tcp_conn->tc_SND_UP= tcp_conn->tc_ISS;
tcp_conn->tc_SND_PSH= tcp_conn->tc_ISS+1;
tcp_conn->tc_IRS= 0;
tcp_conn->tc_RCV_LO= tcp_conn->tc_IRS;
tcp_conn->tc_RCV_NXT= tcp_conn->tc_IRS;
tcp_conn->tc_RCV_HI= tcp_conn->tc_IRS;
tcp_conn->tc_RCV_UP= tcp_conn->tc_IRS;

assert(tcp_conn->tc_rcvd_data == NULL);
assert(tcp_conn->tc_adv_data == NULL);
assert(tcp_conn->tc_send_data == NULL);

tcp_conn->tc_ka_time= TCP_DEF_KEEPA_LIVE;

tcp_conn->tc_remipopt= NULL;
tcp_conn->tc_tcpopt= NULL;

assert(tcp_conn->tc_frag2send == NULL);

tcp_conn->tc_stt= 0;
tcp_conn->tc_rt_dead= TCP_DEF_RT_DEAD;
tcp_conn->tc_0wnd_to= 0;
tcp_conn->tc_artt= TCP_DEF_RTT*TCP_RTT_SCALE;
tcp_conn->tc_drtd= 0;
tcp_conn->tc_rtt= TCP_DEF_RTT;
tcp_conn->tc_max_mtu= tcp_conn->tc_port->tp_mtu;
tcp_conn->tc_mtu= tcp_conn->tc_max_mtu;
tcp_conn->tc_mtutim= 0;
tcp_conn->tc_error= NW_OK;
mss= tcp_conn->tc_mtu-IP_TCP_MIN_HDR_SIZE;
tcp_conn->tc_snd_cwnd= tcp_conn->tc_SND_UNA + 2*mss;
tcp_conn->tc_snd_cthresh= TCP_MAX_SND_WND_SIZE;
tcp_conn->tc_snd_cinc=
    (long)TCP_DEF_MSS*TCP_DEF_MSS/TCP_MAX_SND_WND_SIZE+1;
tcp_conn->tc_snd_wnd= TCP_MAX_SND_WND_SIZE;
tcp_conn->tc_rt_time= 0;
tcp_conn->tc_rt_seq= 0;
tcp_conn->tc_rt_threshold= tcp_conn->tc_ISS;
tcp_conn->tc_flags= TCF_INUSE;
tcp_conn->tc_flags |= TCF_PMTU;

clk_untimer(&tcp_conn->tc_transmit_timer);
tcp_conn->tc_transmit_seq= 0;
}

PRIVATE u32_t tcp_rand32()
{
    u8_t bits[RAND256_BUFSIZE];

    rand256(bits);
    return bits[0] | (bits[1] << 8) | (bits[2] << 16) | (bits[3] << 24);
}

/*
 * $PchId: tcp.c,v 1.34 2005/06/28 14:20:27 philip Exp $
 */
```

```

/*
tcp.h

Copyright 1995 Philip Homburg
*/

#ifndef TCP_H
#define TCP_H

#define TCP_MAX_DATAGRAM      8192

#ifndef TCP_MAX_SND_WND_SIZE
#define TCP_MAX_SND_WND_SIZE  (32*1024)
#endif

#ifndef TCP_MIN_RCV_WND_SIZE
#define TCP_MIN_RCV_WND_SIZE  (4*1024)
#endif

#ifndef TCP_MAX_RCV_WND_SIZE
#define TCP_MAX_RCV_WND_SIZE  (TCP_MIN_RCV_WND_SIZE + 28*1024)
#endif

#define TCP_DEF_TOS           0
#define TCP_DEF_TTL           5      /* hops/seconds */
#define TCP_DEF_TTL_NEXT      30     /* hops/seconds */

/* An established TCP connection times out if no communication is possible
 * for TCP_DEF_RT_DEAD clock ticks
 */
#ifndef TCP_DEF_RT_DEAD
#define TCP_DEF_RT_DEAD      (20L*60*HZ)
#endif

#define TCP_DEF_RT_MAX_CONNECT (5L*60*HZ) /* 5 minutes in ticks */
#define TCP_DEF_RT_MAX_LISTEN  (1L*60*HZ) /* 1 minute in ticks */
#define TCP_DEF_RT_MAX_CLOSING (1L*60*HZ) /* 1 minute in ticks */

/* Minimum and maximum intervals for zero window probes. */
#define TCP_0WND_MIN           (HZ)
#define TCP_0WND_MAX           (5*60*HZ)

#define TCP_DEF_RTT            15      /* initial retransmission time in
 * ticks
 */
#define TCP_RTT_GRAN           5      /* minimal value of the rtt is
 * TCP_RTT_GRAN * CLOCK_GRAN
 */
#define TCP_RTT_MAX            (10*HZ) /* The maximum retransmission interval
 * is TCP_RTT_MAX ticks
 */
#define TCP_RTT_SMOOTH         16     /* weight is 15/16 */
#define TCP_DRTT_MULT          4      /* weight of the deviation */
#define TCP_RTT_SCALE          256    /* Scaled values for more accuracy */

#ifndef TCP_DEF_KEEPALIVE
#define TCP_DEF_KEEPALIVE      (20L*60*HZ) /* Keepalive interval */
#endif

#ifndef TCP_DEF_MSS
#define TCP_DEF_MSS            1400
#endif

#define TCP_MIN_PATH_MTU       500
#define TCP_PMTU_INCR_IV       (1L*60*HZ) /* 1 minute in ticks */
#define TCP_PMTU_EN_IV         (10L*60*HZ) /* 10 minutes in ticks */
#define TCP_PMTU_INCR_FRAC     100     /* Add 1% each time */
#define TCP_PMTU_BLACKHOLE     (10*HZ) /* Assume a PMTU blackhole
 * after 10 seconds.
 */

#define TCP_DEF_CONF            (NWTC_COPY | NWTC_LP_UNSET | NWTC_UNSET_RA | \
                                NWTC_UNSET_RP)
#define TCP_DEF_OPT            (NWTO_NOFLAG)

```

```
#define TCP_DACK_RETRANS      3      /* # dup ACKs to start fast retrans. */

struct acc;

void tcp_prep ARGS(( void ));
void tcp_init ARGS(( void ));
int tcp_open ARGS(( int port, int srfd,
    get_userdata_t get_userdata, put_userdata_t put_userdata,
    put_pkt_t put_pkt, select_res_t select_res ));
int tcp_read ARGS(( int fd, size_t count));
int tcp_write ARGS(( int fd, size_t count));
int tcp_ioctl ARGS(( int fd, ioreq_t req));
int tcp_cancel ARGS(( int fd, int which_operation ));
void tcp_close ARGS(( int fd));

#endif /* TCP_H */

/*
 * $PchId: tcp.h,v 1.17 2005/06/28 14:20:54 philip Exp $
 */
```

```

/*
tcp_int.h

Copyright 1995 Philip Homburg
*/

#ifndef TCP_INT_H
#define TCP_INT_H

#define IP_TCP_MIN_HDR_SIZE      (IP_MIN_HDR_SIZE+TCP_MIN_HDR_SIZE)

#define TCP_CONN_HASH_SHIFT      4
#define TCP_CONN_HASH_NR        (1 << TCP_CONN_HASH_SHIFT)

typedef struct tcp_port
{
    int tp_ipdev;
    int tp_flags;
    int tp_state;
    int tp_ipfd;
    acc_t *tp_pack;
    ipaddr_t tp_ipaddr;
    ipaddr_t tp_subnetmask;
    ul6_t tp_mtu;
    struct tcp_conn *tp_snd_head;
    struct tcp_conn *tp_snd_tail;
    event_t tp_snd_event;
    struct tcp_conn *tp_conn_hash[TCP_CONN_HASH_NR][4];
} tcp_port_t;

#define TPF_EMPTY      0x0
#define TPF_SUSPEND    0x1
#define TPF_READ_IP    0x2
#define TPF_READ_SP    0x4
#define TPF_WRITE_IP   0x8
#define TPF_WRITE_SP   0x10
#define TPF_DELAY_TCP  0x40

#define TPS_EMPTY      0
#define TPS_SETPROTO   1
#define TPS_GETCONF    2
#define TPS_MAIN       3
#define TPS_ERROR      4

#define TFL_LISTEN_MAX 5

typedef struct tcp_fd
{
    unsigned long tf_flags;
    tcp_port_t *tf_port;
    int tf_srfd;
    ioreq_t tf_ioreq;
    nwio_tcpconf_t tf_tcpconf;
    nwio_tcptopt_t tf_tcptopt;
    get_userdata_t tf_get_userdata;
    put_userdata_t tf_put_userdata;
    select_res_t tf_select_res;
    struct tcp_conn *tf_conn;
    struct tcp_conn *tf_listenq[TFL_LISTEN_MAX];
    size_t tf_write_offset;
    size_t tf_write_count;
    size_t tf_read_offset;
    size_t tf_read_count;
    int tf_error;
    tcp_cookie_t tf_cookie;
} tcp_fd_t;

#define TFF_EMPTY      0x0
#define TFF_INUSE      0x1
#define TFF_READ_IP    0x2
#define TFF_WRITE_IP   0x4
#define TFF_IOCTL_IP   0x8
#define TFF_CONF_SET    0x10
#define TFF_IOC_INIT_SP 0x20

```

```

#define TFF_LISTENQ      0x40
#define TFF_CONNECTING  0x80
#define TFF_CONNECTED   0x100
#define TFF_WR_URG       0x200
#define TFF_PUSH_DATA    0x400
#define TFF_RECV_URG     0x800
#define TFF_SEL_READ     0x1000
#define TFF_SEL_WRITE    0x2000
#define TFF_SEL_EXCEPT 0x4000
#define TFF_DEL_RST      0x8000
#define TFF_COOKIE       0x10000

typedef struct tcp_conn
{
    int tc_flags;
    int tc_state;
    int tc_busy;           /* do not steal buffer when a connection is
                           * busy
                           */

    tcp_port_t *tc_port;
    tcp_fd_t *tc_fd;

    tcpport_t tc_locport;
    ipaddr_t tc_locaddr;
    tcpport_t tc_rempport;
    ipaddr_t tc_remaddr;

    int tc_connInprogress;
    int tc_orglisten;
    clock_t tc_senddis;

    /* Sending side */
    u32_t tc_ISS;          /* initial sequence number */
    u32_t tc_SND_UNA;       /* least unacknowledged sequence number */
    u32_t tc_SND_TRM;       /* next sequence number to be transmitted */
    u32_t tc_SND_NXT;       /* next sequence number for new data */
    u32_t tc_SND_UP;        /* urgent pointer, first sequence number not
                           * urgent */
    u32_t tc_SND_PSH;       /* push pointer, data should be pushed until
                           * the push pointer is reached */

    u32_t tc_snd_cwnd;      /* highest sequence number to be sent */
    u32_t tc_snd_athresh;   /* threshold for send window */
    u32_t tc_snd_cinc;      /* increment for send window threshold */
    u16_t tc_snd_wnd;       /* max send queue size */
    u16_t tc_snd_dack;      /* # of duplicate ACKs */

    /* round trip calculation. */
    clock_t tc_rt_time;
    u32_t tc_rt_seq;
    u32_t tc_rt_threshold;
    clock_t tc_artt;        /* Avg. retransmission time. Scaled. */
    clock_t tc_drtrt;       /* Deviation, also scaled. */
    clock_t tc_rtt;         /* Computed retrans time */

    acc_t *tc_send_data;
    acc_t *tc_frag2send;
    struct tcp_conn *tc_send_link;

    /* Receiving side */
    u32_t tc_IRS;
    u32_t tc_RCV_LO;
    u32_t tc_RCV_NXT;
    u32_t tc_RCV_HI;
    u32_t tc_RCV_UP;

    u16_t tc_rcv_wnd;
    acc_t *tc_rcvd_data;
    acc_t *tc_adv_data;
    u32_t tc_adv_seq;

    /* Keep alive. Record SDN_NXT and RCV_NXT in tc_ka_snd and
     * tc_ka_rcv when setting the keepalive timer to detect
     * any activity that may have happend before the timer
    */

```

```

    * expired.
    */
    u32_t tc_ka_snd;
    u32_t tc_ka_rcv;
    clock_t tc_ka_time;

    acc_t *tc_remipopt;
    acc_t *tc_tcpopt;
    u8_t tc_tos;
    u8_t tc_ttl;
    u16_t tc_max_mtu;          /* Max. negotiated (or selected) MTU */
    u16_t tc_mtu;              /* discovered PMTU */
    clock_t tc_mtutim;         /* Last time MTU/TCF_PMTU flag was changed */

    struct timer tc_transmit_timer;
    u32_t tc_transmit_seq;
    clock_t tc_0wnd_to;
    clock_t tc_stt;             /* time of first send after last ack */
    clock_t tc_rt_dead;

    int tc_error;
    int tc_inconsistent;
} tcp_conn_t;

#define TCF_EMPTY                0x0
#define TCF_INUSE                0x1
#define TCF_FIN_RECV             0x2
#define TCF_RCV_PUSH             0x4
#define TCF_MORE2WRITE           0x8
#define TCF_SEND_ACK             0x10
#define TCF_FIN_SENT             0x20
#define TCF_BSD_URG              0x40
#define TCF_NO_PUSH              0x80
#define TCF_PUSH_NOW             0x100
#define TCF_PMTU                 0x200

#if DEBUG & 0x200
#define TCF_DEBUG                0x1000
#endif

#define TCS_CLOSED               0
#define TCS_LISTEN               1
#define TCS_SYN_RECEIVED         2
#define TCS_SYN_SENT             3
#define TCS_ESTABLISHED         4
#define TCS_CLOSING              5

/* tcp_recv.c */
void tcp_frag2conn ARGS(( tcp_conn_t *tcp_conn, ip_hdr_t *ip_hdr,
    tcp_hdr_t *tcp_hdr, acc_t *tcp_data, size_t data_len ));
void tcp_fd_read ARGS(( tcp_conn_t *tcp_conn, int enq ));
unsigned tcp_sel_read ARGS(( tcp_conn_t *tcp_conn ));
void tcp_rsel_read ARGS(( tcp_conn_t *tcp_conn ));
void tcp_bytesavailable ARGS(( tcp_fd_t *tcp_fd, int *bytesp ));

/* tcp_send.c */
void tcp_conn_write ARGS(( tcp_conn_t *tcp_conn, int enq ));
void tcp_release_retrans ARGS(( tcp_conn_t *tcp_conn, u32_t seg_ack,
    U16_t new_win ));
void tcp_fast_retrans ARGS(( tcp_conn_t *tcp_conn ));
void tcp_set_send_timer ARGS(( tcp_conn_t *tcp_conn ));
void tcp_fd_write ARGS(( tcp_conn_t *tcp_conn ));
unsigned tcp_sel_write ARGS(( tcp_conn_t *tcp_conn ));
void tcp_rsel_write ARGS(( tcp_conn_t *tcp_conn ));
void tcp_close_connection ARGS(( tcp_conn_t *tcp_conn,
    int error ));
void tcp_port_write ARGS(( tcp_port_t *tcp_port ));
void tcp_shutdown ARGS(( tcp_conn_t *tcp_conn ));

/* tcp_lib.c */
void tcp_extract_ipopt ARGS(( tcp_conn_t *tcp_conn,
    ip_hdr_t *ip_hdr ));
void tcp_extract_tcpopt ARGS(( tcp_conn_t *tcp_conn,
    tcp_hdr_t *tcp_hdr, size_t *mssp ));

```

```

void tcp_get_ipopt ARGS(( tcp_conn_t *tcp_conn, ip_hdr_t
    *ip_hdr_opt ));
void tcp_get_tcptopt ARGS(( tcp_conn_t *tcp_conn, tcp_hdr_t
    *tcp_hdr_opt ));
acc_t *tcp_make_header ARGS(( tcp_conn_t *tcp_conn,
    ip_hdr_t **ref_ip_hdr, tcp_hdr_t **ref_tcp_hdr, acc_t *data ));
u16_t tcp_pack_oneCsum ARGS(( ip_hdr_t *ip_hdr, acc_t *tcp_pack ));
int tcp_check_conn ARGS(( tcp_conn_t *tcp_conn ));
void tcp_print_pack ARGS(( ip_hdr_t *ip_hdr, tcp_hdr_t *tcp_hdr ));
void tcp_print_state ARGS(( tcp_conn_t *tcp_conn ));
void tcp_print_conn ARGS(( tcp_conn_t *tcp_conn ));
int tcp_LEmod4G ARGS(( u32_t n1, u32_t n2 ));
int tcp_Lmod4G ARGS(( u32_t n1, u32_t n2 ));
int tcp_GEmod4G ARGS(( u32_t n1, u32_t n2 ));
int tcp_Gmod4G ARGS(( u32_t n1, u32_t n2 ));

/* tcp.c */
void tcp_restart_connect ARGS(( tcp_conn_t *tcp_conn ));
int tcp_su4listen ARGS(( tcp_fd_t *tcp_fd, tcp_conn_t *tcp_conn,
    int do_listenq ));
void tcp_reply_ioctl ARGS(( tcp_fd_t *tcp_fd, int reply ));
void tcp_reply_write ARGS(( tcp_fd_t *tcp_fd, size_t reply ));
void tcp_reply_read ARGS(( tcp_fd_t *tcp_fd, size_t reply ));
void tcp_notreach ARGS(( tcp_conn_t *tcp_conn ));
void tcp_mtu_exceeded ARGS(( tcp_conn_t *tcp_conn ));
void tcp_mtu_incr ARGS(( tcp_conn_t *tcp_conn ));

#define TCP_FD_NR          (10*IP_PORT_MAX)
#define TCP_CONN_NR       (2*TCP_FD_NR)

EXTERN tcp_port_t *tcp_port_table;
EXTERN tcp_conn_t tcp_conn_table[TCP_CONN_NR];
EXTERN tcp_fd_t tcp_fd_table[TCP_FD_NR];

#define tcp_Lmod4G(n1,n2)      (!(((n1)-(n2)) & 0x80000000L))
#define tcp_GEmod4G(n1,n2)    (!(((n1)-(n2)) & 0x80000000L))
#define tcp_Gmod4G(n1,n2)     (!(((n2)-(n1)) & 0x80000000L))
#define tcp_LEmod4G(n1,n2)    (!(((n2)-(n1)) & 0x80000000L))

#endif /* TCP_INT_H */

/*
 * $PchId: tcp_int.h,v 1.17 2005/06/28 14:21:08 philip Exp $
 */

```



```
/*
tcp_lib.c

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "buf.h"
#include "clock.h"
#include "event.h"
#include "io.h"
#include "type.h"

#include "assert.h"
#include "tcp_int.h"

THIS_FILE

#undef tcp_LEmod4G
PUBLIC int tcp_LEmod4G(n1, n2)
u32_t n1;
u32_t n2;
{
    return !((u32_t)(n2-n1) & 0x80000000L);
}

#undef tcp_GEmod4G
PUBLIC int tcp_GEmod4G(n1, n2)
u32_t n1;
u32_t n2;
{
    return !((u32_t)(n1-n2) & 0x80000000L);
}

#undef tcp_Lmod4G
PUBLIC int tcp_Lmod4G(n1, n2)
u32_t n1;
u32_t n2;
{
    return !((u32_t)(n1-n2) & 0x80000000L);
}

#undef tcp_Gmod4G
PUBLIC int tcp_Gmod4G(n1, n2)
u32_t n1;
u32_t n2;
{
    return !((u32_t)(n2-n1) & 0x80000000L);
}

PUBLIC void tcp_extract_ipopt(tcp_conn, ip_hdr)
tcp_conn_t *tcp_conn;
ip_hdr_t *ip_hdr;
{
    int ip_hdr_len;

    ip_hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
    if (ip_hdr_len == IP_MIN_HDR_SIZE)
        return;

    DBLOCK(1, printf("ip_hdr options NOT supported (yet?)\n"));
}

PUBLIC void tcp_extract_tcptopt(tcp_conn, tcp_hdr, mssp)
tcp_conn_t *tcp_conn;
tcp_hdr_t *tcp_hdr;
size_t *mssp;
{
    int i, tcp_hdr_len, type, len;
    u8_t *cp;
    ul6_t mss;

    *mssp= 0;          /* No mss */
}
```

```

tcp_hdr_len= (tcp_hdr->th_data_off & TH_DO_MASK) >> 2;
if (tcp_hdr_len == TCP_MIN_HDR_SIZE)
    return;
i= TCP_MIN_HDR_SIZE;
while (i<tcp_hdr_len)
{
    cp= ((u8_t *)tcp_hdr)+i;
    type= cp[0];
    if (type == TCP_OPT_NOP)
    {
        i++;
        continue;
    }
    if (type == TCP_OPT_EOL)
        break;
    if (i+2 > tcp_hdr_len)
        break; /* No length field */
    len= cp[1];
    if (i+len > tcp_hdr_len)
        break; /* Truncated option */
    i += len;
    switch(type)
    {
    case TCP_OPT_MSS:
        if (len != 4)
            break;
        mss= (cp[2] << 8) | cp[3];
        DBLOCK(1, printf("tcp_extract_tcpopt: got mss %d\n",
            mss));
        *mssp= mss;
        break;
    case TCP_OPT_WSOPT:      /* window scale option */
    case TCP_OPT_SACKOK:     /* SACK permitted */
    case TCP_OPT_TS:         /* Timestamps option */
    case TCP_OPT_CCNEW:      /* new connection count */
        /* Ignore this option. */
        break;
    default:
        DBLOCK(0x1,
            printf(
                "tcp_extract_tcpopt: unknown option %d, len %d\n",
                type, len));
        break;
    }
}
}

PUBLIC ul6_t tcp_pack_oneCsum(ip_hdr, tcp_pack)
ip_hdr_t *ip_hdr;
acc_t *tcp_pack;
{
    size_t ip_hdr_len;
    acc_t *pack;
    ul6_t sum;
    ul6_t word_buf[6];
    int odd_length;
    char *data_ptr;
    int length;

    ip_hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
    word_buf[0]= ip_hdr->ih_src & 0xffff;
    word_buf[1]= (ip_hdr->ih_src >> 16) & 0xffff;
    word_buf[2]= ip_hdr->ih_dst & 0xffff;
    word_buf[3]= (ip_hdr->ih_dst >> 16) & 0xffff;
    word_buf[4]= HTONS(IPPROTO_TCP);
    word_buf[5]= htons(ntohs(ip_hdr->ih_length)-ip_hdr_len);
    sum= oneC_sum(0, word_buf, sizeof(word_buf));

    pack= tcp_pack;
    odd_length= 0;
    for (; pack; pack= pack->acc_next)
    {
        data_ptr= ptr2acc_data(pack);

```

```

        length= pack->acc_length;

        if (!length)
            continue;
        sum= oneC_sum (sum, (u16_t *)data_ptr, length);
        if (length & 1)
        {
            odd_length= !odd_length;
            sum= ((sum >> 8) & 0xff) | ((sum & 0xff) << 8);
        }
    }
    if (odd_length)
    {
        /* Undo the last swap */
        sum= ((sum >> 8) & 0xff) | ((sum & 0xff) << 8);
    }
    return sum;
}

PUBLIC void tcp_get_ipopt(tcp_conn, ip_hdropt)
tcp_conn_t *tcp_conn;
ip_hdropt_t *ip_hdropt;
{
    if (!tcp_conn->tc_remipopt)
    {
        ip_hdropt->iho_opt_siz= 0;
        return;
    }
    DBLOCK(1, printf("ip_hdr options NOT supported (yet?)\n"));
    ip_hdropt->iho_opt_siz= 0;
    return;
}

PUBLIC void tcp_get_tcptopt(tcp_conn, tcp_hdropt)
tcp_conn_t *tcp_conn;
tcp_hdropt_t *tcp_hdropt;
{
    int optsiz;

    if (!tcp_conn->tc_tcptopt)
    {
        tcp_hdropt->tho_opt_siz= 0;
        return;
    }
    tcp_conn->tc_tcptopt= bf_pack(tcp_conn->tc_tcptopt);
    optsiz= bf_bufsize(tcp_conn->tc_tcptopt);
    memcpy(tcp_hdropt->tho_data, ptr2acc_data(tcp_conn->tc_tcptopt),
           optsiz);
    if ((optsiz & 3) != 0)
    {
        tcp_hdropt->tho_data[optsiz]= TCP_OPT_EOL;
        optsiz= (optsiz+3) & ~3;
    }
    tcp_hdropt->tho_opt_siz= optsiz;

    return;
}

PUBLIC acc_t *tcp_make_header(tcp_conn, ref_ip_hdr, ref_tcp_hdr, data)
tcp_conn_t *tcp_conn;
ip_hdr_t **ref_ip_hdr;
tcp_hdr_t **ref_tcp_hdr;
acc_t *data;
{
    ip_hdropt_t ip_hdropt;
    tcp_hdropt_t tcp_hdropt;
    ip_hdr_t *ip_hdr;
    tcp_hdr_t *tcp_hdr;
    acc_t *hdr_acc;
    char *ptr2hdr;
    int closed_connection;

    closed_connection= (tcp_conn->tc_state == TCS_CLOSED);

```

```

if (tcp_conn->tc_remipopt || tcp_conn->tc_tcpopt)
{
    tcp_get_ipopt (tcp_conn, &ip_hdropt);
    tcp_get_tcpopt (tcp_conn, &tcp_hdropt);
    assert (!(ip_hdropt.iho_opt_siz & 3));
    assert (!(tcp_hdropt.tho_opt_siz & 3));

    hdr_acc= bf_memreq(IP_MIN_HDR_SIZE+
                      ip_hdropt.iho_opt_siz+TCP_MIN_HDR_SIZE+
                      tcp_hdropt.tho_opt_siz);
    ptr2hdr= ptr2acc_data(hdr_acc);

    ip_hdr= (ip_hdr_t *)ptr2hdr;
    ptr2hdr += IP_MIN_HDR_SIZE;

    if (ip_hdropt.iho_opt_siz)
    {
        memcpy(ptr2hdr, (char *)ip_hdropt.iho_data,
               ip_hdropt.iho_opt_siz);
    }
    ptr2hdr += ip_hdropt.iho_opt_siz;

    tcp_hdr= (tcp_hdr_t *)ptr2hdr;
    ptr2hdr += TCP_MIN_HDR_SIZE;

    if (tcp_hdropt.tho_opt_siz)
    {
        memcpy (ptr2hdr, (char *)tcp_hdropt.tho_data,
                tcp_hdropt.tho_opt_siz);
    }
    hdr_acc->acc_next= data;

    ip_hdr->ih_vers_ihl= (IP_MIN_HDR_SIZE+
                        ip_hdropt.iho_opt_siz) >> 2;
    tcp_hdr->th_data_off= (TCP_MIN_HDR_SIZE+
                        tcp_hdropt.tho_opt_siz) << 2;
}
else
{
    hdr_acc= bf_memreq(IP_MIN_HDR_SIZE+TCP_MIN_HDR_SIZE);
    ip_hdr= (ip_hdr_t *)ptr2acc_data(hdr_acc);
    tcp_hdr= (tcp_hdr_t *)&ip_hdr[1];
    hdr_acc->acc_next= data;

    ip_hdr->ih_vers_ihl= IP_MIN_HDR_SIZE >> 2;
    tcp_hdr->th_data_off= TCP_MIN_HDR_SIZE << 2;
}

if (!closed_connection && (tcp_conn->tc_state == TCS_CLOSED))
{
    DBLOCK(1, printf("connection closed while inuse\n"));
    bf_afree(hdr_acc);
    return 0;
}

ip_hdr->ih_tos= tcp_conn->tc_tos;
ip_hdr->ih_ttl= tcp_conn->tc_ttl;
ip_hdr->ih_proto= IPPROTO_TCP;
ip_hdr->ih_src= tcp_conn->tc_locaddr;
ip_hdr->ih_dst= tcp_conn->tc_remaddr;
ip_hdr->ih_flags_fragoff= 0;
if (tcp_conn->tc_flags & TCF_PMTU)
    ip_hdr->ih_flags_fragoff |= HTONS(IH_DONT_FRAG);

tcp_hdr->th_srcport= tcp_conn->tc_locport;
tcp_hdr->th_dstport= tcp_conn->tc_rempport;
tcp_hdr->th_seq_nr= tcp_conn->tc_RCV_NXT;
tcp_hdr->th_flags= 0;
tcp_hdr->th_window= htons(tcp_conn->tc_RCV_HI-tcp_conn->tc_RCV_LO);
tcp_hdr->th_chksum= 0;
*ref_ip_hdr= ip_hdr;
*ref_tcp_hdr= tcp_hdr;
return hdr_acc;
}

```

```

PUBLIC void tcp_print_state (tcp_conn)
tcp_conn_t *tcp_conn;
{
#if DEBUG
    printf("tcp_conn_table[%d]->tc_state= ", tcp_conn-
        tcp_conn_table);
    if (!(tcp_conn->tc_flags & TCF_INUSE))
    {
        printf("not inuse\n");
        return;
    }
    switch (tcp_conn->tc_state)
    {
    case TCS_CLOSED: printf("CLOSED"); break;
    case TCS_LISTEN: printf("LISTEN"); break;
    case TCS_SYN_RECEIVED: printf("SYN_RECEIVED"); break;
    case TCS_SYN_SENT: printf("SYN_SENT"); break;
    case TCS_ESTABLISHED: printf("ESTABLISHED"); break;
    case TCS_CLOSING: printf("CLOSING"); break;
    default: printf("unknown(=%d)", tcp_conn->tc_state); break;
    }
#endif
}

PUBLIC int tcp_check_conn(tcp_conn)
tcp_conn_t *tcp_conn;
{
    int allright;
    u32_t lo_queue, hi_queue;
    int size;

    allright= TRUE;
    if (tcp_conn->tc_inconsistent)
    {
        assert(tcp_conn->tc_inconsistent == 1);
        printf("tcp_check_conn: connection is inconsistent\n");
        return allright;
    }

    /* checking receive queue */
    lo_queue= tcp_conn->tc_RCV_LO;
    if (lo_queue == tcp_conn->tc_IRS)
        lo_queue++;
    if (lo_queue == tcp_conn->tc_RCV_NXT && (tcp_conn->tc_flags &
        TCF_FIN_RECV))
        lo_queue--;
    hi_queue= tcp_conn->tc_RCV_NXT;
    if (hi_queue == tcp_conn->tc_IRS)
        hi_queue++;
    if (tcp_conn->tc_flags & TCF_FIN_RECV)
        hi_queue--;

    size= hi_queue-lo_queue;
    if (size<0)
    {
        printf("rcv hi_queue-lo_queue < 0\n");
        printf("SND_NXT= 0x%lx, SND_UNA= 0x%lx\n",
            (unsigned long)tcp_conn->tc_SND_NXT,
            (unsigned long)tcp_conn->tc_SND_UNA);
        printf("lo_queue= 0x%lx, hi_queue= 0x%lx\n",
            (unsigned long)lo_queue,
            (unsigned long)hi_queue);
        printf("size= %d\n", size);
        allright= FALSE;
    }
    else if (!tcp_conn->tc_rcvd_data)
    {
        if (size)
        {
            printf("RCV_NXT-RCV_LO != 0\n");
            tcp_print_conn(tcp_conn);
            printf("lo_queue= %lu, hi_queue= %lu\n",
                lo_queue, hi_queue);
        }
    }
}

```

```

        allright= FALSE;
    }
}
else if (size != bf_bufsize(tcp_conn->tc_rcvd_data))
{
    printf("RCV_NXT-RCV_LO != sizeof tc_rcvd_data\n");
    tcp_print_conn(tcp_conn);
    printf(
        "lo_queue= %lu, hi_queue= %lu, sizeof tc_rcvd_data= %d\n",
        lo_queue, hi_queue, bf_bufsize(tcp_conn->tc_rcvd_data));
    allright= FALSE;
}
else if (size != 0 && (tcp_conn->tc_state == TCS_CLOSED ||
    tcp_conn->tc_state == TCS_LISTEN ||
    tcp_conn->tc_state == TCS_SYN_RECEIVED ||
    tcp_conn->tc_state == TCS_SYN_SENT))
{
    printf("received data but not connected\n");
    tcp_print_conn(tcp_conn);
    allright= FALSE;
}
if (tcp_Lmod4G(tcp_conn->tc_RCV_HI, tcp_conn->tc_RCV_NXT))
{
    printf("tc_RCV_HI (0x%lx) < tc_RCV_NXT (0x%lx)\n",
        (unsigned long)tcp_conn->tc_RCV_HI,
        (unsigned long)tcp_conn->tc_RCV_NXT);
    allright= FALSE;
}

/* checking send data */
lo_queue= tcp_conn->tc_SND_UNA;
if (lo_queue == tcp_conn->tc_ISS)
    lo_queue++;
if (lo_queue == tcp_conn->tc_SND_NXT &&
    (tcp_conn->tc_flags & TCF_FIN_SENT))
{
    lo_queue--;
}
hi_queue= tcp_conn->tc_SND_NXT;
if (hi_queue == tcp_conn->tc_ISS)
    hi_queue++;
if (tcp_conn->tc_flags & TCF_FIN_SENT)
    hi_queue--;

size= hi_queue-lo_queue;
if (size<0)
{
    printf("snd hi_queue-lo_queue < 0\n");
    printf("SND_ISS= 0x%lx, SND_UNA= 0x%lx, SND_NXT= 0x%lx\n",
        (unsigned long)tcp_conn->tc_ISS,
        (unsigned long)tcp_conn->tc_SND_UNA,
        (unsigned long)tcp_conn->tc_SND_NXT);
    printf("hi_queue= 0x%lx, lo_queue= 0x%lx, size= %d\n",
        (unsigned long)hi_queue, (unsigned long)lo_queue,
        size);
    allright= FALSE;
}
else if (!tcp_conn->tc_send_data)
{
    if (size)
    {
        printf("SND_NXT-SND_UNA != 0\n");
        printf("SND_NXT= 0x%lx, SND_UNA= 0x%lx\n",
            (unsigned long)tcp_conn->tc_SND_NXT,
            (unsigned long)tcp_conn->tc_SND_UNA);
        printf("lo_queue= 0x%lx, hi_queue= 0x%lx\n",
            (unsigned long)lo_queue,
            (unsigned long)hi_queue);
        allright= FALSE;
    }
}
else if (size != bf_bufsize(tcp_conn->tc_send_data))
{
    printf("SND_NXT-SND_UNA != sizeof tc_send_data\n");

```

```

        printf("SND_NXT= 0x%lx, SND_UNA= 0x%lx\n",
               (unsigned long)tcp_conn->tc_SND_NXT,
               (unsigned long)tcp_conn->tc_SND_UNA);
        printf("lo_queue= 0x%lx, lo_queue= 0x%lx\n",
               (unsigned long)lo_queue,
               (unsigned long)hi_queue);
        printf("bf_bufsize(data)= %d\n",
               bf_bufsize(tcp_conn->tc_send_data));

        allright= FALSE;
    }

    /* checking counters */
    if (!tcp_GEmod4G(tcp_conn->tc_SND_UNA, tcp_conn->tc_ISS))
    {
        printf("SND_UNA < ISS\n");
        allright= FALSE;
    }
    if (!tcp_GEmod4G(tcp_conn->tc_SND_NXT, tcp_conn->tc_SND_UNA))
    {
        printf("SND_NXT<SND_UNA\n");
        allright= FALSE;
    }
    if (!tcp_GEmod4G(tcp_conn->tc_SND_TRM, tcp_conn->tc_SND_UNA))
    {
        printf("SND_TRM<SND_UNA\n");
        allright= FALSE;
    }
    if (!tcp_GEmod4G(tcp_conn->tc_SND_NXT, tcp_conn->tc_SND_TRM))
    {
        printf("SND_NXT<SND_TRM\n");
        allright= FALSE;
    }

    DIFBLOCK(1, (!allright), printf("tcp_check_conn: not allright\n"));
    return allright;
}

PUBLIC void tcp_print_pack(ip_hdr, tcp_hdr)
ip_hdr_t *ip_hdr;
tcp_hdr_t *tcp_hdr;
{
    int tcp_hdr_len;

    assert(tcp_hdr);
    if (ip_hdr)
        writeIpAddr(ip_hdr->ih_src);
    else
        printf("???");
    printf(",%u ", ntohs(tcp_hdr->th_srcport));
    if (ip_hdr)
        writeIpAddr(ip_hdr->ih_dst);
    else
        printf("???");
    printf(",%u ", ntohs(tcp_hdr->th_dstport));
    printf(" 0x%lx", ntohl(tcp_hdr->th_seq_nr));
    if (tcp_hdr->th_flags & THF_FIN)
        printf(" <FIN>");
    if (tcp_hdr->th_flags & THF_SYN)
        printf(" <SYN>");
    if (tcp_hdr->th_flags & THF_RST)
        printf(" <RST>");
    if (tcp_hdr->th_flags & THF_PSH)
        printf(" <PSH>");
    if (tcp_hdr->th_flags & THF_ACK)
        printf(" <ACK 0x%lx %u>", ntohl(tcp_hdr->th_ack_nr),
               ntohs(tcp_hdr->th_window));
    if (tcp_hdr->th_flags & THF_URG)
        printf(" <URG %u>", tcp_hdr->th_urgptr);
    tcp_hdr_len= (tcp_hdr->th_data_off & TH_DO_MASK) >> 2;
    if (tcp_hdr_len != TCP_MIN_HDR_SIZE)
        printf(" <options %d>", tcp_hdr_len-TCP_MIN_HDR_SIZE);
}

```

```

PUBLIC void tcp_print_conn(tcp_conn)
tcp_conn_t *tcp_conn;
{
    u32_t iss, irs;
    tcp_fd_t *tcp_fd;

    iss= tcp_conn->tc_ISS;
    irs= tcp_conn->tc_IRS;

    tcp_print_state (tcp_conn);
    printf(
        " ISS 0x%lx UNA +0x%lx(0x%lx) TRM +0x%lx(0x%lx) NXT +0x%lx(0x%lx)",
        iss, tcp_conn->tc_SND_UNA-iss, tcp_conn->tc_SND_UNA,
        tcp_conn->tc_SND_TRM-iss, tcp_conn->tc_SND_TRM,
        tcp_conn->tc_SND_NXT-iss, tcp_conn->tc_SND_NXT);
    printf(
        " UP +0x%lx(0x%lx) PSH +0x%lx(0x%lx) ",
        tcp_conn->tc_SND_UP-iss, tcp_conn->tc_SND_UP,
        tcp_conn->tc_SND_PSH-iss, tcp_conn->tc_SND_PSH);
    printf(" snd_cwnd +0x%lx(0x%lx)",
        tcp_conn->tc_snd_cwnd-tcp_conn->tc_SND_UNA,
        tcp_conn->tc_snd_cwnd);
    printf(" transmit_seq ");
    if (tcp_conn->tc_transmit_seq == 0)
        printf("0");
    else
    {
        printf("+0x%lx(0x%lx)", tcp_conn->tc_transmit_seq-iss,
            tcp_conn->tc_transmit_seq);
    }
    printf(" IRS 0x%lx LO +0x%lx(0x%lx) NXT +0x%lx(0x%lx) HI +0x%lx(0x%lx)",
        irs, tcp_conn->tc_RCV_LO-irs, tcp_conn->tc_RCV_LO,
        tcp_conn->tc_RCV_NXT-irs, tcp_conn->tc_RCV_NXT,
        tcp_conn->tc_RCV_HI-irs, tcp_conn->tc_RCV_HI);
    if (tcp_conn->tc_flags & TCF_INUSE)
        printf(" TCF_INUSE");
    if (tcp_conn->tc_flags & TCF_FIN_RECV)
        printf(" TCF_FIN_RECV");
    if (tcp_conn->tc_flags & TCF_RCV_PUSH)
        printf(" TCF_RCV_PUSH");
    if (tcp_conn->tc_flags & TCF_MORE2WRITE)
        printf(" TCF_MORE2WRITE");
    if (tcp_conn->tc_flags & TCF_SEND_ACK)
        printf(" TCF_SEND_ACK");
    if (tcp_conn->tc_flags & TCF_FIN_SENT)
        printf(" TCF_FIN_SENT");
    if (tcp_conn->tc_flags & TCF_BSD_URG)
        printf(" TCF_BSD_URG");
    if (tcp_conn->tc_flags & TCF_NO_PUSH)
        printf(" TCF_NO_PUSH");
    if (tcp_conn->tc_flags & TCF_PUSH_NOW)
        printf(" TCF_PUSH_NOW");
    if (tcp_conn->tc_flags & TCF_PMTU)
        printf(" TCF_PMTU");
    printf("\n");
    writeIpAddr(tcp_conn->tc_locaddr);
    printf(",%u->", ntohs(tcp_conn->tc_locport));
    writeIpAddr(tcp_conn->tc_remaddr);
    printf(",%u\n", ntohs(tcp_conn->tc_rempport));
    tcp_fd= tcp_conn->tc_fd;
    if (!tcp_fd)
        printf("tc_fd NULL");
    else
    {
        printf("tc_fd #%d: flags 0x%x, r %u@%u, w %u@%u",
            tcp_fd-tcp_fd_table, tcp_fd->tf_flags,
            tcp_fd->tf_read_count, tcp_fd->tf_read_offset,
            tcp_fd->tf_write_count, tcp_fd->tf_write_offset);
    }
}

/*
 * $PchId: tcp_lib.c,v 1.14 2005/01/31 21:41:38 philip Exp $
 */

```



```

/*
tcp_recv.c

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "buf.h"
#include "clock.h"
#include "event.h"
#include "type.h"
#include "sr.h"

#include "io.h"
#include "tcp_int.h"
#include "tcp.h"
#include "assert.h"

THIS_FILE

FORWARD void create_RST ARGS(( tcp_conn_t *tcp_conn,
                             ip_hdr_t *ip_hdr, tcp_hdr_t *tcp_hdr, int data_len ));
FORWARD void process_data ARGS(( tcp_conn_t *tcp_conn,
                                tcp_hdr_t *tcp_hdr, acc_t *tcp_data, int data_len ));
FORWARD void process_advanced_data ARGS(( tcp_conn_t *tcp_conn,
                                          tcp_hdr_t *tcp_hdr, acc_t *tcp_data, int data_len ));

PUBLIC void tcp_frag2conn(tcp_conn, ip_hdr, tcp_hdr, tcp_data, data_len)
tcp_conn_t *tcp_conn;
ip_hdr_t *ip_hdr;
tcp_hdr_t *tcp_hdr;
acc_t *tcp_data;
size_t data_len;
{
    tcp_fd_t *connuser;
    int tcp_hdr_flags;
    int ip_hdr_len, tcp_hdr_len;
    u32_t seg_ack, seg_seq, rcv_hi, snd_una, snd_nxt;
    u16_t seg_wnd, mtu;
    size_t mss;
    int acceptable_ACK, segm_acceptable, send_rst, close_connection;

    ip_hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
    tcp_hdr_len= (tcp_hdr->th_data_off & TH_DO_MASK) >> 2;

    tcp_hdr_flags= tcp_hdr->th_flags & TH_FLAGS_MASK;
    seg_ack= ntohl(tcp_hdr->th_ack_nr);
    seg_seq= ntohl(tcp_hdr->th_seq_nr);
    seg_wnd= ntohs(tcp_hdr->th_window);

    #if 0
    { where(); tcp_print_conn(tcp_conn); printf("\n");
      tcp_print_pack(ip_hdr, tcp_hdr); printf("\n"); }
    #endif

    switch (tcp_conn->tc_state)
    {
    case TCS_CLOSED:
        /*
        CLOSED:
        discard all data.
        !RST ?
            ACK ?
                <SEQ=SEG.ACK><CTL=RST>
                exit
            :
                <SEQ=0><ACK=SEG.SEG+SEG.LEN><CTL=RST,ACK>
                exit
            :
                discard packet
                exit
        */

        if (!(tcp_hdr_flags & THF_RST))

```

```

        {
            create_RST(tcp_conn, ip_hdr, tcp_hdr, data_len);
            tcp_conn_write(tcp_conn, 1);
        }
        break;
case TCS_LISTEN:
/*
LISTEN:
    RST ?
        discard packet
        exit
    ACK ?
        <SEQ=SEG.ACK><CTL=RST>
        exit
    SYN ?
        BUG: no security check
        RCV.NXT= SEG.SEQ+1
        IRS= SEG.SEQ
        ISS should already be selected
        <SEQ=ISS><ACK=RCV.NXT><CTL=SYN,ACK>
        SND.NXT=ISS+1
        SND.UNA=ISS
        state= SYN-RECEIVED
        exit
:
        shouldn't occur
        discard packet
        exit
*/

    if (tcp_hdr_flags & THF_RST)
        break;
    if (tcp_hdr_flags & THF_ACK)
    {
        create_RST (tcp_conn, ip_hdr, tcp_hdr, data_len);
        tcp_conn_write(tcp_conn, 1);
        break;
    }
    if (tcp_hdr_flags & THF_SYN)
    {
        tcp_extract_ipopt(tcp_conn, ip_hdr);
        tcp_extract_tcptopt(tcp_conn, tcp_hdr, &mss);
        mtu= mss+IP_TCP_MIN_HDR_SIZE;
        if (mtu < IP_MIN_MTU)
        {
            /* No or unrealistic mss, use default MTU */
            mtu= IP_DEF_MTU;
        }
        if (mtu < tcp_conn->tc_max_mtu)
        {
            tcp_conn->tc_max_mtu= mtu;
            tcp_conn->tc_mtu= mtu;
            DBLOCK(1, printf(
                "tcp[%d]: conn[%d]: mtu = %d\n",
                tcp_conn->tc_port-tcp_port_table,
                tcp_conn-tcp_conn_table,
                mtu));
        }

        tcp_conn->tc_RCV_LO= seg_seq+1;
        tcp_conn->tc_RCV_NXT= seg_seq+1;
        tcp_conn->tc_RCV_HI= tcp_conn->tc_RCV_LO+
            tcp_conn->tc_rcv_wnd;
        tcp_conn->tc_RCV_UP= seg_seq;
        tcp_conn->tc_IRS= seg_seq;
        tcp_conn->tc_SND_UNA= tcp_conn->tc_ISS;
        tcp_conn->tc_SND_TRM= tcp_conn->tc_ISS;
        tcp_conn->tc_SND_NXT= tcp_conn->tc_ISS+1;
        tcp_conn->tc_SND_UP= tcp_conn->tc_ISS-1;
        tcp_conn->tc_SND_PSH= tcp_conn->tc_ISS-1;
        tcp_conn->tc_state= TCS_SYN_RECEIVED;
        tcp_conn->tc_stt= 0;
        assert (tcp_check_conn(tcp_conn));
        tcp_conn->tc_locaddr= ip_hdr->ih_dst;
        tcp_conn->tc_locport= tcp_hdr->th_dstport;
    }

```

```

        tcp_conn->tc_remaddr= ip_hdr->ih_src;
        tcp_conn->tc_rempport= tcp_hdr->th_srcport;
        tcp_conn_write(tcp_conn, 1);

        DIFBLOCK(0x10, seg_seq == 0,
            printf("warning got 0 IRS from ");
            writeIpAddr(tcp_conn->tc_remaddr);
            printf("\n"));

        /* Start the timer (if necessary) */
        tcp_set_send_timer(tcp_conn);

        break;
    }
    /* do nothing */
    break;
case TCS_SYN_SENT:
/*
SYN-SENT:
    ACK ?
        SEG.ACK <= ISS || SEG.ACK > SND.NXT ?
            RST ?
                discard packet
                exit
            :
                <SEQ=SEG.ACK><CTL=RST>
                exit
        SND.UNA <= SEG.ACK && SEG.ACK <= SND.NXT ?
            ACK is acceptable
        :
            ACK is !acceptable
    :
        ACK is !acceptable
    RST ?
        ACK acceptable ?
            discard segment
            state= CLOSED
            error "connection refused"
            exit
        :
            discard packet
            exit
    BUG: no security check
    SYN ?
        IRS= SEG.SEQ
        RCV.NXT= IRS+1
        ACK ?
            SND.UNA= SEG.ACK
        SND.UNA > ISS ?
            state= ESTABLISHED
            <SEQ=SND.NXT><ACK= RCV.NXT><CTL=ACK>
            process ev. URG and text
            exit
        :
            state= SYN-RECEIVED
            SND.WND= SEG.WND
            SND.WL1= SEG.SEQ
            SND.WL2= SEG.ACK
            <SEQ=ISS><ACK=RCV.NXT><CTL=SYN,ACK>
            exit
    :
        discard segment
        exit
*/
    if (tcp_hdr_flags & THF_ACK)
    {
        if (tcp_LEmod4G(seg_ack, tcp_conn->tc_ISS) ||
            tcp_Gmod4G(seg_ack, tcp_conn->tc_SND_NXT))
            if (tcp_hdr_flags & THF_RST)
                break;
            else
            {
                /* HACK: force sending a RST,
                 * normally, RSTs are not send

```

```

        * if the segment is an ACK.
        */
        create_RST (tcp_conn, ip_hdr,
                    tcp_hdr, data_len+1);
        tcp_conn_write(tcp_conn, 1);
        break;
    }
    acceptable_ACK= (tcp_LEmod4G(tcp_conn->tc_SND_UNA,
                                seg_ack) && tcp_LEmod4G(seg_ack,
                                tcp_conn->tc_SND_NXT));
}
else
    acceptable_ACK= FALSE;
if (tcp_hdr_flags & THF_RST)
{
    if (acceptable_ACK)
    {
        DBLOCK(1, printf(
            "calling tcp_close_connection\n"));

        tcp_close_connection(tcp_conn,
                            ECONNREFUSED);
    }
    break;
}
if (tcp_hdr_flags & THF_SYN)
{
    tcp_extract_ipopt(tcp_conn, ip_hdr);
    tcp_extract_tcptopt(tcp_conn, tcp_hdr, &mss);
    mtu= mss+IP_TCP_MIN_HDR_SIZE;
    if (mtu < IP_MIN_MTU)
    {
        /* No or unrealistic mss, use default MTU */
        mtu= IP_DEF_MTU;
    }
    if (mtu < tcp_conn->tc_max_mtu)
    {
        tcp_conn->tc_max_mtu= mtu;
        tcp_conn->tc_mtu= mtu;
        DBLOCK(1, printf(
            "tcp[%d]: conn[%d]: mtu = %d\n",
            tcp_conn->tc_port-tcp_port_table,
            tcp_conn-tcp_conn_table,
            mtu));
    }
    tcp_conn->tc_RCV_LO= seg_seq+1;
    tcp_conn->tc_RCV_NXT= seg_seq+1;
    tcp_conn->tc_RCV_HI= tcp_conn->tc_RCV_LO +
        tcp_conn->tc_rcv_wnd;
    tcp_conn->tc_RCV_UP= seg_seq;
    tcp_conn->tc_IRS= seg_seq;
    if (tcp_hdr_flags & THF_ACK)
        tcp_conn->tc_SND_UNA= seg_ack;
    if (tcp_Gmod4G(tcp_conn->tc_SND_UNA,
        tcp_conn->tc_ISS))
    {
        tcp_conn->tc_state= TCS_ESTABLISHED;
        tcp_conn->tc_rt_dead= TCP_DEF_RT_DEAD;

        assert (tcp_check_conn(tcp_conn));
        assert(tcp_conn->tc_connInProgress);

        tcp_restart_connect(tcp_conn);

        tcp_conn->tc_flags |= TCF_SEND_ACK;
        tcp_conn_write(tcp_conn, 1);
        if (data_len != 0)
        {
            tcp_frag2conn(tcp_conn, ip_hdr,
                        tcp_hdr, tcp_data, data_len);
            /* tcp_data is already freed */
            return;
        }
    }
    break;
}

```

```

    }
    tcp_conn->tc_state= TCS_SYN_RECEIVED;

    assert (tcp_check_conn(tcp_conn));

    tcp_conn->tc_SND_TRM= tcp_conn->tc_ISS;
    tcp_conn_write(tcp_conn, 1);
}
break;

case TCS_SYN_RECEIVED:
/*
SYN-RECEIVED:
    test if segment is acceptable:

Segment Receive Test
Length Window
0 0 SEG.SEQ == RCV.NXT
0 >0 RCV.NXT <= SEG.SEQ && SEG.SEQ < RCV.NXT+RCV.WND
>0 0 not acceptable
>0 >0 (RCV.NXT <= SEG.SEQ && SEG.SEQ < RCV.NXT+RCV.WND)
      || (RCV.NXT <= SEG.SEQ+SEG.LEN-1 &&
      SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND)
for urgent data: use RCV.WND+1 for RCV.WND

Special: Send RST if SEG.SEQ < IRS or SEG.SEQ > RCV.NXT+64K (and
the packet is not a RST packet itself).
*/

rcv_hi= tcp_conn->tc_RCV_HI;
if (tcp_hdr_flags & THF_URG)
    rcv_hi++;
send_rst= tcp_Lmod4G(seg_seq, tcp_conn->tc_IRS) ||
    tcp_Gmod4G(seg_seq, tcp_conn->tc_RCV_NXT+0x10000);
close_connection= 0;

if (!data_len)
{
    if (rcv_hi == tcp_conn->tc_RCV_NXT)
        segm_acceptable= (seg_seq == rcv_hi);
    else
    {
        assert (tcp_Gmod4G(rcv_hi,
            tcp_conn->tc_RCV_NXT));
        segm_acceptable= (tcp_LEmod4G(tcp_conn->
            tc_RCV_NXT, seg_seq) &&
            tcp_Lmod4G(seg_seq, rcv_hi));
    }
}
else
{
    if (tcp_Gmod4G(rcv_hi, tcp_conn->tc_RCV_NXT))
    {
        segm_acceptable= (tcp_LEmod4G(tcp_conn->
            tc_RCV_NXT, seg_seq) &&
            tcp_Lmod4G(seg_seq, rcv_hi)) ||
            (tcp_LEmod4G(tcp_conn->tc_RCV_NXT,
            seg_seq+data_len-1) &&
            tcp_Lmod4G(seg_seq+data_len-1,
            rcv_hi));
    }
    else
    {
        segm_acceptable= FALSE;
    }
}

/*
!segment acceptable ?
RST ?
    discard packet
    exit
:
    <SEG=SND.NXT><ACK=RCV.NXT><CTL=ACK>
    exit
*/

```

```

        if (!segm_acceptable)
        {
            if (tcp_hdr_flags & THF_RST)
                ; /* do nothing */
            else if (send_rst)
            {
                create_RST(tcp_conn, ip_hdr, tcp_hdr,
                           data_len);
                tcp_conn_write(tcp_conn, 1);
            }
            else
            {
                tcp_conn->tc_flags |= TCF_SEND_ACK;
                tcp_conn_write(tcp_conn, 1);
            }
            break;
        }
    }
/*
    RST ?
    initiated by a LISTEN ?
        state= LISTEN
        exit
    :
        state= CLOSED
        error "connection refused"
        exit
*/

    if (tcp_hdr_flags & THF_RST)
        close_connection= 1;
/*
    SYN in window ?
    initiated by a LISTEN ?
        state= LISTEN
        exit
    :
        state= CLOSED
        error "connection reset"
        exit
*/

    if ((tcp_hdr_flags & THF_SYN) && tcp_GEmod4G(seg_seq,
        tcp_conn->tc_RCV_NXT))
    {
        close_connection= 1;
    }

    if (close_connection)
    {
        if (!tcp_conn->tc_orglisten)
        {
            tcp_close_connection(tcp_conn, ECONNREFUSED);
            break;
        }

        connuser= tcp_conn->tc_fd;
        assert(connuser);
        if (connuser->tf_flags & TFF_LISTENQ)
        {
            tcp_close_connection (tcp_conn,
                                   ECONNREFUSED);
        }
        else
        {
            tcp_conn->tc_connInprogress= 0;
            tcp_conn->tc_fd= NULL;

            tcp_close_connection (tcp_conn,
                                   ECONNREFUSED);

            /* Pick a new ISS next time */
            tcp_conn->tc_ISS= 0;

            (void)tcp_su4listen(connuser, tcp_conn,

```

```

                                0 /* !do_listenq */);
                                }
                                break;
                                }
/*
    !ACK ?
    discard packet
    exit
*/
    if (!(tcp_hdr_flags & THF_ACK))
        break;
/*
    SND.UNA < SEG.ACK <= SND.NXT ?
    state= ESTABLISHED
:
    <SEG=SEG.ACK><CTL=RST>
    exit
*/
    if (tcp_Lmod4G(tcp_conn->tc_SND_UNA, seg_ack) &&
        tcp_LEmod4G(seg_ack, tcp_conn->tc_SND_NXT))
    {
        tcp_conn->tc_state= TCS_ESTABLISHED;
        tcp_conn->tc_rt_dead= TCP_DEF_RT_DEAD;

        tcp_release_retrans(tcp_conn, seg_ack, seg_wnd);

        assert (tcp_check_conn(tcp_conn));
        assert(tcp_conn->tc_connInprogress);

        tcp_restart_connect(tcp_conn);
        tcp_frag2conn(tcp_conn, ip_hdr, tcp_hdr, tcp_data,
                      data_len);
        /* tcp_data is already freed */
        return;
    }
    else
    {
        create_RST (tcp_conn, ip_hdr, tcp_hdr, data_len);
        tcp_conn_write(tcp_conn, 1);
        break;
    }
    break;

    case TCS_ESTABLISHED:
    case TCS_CLOSING:
/*
ESTABLISHED:
FIN-WAIT-1:
FIN-WAIT-2:
CLOSE-WAIT:
CLOSING:
LAST-ACK:
TIME-WAIT:
    test if segment is acceptable:
    Segment Receive Test
    Length Window
    0      0      SEG.SEQ == RCV.NXT
    0      >0     RCV.NXT <= SEG.SEQ && SEG.SEQ < RCV.NXT+RCV.WND
    >0     0      not acceptable
    >0     >0     (RCV.NXT <= SEG.SEQ && SEG.SEQ < RCV.NXT+RCV.WND)
                  || (RCV.NXT <= SEG.SEQ+SEG.LEN-1 &&
                      SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND)
    for urgent data: use RCV.WND+1 for RCV.WND
*/
    rcv_hi= tcp_conn->tc_RCV_HI;
    if (tcp_hdr_flags & THF_URG)
        rcv_hi++;
    if (!data_len)
    {
        if (rcv_hi == tcp_conn->tc_RCV_NXT)
            segm_acceptable= (seg_seq == rcv_hi);
        else
        {
            assert (tcp_Gmod4G(rcv_hi,

```

```

        tcp_conn->tc_RCV_NXT));
        segm_acceptable= (tcp_LEmod4G(tcp_conn->
        tc_RCV_NXT, seg_seq) &&
        tcp_Lmod4G(seg_seq, rcv_hi));
    }
}
else
{
    if (tcp_Gmod4G(rcv_hi, tcp_conn->tc_RCV_NXT))
    {
        segm_acceptable= (tcp_LEmod4G(tcp_conn->
        tc_RCV_NXT, seg_seq) &&
        tcp_Lmod4G(seg_seq, rcv_hi)) ||
        (tcp_LEmod4G(tcp_conn->tc_RCV_NXT,
        seg_seq+data_len-1) &&
        tcp_Lmod4G(seg_seq+data_len-1,
        rcv_hi));
    }
    else
    {
        segm_acceptable= FALSE;
    }
}
}
/*
!segment acceptable ?
RST ?
    discard packet
    exit
:
    <SEG=SND.NXT><ACK=RCV.NXT><CTL=ACK>
    exit
*/
if (!segm_acceptable)
{
    if (!(tcp_hdr_flags & THF_RST))
    {
        DBLOCK(0x20,
        printf("segment is not acceptable\n");
        printf("\t");
        tcp_print_pack(ip_hdr, tcp_hdr);
        printf("\n\t");
        tcp_print_conn(tcp_conn);
        printf("\n");
        tcp_conn->tc_flags |= TCF_SEND_ACK;
        tcp_conn_write(tcp_conn, 1);

        /* Sometimes, a retransmission sets the PSH
        * flag (Solaris 2.4)
        */
        if (tcp_conn->tc_rcvd_data != NULL &&
        (tcp_hdr_flags & THF_PSH))
        {
            tcp_conn->tc_flags |= TCF_RCV_PUSH;
            if (tcp_conn->tc_fd &&
            (tcp_conn->tc_fd->tf_flags &
            TFF_READ_IP))
            {
                tcp_fd_read(tcp_conn, 1);
            }
            if (tcp_conn->tc_fd &&
            (tcp_conn->tc_fd->tf_flags &
            TFF_SEL_READ))
            {
                tcp_rsel_read(tcp_conn);
            }
        }
    }
    break;
}
}
/*
RST ?
    state == CLOSING || state == LAST-ACK ||
    state == TIME-WAIT ?
    state= CLOSED

```



```

        exit
    :
        state= CLOSED
        error "connection reset"
        exit
*/

if (tcp_hdr_flags & THF_RST)
{
    if ((tcp_conn->tc_flags &
        (TCF_FIN_SENT|TCF_FIN_RECV)) ==
        (TCF_FIN_SENT|TCF_FIN_RECV) &&
        tcp_conn->tc_send_data == NULL)
    {
        /* Clean shutdown, but the other side
         * doesn't want to ACK our FIN.
         */
        tcp_close_connection (tcp_conn, 0);
    }
    else
        tcp_close_connection(tcp_conn, ECONNRESET);
    break;
}

/*
SYN in window ?
state= CLOSED
error "connection reset"
exit
*/

if ((tcp_hdr_flags & THF_SYN) && tcp_GEmod4G(seg_seq,
    tcp_conn->tc_RCV_NXT))
{
    tcp_close_connection(tcp_conn, ECONNRESET);
    break;
}

/*
!ACK ?
discard packet
exit
*/

if (!(tcp_hdr_flags & THF_ACK))
    break;

/*
SND.UNA < SEG.ACK <= SND.NXT ?
SND.UNA= SEG.ACK
reply "send ok"
SND.WL1 < SEG.SEQ || (SND.WL1 == SEG.SEQ &&
    SND.WL2 <= SEG.ACK ?
    SND.WND= SEG.WND
    SND.WL1= SEG.SEQ
    SND.WL2= SEG.ACK
SEG.ACK <= SND.UNA ?
ignore ACK
SEG.ACK > SND.NXT ?
<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>
discard packet
exit
*/

/* Always reset the send timer after a valid ack is
 * received. The assumption is that either the ack really
 * acknowledges some data (normal case), contains a zero
 * window, or the remote host has another reason not
 * to accept any data. In all cases, the remote host is
 * alive, so the connection should stay alive too.
 * Do not reset stt if the state is CLOSING, i.e. if
 * the user closed the connection and we still have
 * some data to deliver. We don't want a zero window
 * to keep us from closing the connection.
 */
if (tcp_conn->tc_state != TCS_CLOSING)
    tcp_conn->tc_stt= 0;

snd_una= tcp_conn->tc_SND_UNA;

```

```

snd_nxt= tcp_conn->tc_SND_NXT;
if (seg_ack == snd_una)
{

    if (tcp_Gmod4G(snd_nxt, snd_una))
    {
        /* Duplicate ACK */
        if (++tcp_conn->tc_snd_dack ==
            TCP_DACK_RETRANS)
        {
            tcp_fast_retrans(tcp_conn);
        }
    }

    /* This ACK doesn't acknowledge any new data, this
     * is a likely situation if we are only receiving
     * data. We only update the window if we are
     * actually sending or if we currently have a
     * zero window.
     */
    if (tcp_conn->tc_snd_cwnd == snd_una &&
        seg_wnd != 0)
    {
        DBLOCK(2, printf("zero window opened\n"));
        /* The other side opened up its receive
         * window. */
        mss= tcp_conn->tc_mtu-IP_TCP_MIN_HDR_SIZE;
        if (seg_wnd > 2*mss)
            seg_wnd= 2*mss;
        tcp_conn->tc_snd_cwnd= snd_una+seg_wnd;
        tcp_conn_write(tcp_conn, 1);
    }
    if (seg_wnd == 0)
    {
        tcp_conn->tc_snd_cwnd= tcp_conn->tc_SND_TRM=
            snd_una;
    }
}
else if (tcp_Lmod4G(snd_una, seg_ack) &&
    tcp_LEmod4G(seg_ack, snd_nxt))
{
    tcp_release_retrans(tcp_conn, seg_ack, seg_wnd);
    if (tcp_conn->tc_state == TCS_CLOSED)
        break;
}
else if (tcp_Gmod4G(seg_ack,
    snd_nxt))
{
    tcp_conn->tc_flags |= TCF_SEND_ACK;
    tcp_conn_write(tcp_conn, 1);
    DBLOCK(1, printf(
        "got an ack of something I haven't send\n");
        printf( "seg_ack= %lu, SND_NXT= %lu\n",
            seg_ack, snd_nxt));
    break;
}

/*
    process data...
*/

tcp_extract_ipopt(tcp_conn, ip_hdr);
tcp_extract_tcpopt(tcp_conn, tcp_hdr, &mss);

if (data_len)
{
    if (tcp_LEmod4G(seg_seq, tcp_conn->tc_RCV_NXT))
    {
        process_data (tcp_conn, tcp_hdr,
            tcp_data, data_len);
    }
    else
    {
        process_advanced_data (tcp_conn,
            tcp_hdr, tcp_data, data_len);
    }
}

```

```

    }
    tcp_conn->tc_flags |= TCF_SEND_ACK;
    tcp_conn_write(tcp_conn, 1);

    /* Don't process a FIN if we got new data */
    break;
}
/*
    FIN ?
    reply pending receives
    advance RCV.NXT over the FIN
    <SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

    state == ESTABLISHED ?
        state= CLOSE-WAIT
    state == FIN-WAIT-1 ?
        state= CLOSING
    state == FIN-WAIT-2 ?
        state= TIME-WAIT
    state == TIME-WAIT ?
        restart the TIME-WAIT timer
exit
*/
if ((tcp_hdr_flags & THF_FIN) && tcp_LMod4G(seg_seq,
    tcp_conn->tc_RCV_NXT))
{
    if (!(tcp_conn->tc_flags & TCF_FIN_RECV) &&
        tcp_Lmod4G(tcp_conn->tc_RCV_NXT,
            tcp_conn->tc_RCV_HI))
    {
        tcp_conn->tc_RCV_NXT++;
        tcp_conn->tc_flags |= TCF_FIN_RECV;
    }
    tcp_conn->tc_flags |= TCF_SEND_ACK;
    tcp_conn_write(tcp_conn, 1);
    if (tcp_conn->tc_fd &&
        (tcp_conn->tc_fd->tf_flags & TFF_READ_IP))
    {
        tcp_fd_read(tcp_conn, 1);
    }
    if (tcp_conn->tc_fd &&
        (tcp_conn->tc_fd->tf_flags & TFF_SEL_READ))
    {
        tcp_rsel_read(tcp_conn);
    }
}
break;
default:
    printf("tcp_frag2conn: unknown state ");
    tcp_print_state(tcp_conn);
    break;
}
if (tcp_data != NULL)
    bf_afree(tcp_data);
}

PRIVATE void
process_data(tcp_conn, tcp_hdr, tcp_data, data_len)
tcp_conn_t *tcp_conn;
tcp_hdr_t *tcp_hdr;
acc_t *tcp_data;
int data_len;
{
    u32_t lo_seq, hi_seq, urg_seq, seq_nr, adv_seq, nxt;
    u32_t urgptra;
    int tcp_hdr_flags;
    unsigned int offset;
    acc_t *tmp_data, *rcvd_data, *adv_data;
    int len_diff;

    assert(tcp_conn->tc_busy);

    /* Note, tcp_data will be freed by the caller. */

```

```

assert (!(tcp_hdr->th_flags & THF_SYN));

seq_nr= ntohl(tcp_hdr->th_seq_nr);
urgptr= ntohs(tcp_hdr->th_urgptr);

tcp_data->acc_linkC++;

lo_seq= seq_nr;
tcp_hdr_flags= tcp_hdr->th_flags & TH_FLAGS_MASK;

if (tcp_Lmod4G(lo_seq, tcp_conn->tc_RCV_NXT))
{
    DBLOCK(0x10,
        printf("segment is a retransmission\n"));
    offset= tcp_conn->tc_RCV_NXT-lo_seq;
    tcp_data= bf_delhead(tcp_data, offset);
    lo_seq += offset;
    data_len -= offset;
    if (tcp_hdr_flags & THF_URG)
    {
        printf("process_data: updating urgent pointer\n");
        if (urgptr >= offset)
            urgptr -= offset;
        else
            tcp_hdr_flags &= ~THF_URG;
    }
}
assert(lo_seq == tcp_conn->tc_RCV_NXT);

if (tcp_hdr_flags & THF_URG)
{
    if (!(tcp_conn->tc_flags & TCF_BSD_URG))
    {
        /* Update urgent pointer to point past the urgent
        * data
        */
        urgptr++;
    }
    if (urgptr == 0)
        tcp_hdr_flags &= ~THF_URG;
}

if (tcp_hdr_flags & THF_URG)
{
    if (urgptr > data_len)
        urgptr= data_len;
    urg_seq= lo_seq+urgptr;

    if (tcp_GEmod4G(urg_seq, tcp_conn->tc_RCV_HI))
        urg_seq= tcp_conn->tc_RCV_HI;
    if (tcp_conn->tc_flags & TCF_BSD_URG)
    {
        if (tcp_Gmod4G(tcp_conn->tc_RCV_NXT,
            tcp_conn->tc_RCV_LO))
        {
            DBLOCK(1, printf(
                "ignoring urgent data\n"));

            bf_afree(tcp_data);
            /* Should set advertised window to
            * zero */

            /* Flush */
            tcp_conn->tc_flags |= TCF_RCV_PUSH;
            if (tcp_conn->tc_fd &&
                (tcp_conn->tc_fd->tf_flags &
                 TFF_READ_IP))
            {
                tcp_fd_read(tcp_conn, 1);
            }
            if (tcp_conn->tc_fd &&
                (tcp_conn->tc_fd->tf_flags &
                 TFF_SEL_READ))
            {

```

```

                                tcp_rsel_read(tcp_conn);
                                }
                                return;
                                }
                                }
                                if (tcp_Gmod4G(urg_seq, tcp_conn->tc_RCV_UP))
                                    tcp_conn->tc_RCV_UP= urg_seq;
#if 0
                                if (urgptr < data_len)
                                {
                                    data_len= urgptr;
                                    tmp_data= bf_cut(tcp_data, 0, data_len);
                                    bf_afree(tcp_data);
                                    tcp_data= tmp_data;
                                    tcp_hdr_flags &= ~THF_FIN;
                                }
#endif
                                tcp_conn->tc_flags |= TCF_RCV_PUSH;
                                }
                                else
                                {
                                    /* Normal data. */
                                }

                                if (tcp_hdr_flags & THF_PSH)
                                {
                                    tcp_conn->tc_flags |= TCF_RCV_PUSH;
                                }

                                hi_seq= lo_seq+data_len;
                                if (tcp_Gmod4G(hi_seq, tcp_conn->tc_RCV_HI))
                                {
                                    data_len= tcp_conn->tc_RCV_HI-lo_seq;
                                    tmp_data= bf_cut(tcp_data, 0, data_len);
                                    bf_afree(tcp_data);
                                    tcp_data= tmp_data;
                                    hi_seq= lo_seq+data_len;
                                    tcp_hdr_flags &= ~THF_FIN;
                                }
                                assert (tcp_LEmod4G (hi_seq, tcp_conn->tc_RCV_HI));

                                rcvd_data= tcp_conn->tc_rcvd_data;
                                tcp_conn->tc_rcvd_data= 0;
                                tmp_data= bf_append(rcvd_data, tcp_data);
                                tcp_conn->tc_rcvd_data= tmp_data;
                                tcp_conn->tc_RCV_NXT= hi_seq;

                                if ((tcp_hdr_flags & THF_FIN) &&
                                    tcp_Lmod4G(tcp_conn->tc_RCV_NXT, tcp_conn->tc_RCV_HI) &&
                                    !(tcp_conn->tc_flags & TCF_FIN_RECV))
                                {
                                    tcp_conn->tc_RCV_NXT++;
                                    tcp_conn->tc_flags |= TCF_FIN_RECV;
                                }

                                if (tcp_conn->tc_fd && (tcp_conn->tc_fd->tf_flags & TFF_READ_IP))
                                    tcp_fd_read(tcp_conn, 1);
                                if (tcp_conn->tc_fd && (tcp_conn->tc_fd->tf_flags & TFF_SEL_READ))
                                    tcp_rsel_read(tcp_conn);

                                DIFBLOCK(2, (tcp_conn->tc_RCV_NXT == tcp_conn->tc_RCV_HI),
                                    printf("conn[%d] full receive buffer\n",
                                        tcp_conn-tcp_conn_table));

                                if (tcp_conn->tc_adv_data == NULL)
                                    return;
                                if (tcp_hdr_flags & THF_FIN)
                                {
                                    printf("conn[%d]: advanced data after FIN\n",
                                        tcp_conn-tcp_conn_table);
                                    tcp_data= tcp_conn->tc_adv_data;
                                    tcp_conn->tc_adv_data= NULL;
                                    bf_afree(tcp_data);
                                    return;
                                }

```

```

    }

    lo_seq= tcp_conn->tc_adv_seq;
    if (tcp_Gmod4G(lo_seq, tcp_conn->tc_RCV_NXT))
        return; /* Not yet */

    tcp_data= tcp_conn->tc_adv_data;
    tcp_conn->tc_adv_data= NULL;

    data_len= bf_bufsize(tcp_data);
    if (tcp_Lmod4G(lo_seq, tcp_conn->tc_RCV_NXT))
    {
        offset= tcp_conn->tc_RCV_NXT-lo_seq;
        if (offset >= data_len)
        {
            bf_afree(tcp_data);
            return;
        }
        tcp_data= bf_delhead(tcp_data, offset);
        lo_seq += offset;
        data_len -= offset;
    }
    assert (lo_seq == tcp_conn->tc_RCV_NXT);

    hi_seq= lo_seq+data_len;
    assert (tcp_LEmod4G (hi_seq, tcp_conn->tc_RCV_HI));

    rcvd_data= tcp_conn->tc_rcvd_data;
    tcp_conn->tc_rcvd_data= 0;
    tmp_data= bf_append(rcvd_data, tcp_data);
    tcp_conn->tc_rcvd_data= tmp_data;
    tcp_conn->tc_RCV_NXT= hi_seq;

    assert (tcp_conn->tc_RCV_LO + bf_bufsize(tcp_conn->tc_rcvd_data) ==
            tcp_conn->tc_RCV_NXT ||
            (tcp_print_conn(tcp_conn), printf("\n"), 0));

    if (tcp_conn->tc_fd && (tcp_conn->tc_fd->tf_flags & TFF_READ_IP))
        tcp_fd_read(tcp_conn, 1);
    if (tcp_conn->tc_fd && (tcp_conn->tc_fd->tf_flags & TFF_SEL_READ))
        tcp_rsel_read(tcp_conn);

    adv_data= tcp_conn->tc_adv_data;
    if (adv_data != NULL)
    {
        /* Try to use advanced data. */
        adv_seq= tcp_conn->tc_adv_seq;
        nxt= tcp_conn->tc_RCV_NXT;

        if (tcp_Gmod4G(adv_seq, nxt))
            return; /* not yet */

        tcp_conn->tc_adv_data= NULL;
        data_len= bf_bufsize(adv_data);

        if (tcp_Lmod4G(adv_seq, nxt))
        {
            if (tcp_LEmod4G(adv_seq+data_len, nxt))
            {
                /* Data is not needed anymore. */
                bf_afree(adv_data);
                return;
            }

            len_diff= nxt-adv_seq;
            adv_data= bf_delhead(adv_data, len_diff);
            data_len -= len_diff;
        }

        DBLOCK(1, printf("using advanced data\n"));

        /* Append data to the input buffer */
        if (tcp_conn->tc_rcvd_data == NULL)
        {

```

```

        tcp_conn->tc_rcvd_data= adv_data;
    }
    else
    {
        tcp_conn->tc_rcvd_data=
            bf_append(tcp_conn->tc_rcvd_data, adv_data);
    }
    tcp_conn->tc_SND_NXT += data_len;
    assert(tcp_check_conn(tcp_conn));

    if (tcp_conn->tc_fd &&
        (tcp_conn->tc_fd->tf_flags & TFF_READ_IP))
    {
        tcp_fd_read(tcp_conn, 1);
    }
    if (tcp_conn->tc_fd &&
        (tcp_conn->tc_fd->tf_flags & TFF_SEL_READ))
    {
        tcp_rsel_read(tcp_conn);
    }
}
}

```

```

PRIVATE void process_advanced_data(tcp_conn, tcp_hdr, tcp_data, data_len)
tcp_conn_t *tcp_conn;
tcp_hdr_t *tcp_hdr;
acc_t *tcp_data;
int data_len;
{
    u32_t seq, adv_seq;
    acc_t *adv_data;

    assert(tcp_conn->tc_busy);

    /* Note, tcp_data will be freed by the caller. */

    /* Always send an ACK, this allows the sender to do a fast
     * retransmit.
     */
    tcp_conn->tc_flags |= TCF_SEND_ACK;
    tcp_conn_write(tcp_conn, 1);

    if (tcp_hdr->th_flags & THF_URG)
        return; /* Urgent data is too complicated */

    if (tcp_hdr->th_flags & THF_PSH)
        tcp_conn->tc_flags |= TCF_RCV_PUSH;
    seq= ntohl(tcp_hdr->th_seq_nr);

    /* Make sure that the packet doesn't fall outside of the window
     * we offered.
     */
    if (tcp_Gmod4G(seq+data_len, tcp_conn->tc_RCV_HI))
        return;

    adv_data= tcp_conn->tc_adv_data;
    adv_seq= tcp_conn->tc_adv_seq;
    tcp_conn->tc_adv_data= NULL;

    tcp_data->acc_linkC++;
    if (adv_data == NULL)
    {
        adv_seq= seq;
        adv_data= tcp_data;
    }
    else if (seq + data_len == adv_seq)
    {
        /* New data fits right before exiting data. */
        adv_data= bf_append(tcp_data, adv_data);
        adv_seq= seq;
    }
    else if (adv_seq + bf_bufsize(adv_data) == seq)
    {
        /* New data fits right after exiting data. */

```

```

        adv_data= bf_append(adv_data, tcp_data);
    }
    else
    {
        /* New data doesn't fit. */
        bf_afree(tcp_data);
    }
    tcp_conn->tc_adv_data= adv_data;
    tcp_conn->tc_adv_seq= adv_seq;
}

PRIVATE void create_RST(tcp_conn, ip_hdr, tcp_hdr, data_len)
tcp_conn_t *tcp_conn;
ip_hdr_t *ip_hdr;
tcp_hdr_t *tcp_hdr;
int data_len;
{
    acc_t *tmp_ipopt, *tmp_tcpopt, *tcp_pack;
    acc_t *RST_acc;
    ip_hdr_t *RST_ip_hdr;
    tcp_hdr_t *RST_tcp_hdr;
    size_t pack_size, ip_hdr_len, mss;

    DBLOCK(0x10, printf("in create_RST, bad pack is:\n");
           tcp_print_pack(ip_hdr, tcp_hdr); tcp_print_state(tcp_conn);
           printf("\n"));

    assert(tcp_conn->tc_busy);

    /* Only send RST packets in reponse to actual data (or SYN, FIN)
     * this solves a problem during connection shutdown. The problem
     * is the follow senario: a senders closes the connection instead
     * of doing a shutdown and waiting for the receiver to shutdown.
     * The receiver is slow in processing the last data. After the
     * sender has completely closed the connection, the receiver
     * sends a window update which triggers the sender to send a
     * RST. The receiver closes the connection in reponse to the RST.
     */
    if ((tcp_hdr->th_flags & (THF_FIN|THF_SYN)) == 0 &&
        data_len == 0)
    {
        #if DEBUG
        { printf("tcp_rcv:create_RST: no data, no RST\n"); }
        #endif
        return;
    }

    tmp_ipopt= tcp_conn->tc_remipopt;
    if (tmp_ipopt)
        tmp_ipopt->acc_linkC++;
    tmp_tcpopt= tcp_conn->tc_tcpopt;
    if (tmp_tcpopt)
        tmp_tcpopt->acc_linkC++;

    tcp_extract_ipopt (tcp_conn, ip_hdr);
    tcp_extract_tcpopt (tcp_conn, tcp_hdr, &mss);

    RST_acc= tcp_make_header (tcp_conn, &RST_ip_hdr, &RST_tcp_hdr,
                             (acc_t *)0);

    if (tcp_conn->tc_remipopt)
        bf_afree(tcp_conn->tc_remipopt);
    tcp_conn->tc_remipopt= tmp_ipopt;
    if (tcp_conn->tc_tcpopt)
        bf_afree(tcp_conn->tc_tcpopt);
    tcp_conn->tc_tcpopt= tmp_tcpopt;

    RST_ip_hdr->ih_src= ip_hdr->ih_dst;
    RST_ip_hdr->ih_dst= ip_hdr->ih_src;

    RST_tcp_hdr->th_srcport= tcp_hdr->th_dstport;
    RST_tcp_hdr->th_dstport= tcp_hdr->th_srcport;
    if (tcp_hdr->th_flags & THF_ACK)
    {

```



```

        RST_tcp_hdr->th_seq_nr= tcp_hdr->th_ack_nr;
        RST_tcp_hdr->th_flags= THF_RST;
    }
    else
    {
        RST_tcp_hdr->th_seq_nr= 0;
        RST_tcp_hdr->th_ack_nr=
            htonl(
                ntohs(tcp_hdr->th_seq_nr)+
                data_len +
                (tcp_hdr->th_flags & THF_SYN ? 1 : 0) +
                (tcp_hdr->th_flags & THF_FIN ? 1 : 0));
        RST_tcp_hdr->th_flags= THF_RST|THF_ACK;
    }

    pack_size= bf_bufsize(RST_acc);
    RST_ip_hdr->ih_length= htons(pack_size);
    RST_tcp_hdr->th_window= htons(tcp_conn->tc_rcv_wnd);
    RST_tcp_hdr->th_chksum= 0;

    RST_acc->acc_linkC++;
    ip_hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
    tcp_pack= bf_delhead(RST_acc, ip_hdr_len);
    RST_tcp_hdr->th_chksum= ~tcp_pack_oneCsum (RST_ip_hdr, tcp_pack);
    bf_afree(tcp_pack);

    DBLOCK(2, tcp_print_pack(ip_hdr, tcp_hdr); printf("\n");
            tcp_print_pack(RST_ip_hdr, RST_tcp_hdr); printf("\n"));

    if (tcp_conn->tc_frag2send)
        bf_afree(tcp_conn->tc_frag2send);
    tcp_conn->tc_frag2send= RST_acc;
    tcp_conn_write(tcp_conn, 1);
}

PUBLIC void
tcp_fd_read(tcp_conn, enq)
tcp_conn_t *tcp_conn;
int enq;
/* Enqueue writes. */
{
    tcp_fd_t *tcp_fd;
    size_t data_size, read_size;
    acc_t *data;
    int fin_rcv, urg, push, result;
    i32_t old_window, new_window;
    ul6_t mss;

    assert(tcp_conn->tc_busy);

    tcp_fd= tcp_conn->tc_fd;

    assert (tcp_fd->tf_flags & TFF_READ_IP);
    if (tcp_conn->tc_state == TCS_CLOSED)
    {
        if (tcp_fd->tf_read_offset)
            tcp_reply_read (tcp_fd, tcp_fd->tf_read_offset);
        else
            tcp_reply_read (tcp_fd, tcp_conn->tc_error);
        return;
    }

    urg= tcp_Gmod4G(tcp_conn->tc_RCV_UP, tcp_conn->tc_RCV_LO);
    push= (tcp_conn->tc_flags & TCF_RCV_PUSH);
    fin_rcv= (tcp_conn->tc_flags & TCF_FIN_RECV);

    data_size= tcp_conn->tc_RCV_NXT-tcp_conn->tc_RCV_LO;
    if (fin_rcv)
        data_size--;
    if (urg)
    {
#ifdef DEBUG
        printf("tcp_fd_read: RCV_UP = 0x%x, RCV_LO = 0x%x\n",
            tcp_conn->tc_RCV_UP, tcp_conn->tc_RCV_LO);
#endif
    }
}

```

```
        read_size= tcp_conn->tc_RCV_UP-tcp_conn->tc_RCV_LO;
    }
    else
        read_size= data_size;

    if (read_size >= tcp_fd->tf_read_count)
        read_size= tcp_fd->tf_read_count;
    else if (!push && !fin_recv && !urg &&
        data_size < TCP_MIN_RCV_WND_SIZE)
    {
        /* Defer the copy out until later. */
        return;
    }
    else if (data_size == 0 && !fin_recv)
    {
        /* No data, and no end of file. */
        return;
    }

    if (read_size)
    {
        if (urg && !(tcp_fd->tf_flags & TFF_RECV_URG))
        {
            if (tcp_fd->tf_read_offset)
            {
                tcp_reply_read (tcp_fd,
                               tcp_fd->tf_read_offset);
            }
            else
            {
                tcp_reply_read (tcp_fd, EURG);
            }
            return;
        }
        else if (!urg && (tcp_fd->tf_flags & TFF_RECV_URG))
        {
            if (tcp_fd->tf_read_offset)
            {
                tcp_reply_read (tcp_fd,
                               tcp_fd->tf_read_offset);
            }
            else
            {
                tcp_reply_read(tcp_fd, ENOURG);
            }
            return;
        }

        if (read_size == data_size)
        {
            data= tcp_conn->tc_rcvd_data;
            data->acc_linkC++;
        }
        else
        {
            data= bf_cut(tcp_conn->tc_rcvd_data, 0, read_size);
        }
        result= (*tcp_fd->tf_put_userdata) (tcp_fd->tf_srfd,
            tcp_fd->tf_read_offset, data, FALSE);
        if (result<0)
        {
            if (tcp_fd->tf_read_offset)
                tcp_reply_read(tcp_fd, tcp_fd->
                               tf_read_offset);
            else
                tcp_reply_read(tcp_fd, result);
            return;
        }
        tcp_fd->tf_read_offset += read_size;
        tcp_fd->tf_read_count -= read_size;

        if (data_size == read_size)
        {
            bf_afree(tcp_conn->tc_rcvd_data);
        }
    }
}
```

```

        tcp_conn->tc_rcvd_data= 0;
    }
    else
    {
        tcp_conn->tc_rcvd_data=
            bf_delhead(tcp_conn->tc_rcvd_data,
                read_size);
    }
    tcp_conn->tc_RCV_LO += read_size;
    data_size -= read_size;
}

/* Update IRS and often RCV_UP every 0.5GB */
if (tcp_conn->tc_RCV_LO - tcp_conn->tc_IRS > 0x40000000)
{
    tcp_conn->tc_IRS += 0x20000000;
    DBLOCK(1, printf("tcp_fd_read: updating IRS to 0x%lx\n",
        (unsigned long)tcp_conn->tc_IRS));
    if (tcp_Lmod4G(tcp_conn->tc_RCV_UP, tcp_conn->tc_IRS))
    {
        tcp_conn->tc_RCV_UP= tcp_conn->tc_IRS;
        DBLOCK(1, printf(
            "tcp_fd_read: updating RCV_UP to 0x%lx\n",
            (unsigned long)tcp_conn->tc_RCV_UP));
    }
    DBLOCK(1, printf("tcp_fd_read: RCP_LO = 0x%lx\n",
        (unsigned long)tcp_conn->tc_RCV_LO));
}

mss= tcp_conn->tc_mtu-IP_TCP_MIN_HDR_SIZE;
if (tcp_conn->tc_RCV_HI-tcp_conn->tc_RCV_LO <=
    tcp_conn->tc_rcv_wnd-mss)
{
    old_window= tcp_conn->tc_RCV_HI-tcp_conn->tc_RCV_NXT;
    tcp_conn->tc_RCV_HI= tcp_conn->tc_RCV_LO +
        tcp_conn->tc_rcv_wnd;
    new_window= tcp_conn->tc_RCV_HI-tcp_conn->tc_RCV_NXT;
    assert(old_window >=0 && new_window >= old_window);
    if (old_window < mss && new_window >= mss)
    {
        tcp_conn->tc_flags |= TCF_SEND_ACK;
        DBLOCK(2, printf("opening window\n"));
        tcp_conn_write(tcp_conn, 1);
    }
}
if (tcp_conn->tc_rcvd_data == NULL &&
    tcp_conn->tc_adv_data == NULL)
{
    /* Out of data, clear PUSH flag and reply to a read. */
    tcp_conn->tc_flags &= ~TCF_RCV_PUSH;
}
if (fin_recv || urg || tcp_fd->tf_read_offset ||
    !tcp_fd->tf_read_count)
{
    tcp_reply_read (tcp_fd, tcp_fd->tf_read_offset);
    return;
}
}

PUBLIC unsigned
tcp_sel_read(tcp_conn)
tcp_conn_t *tcp_conn;
{
    tcp_fd_t *tcp_fd;
    size_t data_size;
    int fin_recv, urg, push;

    tcp_fd= tcp_conn->tc_fd;

    if (tcp_conn->tc_state == TCS_CLOSED)
        return 1;

    fin_recv= (tcp_conn->tc_flags & TCF_FIN_RECV);
    if (fin_recv)

```

```

        return 1;

data_size= tcp_conn->tc_RCV_NXT-tcp_conn->tc_RCV_LO;
if (data_size == 0)
{
    /* No data, and no end of file. */
    return 0;
}

urg= tcp_Gmod4G(tcp_conn->tc_RCV_UP, tcp_conn->tc_RCV_LO);
push= (tcp_conn->tc_flags & TCF_RCV_PUSH);

if (!push && !urg && data_size < TCP_MIN_RCV_WND_SIZE)
{
    /* Defer until later. */
    return 0;
}

return 1;
}

PUBLIC void
tcp_rsel_read(tcp_conn)
tcp_conn_t *tcp_conn;
{
    tcp_fd_t *tcp_fd;

    if (tcp_sel_read(tcp_conn) == 0)
        return;

    tcp_fd= tcp_conn->tc_fd;
    tcp_fd->tf_flags &= ~TFF_SEL_READ;
    if (tcp_fd->tf_select_res)
        tcp_fd->tf_select_res(tcp_fd->tf_srfd, SR_SELECT_READ);
    else
        printf("tcp_rsel_read: no select_res\n");
}

PUBLIC void tcp_bytesavailable(tcp_fd, bytesp)
tcp_fd_t *tcp_fd;
int *bytesp;
{
    tcp_conn_t *tcp_conn;
    size_t data_size, read_size;
    acc_t *data;
    int fin_recv, urg, push, result;
    i32_t old_window, new_window;
    ul6_t mss;

    *bytesp= 0;    /* The default is that nothing is available */

    if (!(tcp_fd->tf_flags & TFF_CONNECTED))
        return;
    tcp_conn= tcp_fd->tf_conn;

    if (tcp_conn->tc_state == TCS_CLOSED)
        return;

    urg= tcp_Gmod4G(tcp_conn->tc_RCV_UP, tcp_conn->tc_RCV_LO);
    push= (tcp_conn->tc_flags & TCF_RCV_PUSH);
    fin_recv= (tcp_conn->tc_flags & TCF_FIN_RECV);

    data_size= tcp_conn->tc_RCV_NXT-tcp_conn->tc_RCV_LO;
    if (fin_recv)
        data_size--;
    if (urg)
        data_size= tcp_conn->tc_RCV_UP-tcp_conn->tc_RCV_LO;

    if (urg && !(tcp_fd->tf_flags & TFF_RECV_URG))
        return;
    else if (!urg && (tcp_fd->tf_flags & TFF_RECV_URG))
        return;

    *bytesp= data_size;

```

```
}  
/*  
 * $PchId: tcp_recv.c,v 1.30 2005/06/28 14:21:35 philip Exp $  
 */
```

```
/*
tcp_send.c

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "buf.h"
#include "clock.h"
#include "event.h"
#include "type.h"
#include "sr.h"

#include "assert.h"
#include "io.h"
#include "ip.h"
#include "tcp.h"
#include "tcp_int.h"

THIS_FILE

FORWARD acc_t *make_pack ARGS(( tcp_conn_t *tcp_conn ));
FORWARD void tcp_send_timeout ARGS(( int conn, struct timer *timer ));
FORWARD void do_snd_event ARGS(( event_t *ev, ev_arg_t arg ));

PUBLIC void tcp_conn_write (tcp_conn, enq)
tcp_conn_t *tcp_conn;
int enq;                                /* Writes need to be enqueued. */
{
    tcp_port_t *tcp_port;
    ev_arg_t snd_arg;

    assert (tcp_conn->tc_flags & TCF_INUSE);

    tcp_port= tcp_conn->tc_port;
    if (tcp_conn->tc_flags & TCF_MORE2WRITE)
        return;

    /* Do we really have something to send here? */
    if (tcp_conn->tc_SND_UNA == tcp_conn->tc_SND_NXT &&
        !(tcp_conn->tc_flags & TCF_SEND_ACK) &&
        !tcp_conn->tc_frag2send)
    {
        return;
    }

    tcp_conn->tc_flags |= TCF_MORE2WRITE;
    tcp_conn->tc_send_link= NULL;
    if (!tcp_port->tp_snd_head)
    {
        tcp_port->tp_snd_head= tcp_conn;
        tcp_port->tp_snd_tail= tcp_conn;
        if (enq)
        {
            snd_arg.ev_ptr= tcp_port;
            if (!ev_in_queue(&tcp_port->tp_snd_event))
            {
                ev_enqueue(&tcp_port->tp_snd_event,
                           do_snd_event, snd_arg);
            }
        }
        else
            tcp_port_write(tcp_port);
    }
    else
    {
        tcp_port->tp_snd_tail->tc_send_link= tcp_conn;
        tcp_port->tp_snd_tail= tcp_conn;
    }
}

PRIVATE void do_snd_event(ev, arg)
event_t *ev;
ev_arg_t arg;
```

```

{
    tcp_port_t *tcp_port;

    tcp_port= arg.ev_ptr;

    assert(ev == &tcp_port->tp_snd_event);
    tcp_port_write(tcp_port);
}

PUBLIC void tcp_port_write(tcp_port)
tcp_port_t *tcp_port;
{
    tcp_conn_t *tcp_conn;
    acc_t *pack2write;
    int r;

    assert (!(tcp_port->tp_flags & TPF_WRITE_IP));

    while(tcp_port->tp_snd_head)
    {
        tcp_conn= tcp_port->tp_snd_head;
        assert(tcp_conn->tc_flags & TCF_MORE2WRITE);

        for(;;)
        {
            if (tcp_conn->tc_frag2send)
            {
                pack2write= tcp_conn->tc_frag2send;
                tcp_conn->tc_frag2send= 0;
            }
            else
            {
                tcp_conn->tc_busy++;
                pack2write= make_pack(tcp_conn);
                tcp_conn->tc_busy--;
                if (!pack2write)
                    break;
            }
            r= ip_send(tcp_port->tp_ipfd, pack2write,
                bf_bufsize(pack2write));
            if (r != NW_OK)
            {
                if (r == NW_WOULDBLOCK)
                    break;
                if (r == EPACKSIZE)
                {
                    tcp_mtu_exceeded(tcp_conn);
                    continue;
                }
                if (r == EDSTNOTRCH)
                {
                    tcp_notreach(tcp_conn);
                    continue;
                }
                if (r == EBADDEST)
                    continue;
            }
            assert(r == NW_OK ||
                (printf("ip_send failed, error %d\n", r), 0));
        }

        if (pack2write)
        {
            tcp_port->tp_flags |= TPF_WRITE_IP;
            tcp_port->tp_pack= pack2write;

            r= ip_write (tcp_port->tp_ipfd,
                bf_bufsize(pack2write));
            if (r == NW_SUSPEND)
            {
                tcp_port->tp_flags |= TPF_WRITE_SP;
                return;
            }
            assert(r == NW_OK);
        }
    }
}

```

```

        tcp_port->tp_flags &= ~TPF_WRITE_IP;
        assert(!(tcp_port->tp_flags &
            (TPF_WRITE_IP|TPF_WRITE_SP)));
        continue;
    }
    tcp_conn->tc_flags &= ~TCF_MORE2WRITE;
    tcp_port->tp_snd_head= tcp_conn->tc_send_link;
}

}

PRIVATE acc_t *make_pack(tcp_conn)
tcp_conn_t *tcp_conn;
{
    acc_t *pack2write, *tmp_pack, *tcp_pack;
    tcp_hdr_t *tcp_hdr;
    ip_hdr_t *ip_hdr;
    int tot_hdr_size, ip_hdr_len, no_push, head, more2write;
    u32_t seg_seq, seg_lo_data, queue_lo_data, seg_hi, seg_hi_data;
    u16_t seg_up, mss;
    u8_t seg_flags;
    size_t pack_size;
    clock_t curr_time, new_dis;
    u8_t *optptr;

    mss= tcp_conn->tc_mtu-IP_TCP_MIN_HDR_SIZE;

    assert(tcp_conn->tc_busy);
    curr_time= get_time();
    switch (tcp_conn->tc_state)
    {
    case TCS_CLOSED:
    case TCS_LISTEN:
        return NULL;
    case TCS_SYN_RECEIVED:
    case TCS_SYN_SENT:

        if (tcp_conn->tc_SND_TRM == tcp_conn->tc_SND_NXT &&
            !(tcp_conn->tc_flags & TCF_SEND_ACK))
        {
            return 0;
        }

        tcp_conn->tc_flags &= ~TCF_SEND_ACK;

        /* Advertise a mss based on the port mtu. The current mtu may
         * be lower if the other side sends a smaller mss.
         */
        mss= tcp_conn->tc_port->tp_mtu-IP_TCP_MIN_HDR_SIZE;

        /* Include a max segment size option. */
        assert(tcp_conn->tc_tcpopt == NULL);
        tcp_conn->tc_tcpopt= bf_memreq(4);
        optptr= (u8_t *)ptr2acc_data(tcp_conn->tc_tcpopt);
        optptr[0]= TCP_OPT_MSS;
        optptr[1]= 4;
        optptr[2]= mss >> 8;
        optptr[3]= mss & 0xFF;

        pack2write= tcp_make_header(tcp_conn, &ip_hdr, &tcp_hdr,
            (acc_t *)0);

        bf_afree(tcp_conn->tc_tcpopt);
        tcp_conn->tc_tcpopt= NULL;

        if (!pack2write)
        {
            DBLOCK(1, printf("connection closed while inuse\n"));
            return 0;
        }
        tot_hdr_size= bf_bufsize(pack2write);
        seg_seq= tcp_conn->tc_SND_TRM;
        if (tcp_conn->tc_state == TCS_SYN_SENT)
            seg_flags= 0;

```



```

        else
            seg_flags= THF_ACK;          /* except for TCS_SYN_SENT
                                           * ack is always present */

        if (seg_seq == tcp_conn->tc_ISS)
        {
            assert(tcp_conn->tc_transmit_timer.tim_active ||
                   (tcp_print_conn(tcp_conn), printf("\n"), 0));
            seg_flags |= THF_SYN;
            tcp_conn->tc_SND_TRM++;
        }

        tcp_hdr->th_seq_nr= htonl(seg_seq);
        tcp_hdr->th_ack_nr= htonl(tcp_conn->tc_RCV_NXT);
        tcp_hdr->th_flags= seg_flags;
        tcp_hdr->th_window= htons(mss);
        /* Initially we allow one segment */

        ip_hdr->ih_length= htons(tot_hdr_size);

        pack2write->acc_linkC++;
        ip_hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
        tcp_pack= bf_delhead(pack2write, ip_hdr_len);
        tcp_hdr->th_chksum= ~tcp_pack_oneCsum(ip_hdr, tcp_pack);
        bf_afree(tcp_pack);

        new_dis= curr_time + 2*HZ*tcp_conn->tc_ttl;
        if (new_dis > tcp_conn->tc_senddis)
            tcp_conn->tc_senddis= new_dis;
        return pack2write;

    case TCS_ESTABLISHED:
    case TCS_CLOSING:
        seg_seq= tcp_conn->tc_SND_TRM;

        seg_flags= 0;
        pack2write= 0;
        seg_up= 0;
        if (tcp_conn->tc_flags & TCF_SEND_ACK)
        {
            seg_flags= THF_ACK;
            tcp_conn->tc_flags &= ~TCF_SEND_ACK;

            pack2write= tcp_make_header (tcp_conn, &ip_hdr,
                                         &tcp_hdr, (acc_t *)0);
            if (!pack2write)
            {
                return NULL;
            }
        }

        if (tcp_conn->tc_SND_UNA != tcp_conn->tc_SND_NXT)
        {
            assert(tcp_LEmod4G(seg_seq, tcp_conn->tc_SND_NXT));

            if (seg_seq == tcp_conn->tc_snd_cwnd)
            {
                DBLOCK(2,
                       printf("no data: window is closed\n"));
                goto after_data;
            }

            /* Assert that our SYN has been ACKed. */
            assert(tcp_conn->tc_SND_UNA != tcp_conn->tc_ISS);

            seg_lo_data= seg_seq;
            queue_lo_data= tcp_conn->tc_SND_UNA;

            seg_hi= tcp_conn->tc_SND_NXT;
            seg_hi_data= seg_hi;
            if (tcp_conn->tc_flags & TCF_FIN_SENT)
            {
                if (seg_seq != seg_hi)
                    seg_flags |= THF_FIN;
            }
        }

```

```

        if (queue_lo_data == seg_hi_data)
            queue_lo_data--;
        if (seg_lo_data == seg_hi_data)
            seg_lo_data--;
        seg_hi_data--;
    }

    if (!pack2write)
    {
        pack2write= tcp_make_header (tcp_conn,
            &ip_hdr, &tcp_hdr, (acc_t *)0);
        if (!pack2write)
        {
            return NULL;
        }
    }

    tot_hdr_size= bf_bufsize(pack2write);

    no_push= (tcp_LEmod4G(tcp_conn->tc_SND_PSH, seg_seq));
    head= (seg_seq == tcp_conn->tc_SND_UNA);
    if (no_push)
    {
        /* Shutdown sets SND_PSH */
        seg_flags &= ~THF_FIN;
        if (seg_hi_data-seg_lo_data <= 1)
        {
            /* Always keep at least one byte
             * for a future push.
             */
            DBLOCK(0x20,
                printf("no data: no push\n"));
            if (head)
            {
                DBLOCK(0x1, printf(
                    "no data: setting TCF_NO_PUSH\n"));
                tcp_conn->tc_flags |=
                    TCF_NO_PUSH;
            }
            goto after_data;
        }
        seg_hi_data--;
    }

    if (tot_hdr_size != IP_TCP_MIN_HDR_SIZE)
    {
        printf(
            "tcp_write'make_pack: tot_hdr_size = %d\n",
            tot_hdr_size);
        mss= tcp_conn->tc_mtu-tot_hdr_size;
    }
    if (seg_hi_data - seg_lo_data > mss)
    {
        /* Truncate to at most one segment */
        seg_hi_data= seg_lo_data + mss;
        seg_hi= seg_hi_data;
        seg_flags &= ~THF_FIN;
    }

    if (no_push &&
        seg_hi_data-seg_lo_data != mss)
    {
        DBLOCK(0x20, printf(
            "no data: no push for partial segment\n"));
        more2write= (tcp_conn->tc_fd &&
            (tcp_conn->tc_fd->tf_flags &
                TFF_WRITE_IP));
        DIFBLOCK(2, more2write,
            printf(
                "tcp_send'make_pack: more2write -> !TCF_NO_PUSH\n"));
        if (head && !more2write)
        {
            DBLOCK(0x1, printf(

```

```
        "partial segment: setting TCF_NO_PUSH\n" ));
        tcp_conn->tc_flags |= TCF_NO_PUSH;
    }
    goto after_data;
}

if (tcp_Gmod4G(seg_hi, tcp_conn->tc_snd_cwnd))
{
    seg_hi_data= tcp_conn->tc_snd_cwnd;
    seg_hi= seg_hi_data;
    seg_flags &= ~THF_FIN;
}

if (!head &&
    seg_hi_data-seg_lo_data < mss)
{
    if (tcp_conn->tc_flags & TCF_PUSH_NOW)
    {
        DBLOCK(0x20,
            printf("push: no Nagle\n"));
    }
    else
    {
        DBLOCK(0x20,
            printf("no data: partial packet\n"));
        seg_flags &= ~THF_FIN;
        goto after_data;
    }
}

if (seg_hi-seg_seq == 0)
{
    DBLOCK(0x20,
        printf("no data: no data available\n"));
    goto after_data;
}

if (tcp_GEmod4G(tcp_conn->tc_SND_UP, seg_lo_data))
{
    extern int killer_inet;

    if (tcp_GEmod4G(tcp_conn->tc_SND_UP,
        seg_hi_data))
    {
        seg_up= seg_hi_data-seg_seq;
    }
    else
    {
        seg_up= tcp_conn->tc_SND_UP-seg_seq;
    }
    seg_flags |= THF_URG;
    if (!killer_inet &&
        (tcp_conn->tc_flags & TCF_BSD_URG) &&
        seg_up == 0)
    {
        /* A zero urgent pointer doesn't mean
         * anything when BSD semantics are
         * used (urgent pointer points to the
         * first no urgent byte). The use of
         * a zero urgent pointer also crashes
         * a Solaris 2.3 kernel. If urgent
         * pointer doesn't have BSD semantics
         * then an urgent pointer of zero
         * simply indicates that there is one
         * urgent byte.
         */
        seg_flags &= ~THF_URG;
    }
}
else
    seg_up= 0;

if (tcp_Gmod4G(tcp_conn->tc_SND_PSH, seg_lo_data) &&
```

```

        tcp_LEmod4G(tcp_conn->tc_SND_PSH, seg_hi_data))
    {
        seg_flags |= THF_PSH;
    }

    tcp_conn->tc_SND_TRM= seg_hi;

    assert(tcp_conn->tc_transmit_timer.tim_active ||
        (tcp_print_conn(tcp_conn), printf("\n"), 0));
    if (tcp_conn->tc_rt_seq == 0 &&
        tcp_Gmod4G(seg_seq, tcp_conn->tc_rt_threshold))
    {
        tcp_conn->tc_rt_time= curr_time;
        tcp_conn->tc_rt_seq=
            tcp_conn->tc_rt_threshold= seg_seq;
    }

    if (seg_hi_data-seg_lo_data)
    {
        assert(tcp_check_conn(tcp_conn));
        assert((seg_hi_data-queue_lo_data <=
            bf_bufsize(tcp_conn->tc_send_data) &&
            seg_lo_data-queue_lo_data <=
            bf_bufsize(tcp_conn->tc_send_data) &&
            seg_hi_data>seg_lo_data)||
            (tcp_print_conn(tcp_conn),
            printf(
" seg_hi_data= 0x%x, seg_lo_data= 0x%x, queue_lo_data= 0x%x\n",
            seg_hi_data, seg_lo_data,
            queue_lo_data), 0));

        tmp_pack= pack2write;
        while (tmp_pack->acc_next)
            tmp_pack= tmp_pack->acc_next;
        tmp_pack->acc_next=
            bf_cut(tcp_conn->tc_send_data,
                (unsigned)(seg_lo_data-queue_lo_data),
                (unsigned)(seg_hi_data-seg_lo_data));
    }
    seg_flags |= THF_ACK;
}

after_data:
if (!(seg_flags & THF_ACK))
{
    if (pack2write)
        bf_afree(pack2write);
    return NULL;
}

tcp_hdr->th_seq_nr= htonl(seg_seq);
tcp_hdr->th_ack_nr= htonl(tcp_conn->tc_RCV_NXT);
tcp_hdr->th_flags= seg_flags;
tcp_hdr->th_window= htons(tcp_conn->tc_RCV_HI -
    tcp_conn->tc_RCV_NXT);
tcp_hdr->th_urgptr= htons(seg_up);

pack_size= bf_bufsize(pack2write);
ip_hdr->ih_length= htons(pack_size);

pack2write->acc_linkC++;
ip_hdr_len= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
tcp_pack= bf_delhead(pack2write, ip_hdr_len);
tcp_hdr->th_chksum= ~tcp_pack_oneCsum(ip_hdr, tcp_pack);
bf_afree(tcp_pack);

new_dis= curr_time + 2*HZ*tcp_conn->tc_ttl;
if (new_dis > tcp_conn->tc_senddis)
    tcp_conn->tc_senddis= new_dis;

return pack2write;
default:

```

```

        DBLOCK(1, tcp_print_conn(tcp_conn); printf("\n"));
        ip_panic(( "Illegal state" ));
    }
    assert(0);
    return NULL;
}

/*
tcp_release_retrans
*/

PUBLIC void tcp_release_retrans(tcp_conn, seg_ack, new_win)
tcp_conn_t *tcp_conn;
u32_t seg_ack;
ul6_t new_win;
{
    tcp_fd_t *tcp_fd;
    size_t size, offset;
    acc_t *pack;
    clock_t retrans_time, curr_time, rtt, artt, drtt, srtt;
    u32_t queue_lo, queue_hi;
    ul6_t mss, cthresh;
    unsigned window;

    DBLOCK(0x10, printf("tcp_release_retrans, conn[%d]: ack %lu, win %u\n",
        tcp_conn-tcp_conn_table, (unsigned long)seg_ack, new_win));

    assert(tcp_conn->tc_busy);
    assert (tcp_GEmod4G(seg_ack, tcp_conn->tc_SND_UNA));
    assert (tcp_LEmod4G(seg_ack, tcp_conn->tc_SND_NXT));

    tcp_conn->tc_snd_dack= 0;
    mss= tcp_conn->tc_mtu-IP_TCP_MIN_HDR_SIZE;

    curr_time= get_time();
    if (tcp_conn->tc_rt_seq != 0 &&
        tcp_Gmod4G(seg_ack, tcp_conn->tc_rt_seq))
    {
        assert(curr_time >= tcp_conn->tc_rt_time);
        retrans_time= curr_time-tcp_conn->tc_rt_time;
        rtt= tcp_conn->tc_rtt;

        tcp_conn->tc_rt_seq= 0;

        if (rtt == TCP_RTT_GRAN*CLOCK_GRAN &&
            retrans_time <= TCP_RTT_GRAN*CLOCK_GRAN)
        {
            /* Common in fast networks. Nothing to do. */
        }
        else
        {
            srtt= retrans_time * TCP_RTT_SCALE;

            artt= tcp_conn->tc_artt;
            artt= ((TCP_RTT_SMOOTH-1)*artt+srtt)/TCP_RTT_SMOOTH;

            srtt -= artt;
            if (srtt < 0)
                srtt= -srtt;
            drtt= tcp_conn->tc_drtt;
            drtt= ((TCP_RTT_SMOOTH-1)*drtt+srtt)/TCP_RTT_SMOOTH;

            rtt= (artt+TCP_DRTT_MULT*drtt-1)/TCP_RTT_SCALE+1;
            if (rtt < TCP_RTT_GRAN*CLOCK_GRAN)
            {
                rtt= TCP_RTT_GRAN*CLOCK_GRAN;
            }
            else if (rtt > TCP_RTT_MAX)
            {

```

```

#if DEBUG

```

```

        static int warned /* = 0 */;

```

```

        if (!warned)
        {

```

```

                                printf(
"tcp_release_retrans: warning retransmission time is limited to %d ms\n",
                                TCP_RTT_MAX*1000/HZ);
                                warned= 1;
                                }
#endif

                                rtt= TCP_RTT_MAX;
                                }
                                DBLOCK(0x10, printf(
"tcp_release_retrans, conn[%d]: retrans_time= %ld ms, rtt = %ld ms\n",
                                tcp_conn-tcp_conn_table,
                                retrans_time*1000/HZ,
                                rtt*1000/HZ));

                                DBLOCK(0x10, printf(
"tcp_release_retrans: artt= %ld -> %ld, drtt= %ld -> %ld\n",
                                tcp_conn->tc_artt, artt,
                                tcp_conn->tc_drtt, drtt));

                                tcp_conn->tc_artt= artt;
                                tcp_conn->tc_drtt= drtt;
                                tcp_conn->tc_rtt= rtt;
                                }

                                if (tcp_conn->tc_mtu != tcp_conn->tc_max_mtu &&
                                    curr_time > tcp_conn->tc_mtutim+TCP_PMTU_INCR_IV)
                                {
                                    tcp_mtu_incr(tcp_conn);
                                }
                                }

/* Update the current window. */
window= tcp_conn->tc_snd_cwnd-tcp_conn->tc_SND_UNA;
assert(seg_ack != tcp_conn->tc_SND_UNA);

/* For every real ACK we try to increase the current window
 * with 1 mss.
 */
window += mss;

/* If the window becomes larger than the current threshold,
 * increment the threshold by a small amount and set the
 * window to the threshold.
 */
cthresh= tcp_conn->tc_snd_cthresh;
if (window > cthresh)
{
    cthresh += tcp_conn->tc_snd_cinc;
    tcp_conn->tc_snd_cthresh= cthresh;
    window= cthresh;
}

/* If the window is larger than the window advertised by the
 * receiver, set the window size to the advertisement.
 */
if (window > new_win)
    window= new_win;

tcp_conn->tc_snd_cwnd= seg_ack+window;

/* Release data queued for retransmissions. */
queue_lo= tcp_conn->tc_SND_UNA;
queue_hi= tcp_conn->tc_SND_NXT;

tcp_conn->tc_SND_UNA= seg_ack;
if (tcp_Lmod4G(tcp_conn->tc_SND_TRM, seg_ack))
{
    tcp_conn->tc_SND_TRM= seg_ack;
}
assert(tcp_GEmod4G(tcp_conn->tc_snd_cwnd, seg_ack));

/* Advance ISS every 0.5GB to avoid problem with wrap around */
if (tcp_conn->tc_SND_UNA - tcp_conn->tc_ISS > 0x40000000)
{

```

```

        tcp_conn->tc_ISS += 0x20000000;
        DBLOCK(1, printf(
            "tcp_release_retrans: updating ISS to 0x%lx\n",
            (unsigned long)tcp_conn->tc_ISS));
        if (tcp_Lmod4G(tcp_conn->tc_SND_UP, tcp_conn->tc_ISS))
        {
            tcp_conn->tc_SND_UP= tcp_conn->tc_ISS;
            DBLOCK(1, printf(
                "tcp_release_retrans: updating SND_UP to 0x%lx\n",
                (unsigned long)tcp_conn->tc_SND_UP));
        }
    }

    if (queue_lo == tcp_conn->tc_ISS)
        queue_lo++;

    if (tcp_conn->tc_flags & TCF_FIN_SENT)
    {
        if (seg_ack == queue_hi)
            seg_ack--;
        if (queue_lo == queue_hi)
            queue_lo--;
        queue_hi--;
    }

    offset= seg_ack - queue_lo;
    size= queue_hi - seg_ack;
    pack= tcp_conn->tc_send_data;
    tcp_conn->tc_send_data= 0;

    if (!size)
    {
        bf_afree(pack);
    }
    else
    {
        pack= bf_delhead(pack, offset);
        tcp_conn->tc_send_data= pack;
    }

    if (tcp_Gmod4G(tcp_conn->tc_SND_TRM, tcp_conn->tc_snd_cwnd))
        tcp_conn->tc_SND_TRM= tcp_conn->tc_snd_cwnd;

    /* Copy in new data if an ioctl is pending or if a write request is
     * pending and either the write can be completed or at least one
     * mss buffer space is available.
     */
    tcp_fd= tcp_conn->tc_fd;
    if (tcp_fd)
    {
        if (tcp_fd->tf_flags & TFF_IOCTL_IP)
        {
            tcp_fd_write(tcp_conn);
        }
        if ((tcp_fd->tf_flags & TFF_WRITE_IP) &&
            (size+tcp_fd->tf_write_count <= TCP_MAX_SND_WND_SIZE ||
             size <= TCP_MAX_SND_WND_SIZE-mss))
        {
            tcp_fd_write(tcp_conn);
        }
        if (tcp_fd->tf_flags & TFF_SEL_WRITE)
            tcp_rsel_write(tcp_conn);
    }
    else
    {
        if (tcp_conn->tc_SND_UNA == tcp_conn->tc_SND_NXT)
        {
            assert(tcp_conn->tc_state == TCS_CLOSING);
            DBLOCK(0x10,
                printf("all data sent in abandoned connection\n"));
            tcp_close_connection(tcp_conn, ENOTCONN);
            return;
        }
    }
}

```

```

    if (!size && !tcp_conn->tc_send_data)
    {
        /* Reset window if a write is completed */
        tcp_conn->tc_snd_cwnd= tcp_conn->tc_SND_UNA + mss;
    }

    DIFBLOCK(2, (tcp_conn->tc_snd_cwnd == tcp_conn->tc_SND_TRM),
        printf("not sending: zero window\n"));

    if (tcp_conn->tc_snd_cwnd != tcp_conn->tc_SND_TRM &&
        tcp_conn->tc_SND_NXT != tcp_conn->tc_SND_TRM)
    {
        tcp_conn_write(tcp_conn, 1);
    }
}

/*
tcp_fast_retrans
*/

PUBLIC void tcp_fast_retrans(tcp_conn)
tcp_conn_t *tcp_conn;
{
    ul6_t mss, mss2;

    /* Update threshold sequence number for retransmission calculation. */
    if (tcp_Gmod4G(tcp_conn->tc_SND_TRM, tcp_conn->tc_rt_threshold))
        tcp_conn->tc_rt_threshold= tcp_conn->tc_SND_TRM;

    tcp_conn->tc_SND_TRM= tcp_conn->tc_SND_UNA;

    mss= tcp_conn->tc_mtu-IP_TCP_MIN_HDR_SIZE;
    mss2= 2*mss;

    if (tcp_conn->tc_snd_cwnd == tcp_conn->tc_SND_UNA)
        tcp_conn->tc_snd_cwnd++;
    if (tcp_Gmod4G(tcp_conn->tc_snd_cwnd, tcp_conn->tc_SND_UNA + mss2))
    {
        tcp_conn->tc_snd_cwnd= tcp_conn->tc_SND_UNA + mss2;
        if (tcp_Gmod4G(tcp_conn->tc_SND_TRM, tcp_conn->tc_snd_cwnd))
            tcp_conn->tc_SND_TRM= tcp_conn->tc_snd_cwnd;

        tcp_conn->tc_snd_cthresh /= 2;
        if (tcp_conn->tc_snd_cthresh < mss2)
            tcp_conn->tc_snd_cthresh= mss2;
    }

    tcp_conn_write(tcp_conn, 1);
}

#if 0
PUBLIC void do_tcp_timeout(tcp_conn)
tcp_conn_t *tcp_conn;
{
    tcp_send_timeout(tcp_conn-tcp_conn_table,
        &tcp_conn->tc_transmit_timer);
}
#endif

/*
tcp_send_timeout
*/

PRIVATE void tcp_send_timeout(conn, timer)
int conn;
struct timer *timer;
{
    tcp_conn_t *tcp_conn;
    ul6_t mss, mss2;
    u32_t snd_una, snd_nxt;
    clock_t curr_time, rtt, stt, timeout;
    acc_t *pkt;

```



```

int new_ttl, no_push;

DBLOCK(0x20, printf("tcp_send_timeout: conn[%d]\n", conn));

curr_time= get_time();

tcp_conn= &tcp_conn_table[conn];
assert(tcp_conn->tc_flags & TCF_INUSE);
assert(tcp_conn->tc_state != TCS_CLOSED);
assert(tcp_conn->tc_state != TCS_LISTEN);

snd_una= tcp_conn->tc_SND_UNA;
snd_nxt= tcp_conn->tc_SND_NXT;
no_push= (tcp_conn->tc_flags & TCF_NO_PUSH);
if (snd_nxt == snd_una || no_push)
{
    /* Nothing more to send */
    assert(tcp_conn->tc_SND_TRM == snd_una || no_push);

    /* A new write sets the timer if tc_transmit_seq == SND_UNA */
    tcp_conn->tc_transmit_seq= tcp_conn->tc_SND_UNA;
    tcp_conn->tc_stt= 0;
    tcp_conn->tc_0wnd_to= 0;
    assert(!tcp_conn->tc_fd ||
        !(tcp_conn->tc_fd->tf_flags & TFF_WRITE_IP) ||
        (tcp_print_conn(tcp_conn), printf("\n"), 0));

    if (snd_nxt != snd_una)
    {
        assert(no_push);
        DBLOCK(1, printf("not setting keepalive timer\n"));

        /* No point in setting the keepalive timer if we
         * still have to send more data.
         */
        return;
    }

    assert(tcp_conn->tc_send_data == NULL);
    DBLOCK(0x20, printf("keep alive timer\n"));
    if (tcp_conn->tc_ka_snd != tcp_conn->tc_SND_NXT ||
        tcp_conn->tc_ka_rcv != tcp_conn->tc_RCV_NXT)
    {
        tcp_conn->tc_ka_snd= tcp_conn->tc_SND_NXT;
        tcp_conn->tc_ka_rcv= tcp_conn->tc_RCV_NXT;
        DBLOCK(0x20, printf(
"tcp_send_timeout: conn[%d] setting keepalive timer (+%ld ms)\n",
            tcp_conn-tcp_conn_table,
            tcp_conn->tc_ka_time*1000/HZ));
        clk_timer(&tcp_conn->tc_transmit_timer,
            curr_time+tcp_conn->tc_ka_time,
            tcp_send_timeout,
            tcp_conn-tcp_conn_table);

        return;
    }
    DBLOCK(0x10, printf(
"tcp_send_timeout: conn[%d]: triggering keep alive probe\n",
    tcp_conn-tcp_conn_table));
    tcp_conn->tc_ka_snd--;
    if (!(tcp_conn->tc_flags & TCF_FIN_SENT))
    {
        pkt= bf_memreq(1);
        *ptr2acc_data(pkt)= '\xff'; /* a random char */
        tcp_conn->tc_send_data= pkt; pkt= NULL;
    }
    tcp_conn->tc_SND_UNA--;
    if (tcp_conn->tc_SND_UNA == tcp_conn->tc_ISS)
    {
        /* We didn't send anything so far. Retrying the
         * SYN is too hard. Decrement ISS and hope
         * that the other side doesn't care.
         */
        tcp_conn->tc_ISS--;
    }
}

```

```

    /* Set tc_transmit_seq and tc_stt to trigger packet */
    tcp_conn->tc_transmit_seq= tcp_conn->tc_SND_UNA;
    tcp_conn->tc_stt= curr_time;

    /* Set tc_rt_seq for round trip measurements */
    tcp_conn->tc_rt_time= curr_time;
    tcp_conn->tc_rt_seq= tcp_conn->tc_SND_UNA;

    /* Set PSH to make sure that data gets sent */
    tcp_conn->tc_SND_PSH= tcp_conn->tc_SND_NXT;
    assert(tcp_check_conn(tcp_conn));

    /* Fall through */
}

rtt= tcp_conn->tc_rtt;

if (tcp_conn->tc_transmit_seq != tcp_conn->tc_SND_UNA)
{
    /* Some data has been acknowledged since the last time the
     * timer was set, set the timer again. */
    tcp_conn->tc_transmit_seq= tcp_conn->tc_SND_UNA;
    tcp_conn->tc_stt= 0;
    tcp_conn->tc_0wnd_to= 0;

    DBLOCK(0x20, printf(
"tcp_send_timeout: conn[%d] setting timer to %ld ms (+%ld ms)\n",
        tcp_conn-tcp_conn_table,
        (curr_time+rtt)*1000/HZ, rtt*1000/HZ));

    clk_timer(&tcp_conn->tc_transmit_timer,
        curr_time+rtt, tcp_send_timeout,
        tcp_conn-tcp_conn_table);

    return;
}

stt= tcp_conn->tc_stt;
if (stt == 0)
{
    /* Some packet arrived but did not acknowledge any data.
     * Apparently, the other side is still alive and has a
     * reason to transmit. We can assume a zero window.
     */

    DBLOCK(0x10, printf("conn[%d] setting zero window timer\n",
        tcp_conn-tcp_conn_table));

    if (tcp_conn->tc_0wnd_to < TCP_0WND_MIN)
        tcp_conn->tc_0wnd_to= TCP_0WND_MIN;
    else if (tcp_conn->tc_0wnd_to < rtt)
        tcp_conn->tc_0wnd_to= rtt;
    else
    {
        tcp_conn->tc_0wnd_to *= 2;
        if (tcp_conn->tc_0wnd_to > TCP_0WND_MAX)
            tcp_conn->tc_0wnd_to= TCP_0WND_MAX;
    }
    tcp_conn->tc_stt= curr_time;
    tcp_conn->tc_rt_seq= 0;

    DBLOCK(0x10, printf(
"tcp_send_timeout: conn[%d] setting timer to %ld ms (+%ld ms)\n",
        tcp_conn-tcp_conn_table,
        (curr_time+tcp_conn->tc_0wnd_to)*1000/HZ,
        tcp_conn->tc_0wnd_to*1000/HZ));

    clk_timer(&tcp_conn->tc_transmit_timer,
        curr_time+tcp_conn->tc_0wnd_to,
        tcp_send_timeout, tcp_conn-tcp_conn_table);

    return;
}
assert(stt <= curr_time);

```

```

DIFBLOCK(0x10, (tcp_conn->tc_fd == 0),
    printf("conn[%d] timeout in abandoned connection\n",
        tcp_conn-tcp_conn_table));

/* At this point, we have do a retransmission, or send a zero window
 * probe, which is almost the same.
 */

DBLOCK(0x20, printf("tcp_send_timeout: conn[%d] una= %lu, rtt= %ldms\n",
    tcp_conn-tcp_conn_table,
    (unsigned long)tcp_conn->tc_SND_UNA, rtt*1000/HZ));

/* Update threshold sequence number for retransmission calculation. */
if (tcp_Gmod4G(tcp_conn->tc_SND_TRM, tcp_conn->tc_rt_threshold))
    tcp_conn->tc_rt_threshold= tcp_conn->tc_SND_TRM;

tcp_conn->tc_SND_TRM= tcp_conn->tc_SND_UNA;

if (tcp_conn->tc_flags & TCF_PMTU &&
    curr_time > stt+TCP_PMTU_BLACKHOLE)
{
    /* We can't tell the difference between a PMTU blackhole
     * and a broken link. Assume a PMTU blackhole, and switch
     * off PMTU discovery.
     */
    DBLOCK(1, printf(
        "tcp[%d]: PMTU blackhole (or broken link) on route to ",
        tcp_conn-tcp_conn_table);
        writeIpAddr(tcp_conn->tc_remaddr);
        printf(", max mtu= %u\n", tcp_conn->tc_max_mtu));
    tcp_conn->tc_flags &= ~TCF_PMTU;
    tcp_conn->tc_mtutim= curr_time;
    if (tcp_conn->tc_max_mtu > IP_DEF_MTU)
        tcp_conn->tc_mtu= IP_DEF_MTU;
}

mss= tcp_conn->tc_mtu-IP_TCP_MIN_HDR_SIZE;
mss2= 2*mss;

if (tcp_conn->tc_snd_cwnd == tcp_conn->tc_SND_UNA)
    tcp_conn->tc_snd_cwnd++;
if (tcp_Gmod4G(tcp_conn->tc_snd_cwnd, tcp_conn->tc_SND_UNA + mss2))
{
    tcp_conn->tc_snd_cwnd= tcp_conn->tc_SND_UNA + mss2;
    if (tcp_Gmod4G(tcp_conn->tc_SND_TRM, tcp_conn->tc_snd_cwnd))
        tcp_conn->tc_SND_TRM= tcp_conn->tc_snd_cwnd;

    tcp_conn->tc_snd_cthresh /= 2;
    if (tcp_conn->tc_snd_cthresh < mss2)
        tcp_conn->tc_snd_cthresh= mss2;
}

if (curr_time-stt > tcp_conn->tc_rt_dead)
{
    tcp_close_connection(tcp_conn, ETIMEDOUT);
    return;
}

timeout= (curr_time-stt) >> 3;
if (timeout < rtt)
    timeout= rtt;
timeout += curr_time;

DBLOCK(0x20, printf(
    "tcp_send_timeout: conn[%d] setting timer to %ld ms (+%ld ms)\n",
    tcp_conn-tcp_conn_table, timeout*1000/HZ,
    (timeout-curr_time)*1000/HZ));

clk_timer(&tcp_conn->tc_transmit_timer, timeout,
    tcp_send_timeout, tcp_conn-tcp_conn_table);

#if 0
    if (tcp_conn->tc_rt_seq == 0)
    {

```

```

        printf("tcp_send_timeout: conn[%d]: setting tc_rt_time\n",
               tcp_conn-tcp_conn_table);
        tcp_conn->tc_rt_time= curr_time-rtt;
        tcp_conn->tc_rt_seq= tcp_conn->tc_SND_UNA;
    }
#endif

    if (tcp_conn->tc_state == TCS_SYN_SENT ||
        (curr_time-stt >= tcp_conn->tc_ttl*HZ))
    {
        new_ttl= tcp_conn->tc_ttl+1;
        if (new_ttl> IP_MAX_TTL)
            new_ttl= IP_MAX_TTL;
        tcp_conn->tc_ttl= new_ttl;
    }

    tcp_conn_write(tcp_conn, 0);
}

PUBLIC void tcp_fd_write(tcp_conn)
tcp_conn_t *tcp_conn;
{
    tcp_fd_t *tcp_fd;
    int urg, nourg, push;
    u32_t max_seq;
    size_t max_trans, write_count;
    acc_t *data, *send_data;

    assert(tcp_conn->tc_busy);
    tcp_fd= tcp_conn->tc_fd;

    if ((tcp_fd->tf_flags & TFF_IOCTL_IP) &&
        !(tcp_fd->tf_flags & TFF_WRITE_IP))
    {
        if (tcp_fd->tf_ioreq != NWIOTCPSHUTDOWN)
            return;
        DBLOCK(0x10, printf("NWIOTCPSHUTDOWN\n"));
        if (tcp_conn->tc_state == TCS_CLOSED)
        {
            tcp_reply_ioctl (tcp_fd, tcp_conn->tc_error);
            return;
        }
        if (!(tcp_conn->tc_flags & TCF_FIN_SENT))
        {
            DBLOCK(0x10, printf("calling tcp_shutdown\n"));
            tcp_shutdown (tcp_conn);
        }
        else
        {
            if (tcp_conn->tc_SND_UNA == tcp_conn->tc_SND_NXT)
            {
                tcp_reply_ioctl (tcp_fd, NW_OK);
                DBLOCK(0x10, printf("shutdown completed\n"));
            }
            else
            {
                DBLOCK(0x10,
                       printf("shutdown still inprogress\n"));
            }
        }
        return;
    }

    assert (tcp_fd->tf_flags & TFF_WRITE_IP);
    if (tcp_conn->tc_state == TCS_CLOSED)
    {
        if (tcp_fd->tf_write_offset)
        {
            tcp_reply_write(tcp_fd,
                           tcp_fd->tf_write_offset);
        }
        else
            tcp_reply_write(tcp_fd, tcp_conn->tc_error);
    }
}

```

```

        return;
    }

    urg= (tcp_fd->tf_flags & TFF_WR_URG);
    push= (tcp_fd->tf_flags & TFF_PUSH_DATA);

    max_seq= tcp_conn->tc_SND_UNA + TCP_MAX_SND_WND_SIZE;
    max_trans= max_seq - tcp_conn->tc_SND_NXT;
    if (tcp_fd->tf_write_count <= max_trans)
        write_count= tcp_fd->tf_write_count;
    else
        write_count= max_trans;
    if (write_count)
    {
        if (tcp_conn->tc_flags & TCF_BSD_URG)
        {
            if (tcp_Gmod4G(tcp_conn->tc_SND_NXT,
                tcp_conn->tc_SND_UNA))
            {
                nourg= tcp_LEmod4G(tcp_conn->tc_SND_UP,
                    tcp_conn->tc_SND_UNA);
                if ((urg && nourg) || (!urg && !nourg))
                {
                    DBLOCK(0x20,
                        printf("not sending\n"));
                    return;
                }
            }
        }
        data= (*tcp_fd->tf_get_userdata)
            (tcp_fd->tf_srfd, tcp_fd->tf_write_offset,
                write_count, FALSE);

        if (!data)
        {
            if (tcp_fd->tf_write_offset)
            {
                tcp_reply_write(tcp_fd,
                    tcp_fd->tf_write_offset);
            }
            else
                tcp_reply_write(tcp_fd, EFAULT);
            return;
        }
        tcp_fd->tf_write_offset += write_count;
        tcp_fd->tf_write_count -= write_count;

        send_data= tcp_conn->tc_send_data;
        tcp_conn->tc_send_data= 0;
        send_data= bf_append(send_data, data);
        tcp_conn->tc_send_data= send_data;
        tcp_conn->tc_SND_NXT += write_count;
        if (urg)
        {
            if (tcp_conn->tc_flags & TCF_BSD_URG)
                tcp_conn->tc_SND_UP= tcp_conn->tc_SND_NXT;
            else
                tcp_conn->tc_SND_UP= tcp_conn->tc_SND_NXT-1;
        }
        if (push && !tcp_fd->tf_write_count)
            tcp_conn->tc_SND_PSH= tcp_conn->tc_SND_NXT;
    }
    if (!tcp_fd->tf_write_count)
    {
        tcp_reply_write(tcp_fd, tcp_fd->tf_write_offset);
    }
}

PUBLIC unsigned tcp_sel_write(tcp_conn)
tcp_conn_t *tcp_conn;
{
    tcp_fd_t *tcp_fd;
    int urg, nourg;
    u32_t max_seq;

```

```

    size_t max_trans;

    tcp_fd= tcp_conn->tc_fd;

    if (tcp_conn->tc_state == TCS_CLOSED)
        return 1;

    urg= (tcp_fd->tf_flags & TFF_WR_URG);

    max_seq= tcp_conn->tc_SND_UNA + TCP_MAX_SND_WND_SIZE;
    max_trans= max_seq - tcp_conn->tc_SND_NXT;
    if (max_trans)
    {
        if (tcp_conn->tc_flags & TCF_BSD_URG)
        {
            if (tcp_Gmod4G(tcp_conn->tc_SND_NXT,
                           tcp_conn->tc_SND_UNA))
            {
                nourg= tcp_LEmod4G(tcp_conn->tc_SND_UP,
                                   tcp_conn->tc_SND_UNA);
                if ((urg && nourg) || (!urg && !nourg))
                {
                    DBLOCK(0x20,
                           printf("not sending\n"));
                    return 0;
                }
            }
        }
        return 1;
    }

    return 0;
}

PUBLIC void
tcp_rsel_write(tcp_conn)
tcp_conn_t *tcp_conn;
{
    tcp_fd_t *tcp_fd;

    if (tcp_sel_write(tcp_conn) == 0)
        return;

    tcp_fd= tcp_conn->tc_fd;
    tcp_fd->tf_flags &= ~TFF_SEL_WRITE;
    if (tcp_fd->tf_select_res)
        tcp_fd->tf_select_res(tcp_fd->tf_srfd, SR_SELECT_WRITE);
    else
        printf("tcp_rsel_write: no select_res\n");
}

/*
tcp_shutdown
*/

PUBLIC void tcp_shutdown(tcp_conn)
tcp_conn_t *tcp_conn;
{
    switch (tcp_conn->tc_state)
    {
        case TCS_CLOSED:
        case TCS_LISTEN:
        case TCS_SYN_SENT:
        case TCS_SYN_RECEIVED:
            tcp_close_connection(tcp_conn, ENOTCONN);
            return;
    }

    if (tcp_conn->tc_flags & TCF_FIN_SENT)
        return;
    tcp_conn->tc_flags |= TCF_FIN_SENT;
    tcp_conn->tc_flags &= ~TCF_NO_PUSH;
    tcp_conn->tc_SND_NXT++;
    tcp_conn->tc_SND_PSH= tcp_conn->tc_SND_NXT;
}

```

```

    assert (tcp_check_conn(tcp_conn) ||
            (tcp_print_conn(tcp_conn), printf("\n"), 0));

    tcp_conn_write(tcp_conn, 1);

    /* Start the timer */
    tcp_set_send_timer(tcp_conn);
}

PUBLIC void tcp_set_send_timer(tcp_conn)
tcp_conn_t *tcp_conn;
{
    clock_t curr_time;
    clock_t rtt;

    assert(tcp_conn->tc_state != TCS_CLOSED);
    assert(tcp_conn->tc_state != TCS_LISTEN);

    curr_time= get_time();
    rtt= tcp_conn->tc_rtt;

    DBLOCK(0x20, printf(
        "tcp_set_send_timer: conn[%d] setting timer to %ld ms (+%ld ms)\n",
            tcp_conn-tcp_conn_table,
            (curr_time+rtt)*1000/HZ, rtt*1000/HZ));

    /* Start the timer */
    clk_timer(&tcp_conn->tc_transmit_timer,
        curr_time+rtt, tcp_send_timeout, tcp_conn-tcp_conn_table);
    tcp_conn->tc_stt= curr_time;
}

/*
tcp_close_connection
*/

PUBLIC void tcp_close_connection(tcp_conn, error)
tcp_conn_t *tcp_conn;
int error;
{
    int i;
    tcp_port_t *tcp_port;
    tcp_fd_t *tcp_fd;
    tcp_conn_t *tc;

    assert (tcp_check_conn(tcp_conn) ||
            (tcp_print_conn(tcp_conn), printf("\n"), 0));
    assert (tcp_conn->tc_flags & TCF_INUSE);

    tcp_conn->tc_error= error;
    tcp_port= tcp_conn->tc_port;
    tcp_fd= tcp_conn->tc_fd;
    if (tcp_conn->tc_state == TCS_CLOSED)
        return;

    tcp_conn->tc_state= TCS_CLOSED;
    DBLOCK(0x10, tcp_print_state(tcp_conn); printf("\n"));

    if (tcp_fd && (tcp_fd->tf_flags & TFF_LISTENQ))
    {
        for (i= 0; i<TFL_LISTEN_MAX; i++)
        {
            if (tcp_fd->tf_listenq[i] == tcp_conn)
                break;
        }
        assert(i < TFL_LISTEN_MAX);
        tcp_fd->tf_listenq[i]= NULL;

        assert(tcp_conn->tc_connInprogress);
        tcp_conn->tc_connInprogress= 0;

        tcp_conn->tc_fd= NULL;

```

```
        tcp_fd= NULL;
    }
    else if (tcp_fd)
    {

        tcp_conn->tc_busy++;
        assert(tcp_fd->tf_conn == tcp_conn);

        if (tcp_fd->tf_flags & TFF_READ_IP)
            tcp_fd_read (tcp_conn, 1);
        assert (!(tcp_fd->tf_flags & TFF_READ_IP));
        if (tcp_fd->tf_flags & TFF_SEL_READ)
            tcp_rsel_read (tcp_conn);

        if (tcp_fd->tf_flags & TFF_WRITE_IP)
        {
            tcp_fd_write(tcp_conn);
            tcp_conn_write(tcp_conn, 1);
        }
        assert (!(tcp_fd->tf_flags & TFF_WRITE_IP));
        if (tcp_fd->tf_flags & TFF_IOCTL_IP)
        {
            tcp_fd_write(tcp_conn);
            tcp_conn_write(tcp_conn, 1);
        }
        if (tcp_fd->tf_flags & TFF_IOCTL_IP)
            assert(tcp_fd->tf_ioreq != NWIOTCPSHUTDOWN);
        if (tcp_fd->tf_flags & TFF_SEL_WRITE)
            tcp_rsel_write(tcp_conn);

        if (tcp_conn->tc_connInprogress)
            tcp_restart_connect(tcp_conn);
        assert (!tcp_conn->tc_connInprogress);
        assert (!(tcp_fd->tf_flags & TFF_IOCTL_IP) ||
            (printf("req=0x%x\n",
                (unsigned long)tcp_fd->tf_ioreq), 0));
        tcp_conn->tc_busy--;
    }

    if (tcp_conn->tc_rcvd_data)
    {
        bf_afree(tcp_conn->tc_rcvd_data);
        tcp_conn->tc_rcvd_data= NULL;
    }
    tcp_conn->tc_flags &= ~TCF_FIN_RECV;
    tcp_conn->tc_RCV_LO= tcp_conn->tc_RCV_NXT;

    if (tcp_conn->tc_adv_data)
    {
        bf_afree(tcp_conn->tc_adv_data);
        tcp_conn->tc_adv_data= NULL;
    }

    if (tcp_conn->tc_send_data)
    {
        bf_afree(tcp_conn->tc_send_data);
        tcp_conn->tc_send_data= NULL;
        tcp_conn->tc_SND_TRM=
            tcp_conn->tc_SND_NXT= tcp_conn->tc_SND_UNA;
    }
    tcp_conn->tc_SND_TRM= tcp_conn->tc_SND_NXT= tcp_conn->tc_SND_UNA;

    if (tcp_conn->tc_remipopt)
    {
        bf_afree(tcp_conn->tc_remipopt);
        tcp_conn->tc_remipopt= NULL;
    }

    if (tcp_conn->tc_tcpopt)
    {
        bf_afree(tcp_conn->tc_tcpopt);
        tcp_conn->tc_tcpopt= NULL;
    }
}
```



```
if (tcp_conn->tc_frag2send)
{
    bf_afree(tcp_conn->tc_frag2send);
    tcp_conn->tc_frag2send= NULL;
}
if (tcp_conn->tc_flags & TCF_MORE2WRITE)
{
    for (tc= tcp_port->tp_snd_head; tc; tc= tc->tc_send_link)
    {
        if (tc->tc_send_link == tcp_conn)
            break;
    }
    if (tc == NULL)
    {
        assert(tcp_port->tp_snd_head == tcp_conn);
        tcp_port->tp_snd_head= tcp_conn->tc_send_link;
    }
    else
    {
        tc->tc_send_link= tcp_conn->tc_send_link;
        if (tc->tc_send_link == NULL)
            tcp_port->tp_snd_tail= tc;
    }
    tcp_conn->tc_flags &= ~TCF_MORE2WRITE;
}

clk_untimer (&tcp_conn->tc_transmit_timer);
tcp_conn->tc_transmit_seq= 0;

/* clear all flags but TCF_INUSE */
tcp_conn->tc_flags &= TCF_INUSE;
assert (tcp_check_conn(tcp_conn));
}

/*
 * $PchId: tcp_send.c,v 1.32 2005/06/28 14:21:52 philip Exp $
 */
```

```
/*
type.h

Copyright 1995 Philip Homburg
*/

#ifndef INET_TYPE_H
#define INET_TYPE_H

typedef struct acc *(*get_userdata_t) ARGS(( int fd, size_t offset,
size_t count, int for_ioctl ));
typedef int (*put_userdata_t) ARGS(( int fd, size_t offset,
struct acc *data, int for_ioctl ));
typedef void (*put_pkt_t) ARGS(( int fd, struct acc *data, size_t datalen ));
typedef void (*select_res_t) ARGS(( int fd, unsigned ops ));

#endif /* INET_TYPE_H */

/*
* $PchId: type.h,v 1.6 2005/06/28 14:22:04 philip Exp $
*/
```

```
/*
udp.c

Copyright 1995 Philip Homburg
*/

#include "inet.h"
#include "type.h"

#include "assert.h"
#include "buf.h"
#include "clock.h"
#include "icmp_lib.h"
#include "io.h"
#include "ip.h"
#include "sr.h"
#include "udp.h"
#include "udp_int.h"

THIS_FILE

FORWARD void read_ip_packets ARGS(( udp_port_t *udp_port ));
FORWARD void udp_buffree ARGS(( int priority ));
#ifdef BUF_CONSISTENCY_CHECK
FORWARD void udp_bufcheck ARGS(( void ));
#endif
FORWARD void udp_main ARGS(( udp_port_t *udp_port ));
FORWARD int udp_select ARGS(( int fd, unsigned operations ));
FORWARD acc_t *udp_get_data ARGS(( int fd, size_t offset, size_t count,
int for_ioctl ));
FORWARD int udp_put_data ARGS(( int fd, size_t offset, acc_t *data,
int for_ioctl ));
FORWARD int udp_peek ARGS(( udp_fd_t * ));
FORWARD int udp_sel_read ARGS(( udp_fd_t * ));
FORWARD void udp_restart_write_port ARGS(( udp_port_t *udp_port ));
FORWARD void udp_ip_arrived ARGS(( int port, acc_t *pack, size_t pack_size ));
FORWARD void reply_thr_put ARGS(( udp_fd_t *udp_fd, int reply,
int for_ioctl ));
FORWARD void reply_thr_get ARGS(( udp_fd_t *udp_fd, int reply,
int for_ioctl ));
FORWARD int udp_setopt ARGS(( udp_fd_t *udp_fd ));
FORWARD udpport_t find_unused_port ARGS(( int fd ));
FORWARD int is_unused_port ARGS(( Udpport_t port ));
FORWARD int udp_packet2user ARGS(( udp_fd_t *udp_fd ));
FORWARD void restart_write_fd ARGS(( udp_fd_t *udp_fd ));
FORWARD ul6_t pack_oneCsum ARGS(( acc_t *pack ));
FORWARD void udp_rd_enqueue ARGS(( udp_fd_t *udp_fd, acc_t *pack,
clock_t exp_tim ));

FORWARD void hash_fd ARGS(( udp_fd_t *udp_fd ));
FORWARD void unhash_fd ARGS(( udp_fd_t *udp_fd ));

PUBLIC udp_port_t *udp_port_table;
PUBLIC udp_fd_t udp_fd_table[UDP_FD_NR];

PUBLIC void udp_prep()
{
    udp_port_table= alloc(udp_conf_nr * sizeof(udp_port_table[0]));
}

PUBLIC void udp_init()
{
    udp_fd_t *udp_fd;
    udp_port_t *udp_port;
    int i, j, ifno;

    assert (BUF_S >= sizeof(struct nwio_ipopt));
    assert (BUF_S >= sizeof(struct nwio_ipconf));
    assert (BUF_S >= sizeof(struct nwio_udpopt));
    assert (BUF_S >= sizeof(struct udp_io_hdr));
    assert (UDP_HDR_SIZE == sizeof(udp_hdr_t));
    assert (UDP_IO_HDR_SIZE == sizeof(udp_io_hdr_t));

    for (i= 0, udp_fd= udp_fd_table; i<UDP_FD_NR; i++, udp_fd++)
    {
```

```

        udp_fd->uf_flags= UFF_EMPTY;
        udp_fd->uf_rdbuf_head= NULL;
    }

#ifndef BUF_CONSISTENCY_CHECK
    bf_logon(udp_buffree);
#else
    bf_logon(udp_buffree, udp_bufcheck);
#endif

    for (i= 0, udp_port= udp_port_table; i<udp_conf_nr; i++, udp_port++)
    {
        udp_port->up_ipdev= udp_conf[i].uc_port;

        udp_port->up_flags= UPF_EMPTY;
        udp_port->up_state= UPS_EMPTY;
        udp_port->up_next_fd= udp_fd_table;
        udp_port->up_write_fd= NULL;
        udp_port->up_wr_pack= NULL;
        udp_port->up_port_any= NULL;
        for (j= 0; j<UDP_PORT_HASH_NR; j++)
            udp_port->up_port_hash[j]= NULL;

        ifno= ip_conf[udp_port->up_ipdev].ic_ifno;
        sr_add_minor(if2minor(ifno, UDP_DEV_OFF),
                    i, udp_open, udp_close, udp_read,
                    udp_write, udp_ioctl, udp_cancel, udp_select);

        udp_main(udp_port);
    }
}

PUBLIC int udp_open (port, srfd, get_userdata, put_userdata, put_pkt,
                    select_res)
int port;
int srfd;
get_userdata_t get_userdata;
put_userdata_t put_userdata;
put_pkt_t put_pkt;
select_res_t select_res;
{
    int i;
    udp_fd_t *udp_fd;

    for (i= 0; i<UDP_FD_NR && (udp_fd_table[i].uf_flags & UFF_INUSE);
        i++);

    if (i>= UDP_FD_NR)
    {
        DBLOCK(1, printf("out of fds\n"));
        return EAGAIN;
    }

    udp_fd= &udp_fd_table[i];

    udp_fd->uf_flags= UFF_INUSE;
    udp_fd->uf_port= &udp_port_table[port];
    udp_fd->uf_srfd= srfd;
    udp_fd->uf_udpopt.nwuo_flags= UDP_DEF_OPT;
    udp_fd->uf_get_userdata= get_userdata;
    udp_fd->uf_put_userdata= put_userdata;
    udp_fd->uf_select_res= select_res;
    assert(udp_fd->uf_rdbuf_head == NULL);
    udp_fd->uf_port_next= NULL;

    return i;
}

PUBLIC int udp_ioctl (fd, req)
int fd;
ioreq_t req;
{
    udp_fd_t *udp_fd;

```

```

    udp_port_t *udp_port;
    nwio_udpopt_t *udp_opt;
    acc_t *opt_acc;
    int result;

    udp_fd= &udp_fd_table[fd];
assert (udp_fd->uf_flags & UFF_INUSE);

    udp_port= udp_fd->uf_port;
    udp_fd->uf_flags |= UFF_IOCTL_IP;
    udp_fd->uf_ioreq= req;

    if (udp_port->up_state != UPS_MAIN)
        return NW_SUSPEND;

    switch(req)
    {
    case NWIOSUDPOPT:
        result= udp_setopt(udp_fd);
        break;
    case NWIOGUDPOPT:
        opt_acc= bf_memreq(sizeof(*udp_opt));
assert (opt_acc->acc_length == sizeof(*udp_opt));
        udp_opt= (nwio_udpopt_t *)ptr2acc_data(opt_acc);

        *udp_opt= udp_fd->uf_udpopt;
        udp_opt->nwuo_locaddr= udp_fd->uf_port->up_ipaddr;
        result= (*udp_fd->uf_put_userdata)(udp_fd->uf_srfd, 0, opt_acc,
            TRUE);
        if (result == NW_OK)
            reply_thr_put(udp_fd, NW_OK, TRUE);
        break;
    case NWIOUDPPEEK:
        result= udp_peek(udp_fd);
        break;
    default:
        reply_thr_get(udp_fd, EBADIOCTL, TRUE);
        result= NW_OK;
        break;
    }
    if (result != NW_SUSPEND)
        udp_fd->uf_flags &= ~UFF_IOCTL_IP;
    return result;
}

PUBLIC int udp_read (fd, count)
int fd;
size_t count;
{
    udp_fd_t *udp_fd;
    acc_t *tmp_acc, *next_acc;

    udp_fd= &udp_fd_table[fd];
    if (!(udp_fd->uf_flags & UFF_OPTSET))
    {
        reply_thr_put(udp_fd, EBADMODE, FALSE);
        return NW_OK;
    }

    udp_fd->uf_rd_count= count;

    if (udp_fd->uf_rdbuf_head)
    {
        if (get_time() <= udp_fd->uf_exp_tim)
            return udp_packet2user (udp_fd);
        tmp_acc= udp_fd->uf_rdbuf_head;
        while (tmp_acc)
        {
            next_acc= tmp_acc->acc_ext_link;
            bf_afree(tmp_acc);
            tmp_acc= next_acc;
        }
        udp_fd->uf_rdbuf_head= NULL;
    }
}

```

```

    }
    udp_fd->uf_flags |= UFF_READ_IP;
    return NW_SUSPEND;
}

PRIVATE void udp_main(udp_port)
udp_port_t *udp_port;
{
    udp_fd_t *udp_fd;
    int result, i;

    switch (udp_port->up_state)
    {
    case UPS_EMPTY:
        udp_port->up_state= UPS_SETPROTO;

        udp_port->up_ipfd= ip_open(udp_port->up_ipdev,
            udp_port->up_ipdev, udp_get_data, udp_put_data,
            udp_ip_arrived, 0 /* no select_res */);
        if (udp_port->up_ipfd < 0)
        {
            udp_port->up_state= UPS_ERROR;
            DBLOCK(1, printf("%s,%d: unable to open ip port\n",
                __FILE__, __LINE__));
            return;
        }

        result= ip_ioctl(udp_port->up_ipfd, NWIOSIPOPT);
        if (result == NW_SUSPEND)
            udp_port->up_flags |= UPF_SUSPEND;
        if (result<0)
        {
            return;
        }
        if (udp_port->up_state != UPS_GETCONF)
            return;
        /* drops through */
    case UPS_GETCONF:
        udp_port->up_flags &= ~UPF_SUSPEND;

        result= ip_ioctl(udp_port->up_ipfd, NWIOGIPCONF);
        if (result == NW_SUSPEND)
            udp_port->up_flags |= UPF_SUSPEND;
        if (result<0)
        {
            return;
        }
        if (udp_port->up_state != UPS_MAIN)
            return;
        /* drops through */
    case UPS_MAIN:
        udp_port->up_flags &= ~UPF_SUSPEND;

        for (i= 0, udp_fd= udp_fd_table; i<UDP_FD_NR; i++, udp_fd++)
        {
            if (!(udp_fd->uf_flags & UFF_INUSE))
                continue;
            if (udp_fd->uf_port != udp_port)
                continue;
            if (udp_fd->uf_flags & UFF_IOCTL_IP)
                udp_ioctl(i, udp_fd->uf_ioreq);
        }
        read_ip_packets(udp_port);
        return;
    default:
        DBLOCK(1, printf("udp_port_table[%d].up_state= %d\n",
            udp_port->up_ipdev, udp_port->up_state));
        ip_panic(("unknown state" ));
        break;
    }
}

PRIVATE int udp_select(fd, operations)
int fd;

```

```

unsigned operations;
{
    int i;
    unsigned resops;
    udp_fd_t *udp_fd;

    udp_fd= &udp_fd_table[fd];
    assert (udp_fd->uf_flags & UFF_INUSE);

    resops= 0;

    if (operations & SR_SELECT_READ)
    {
        if (udp_sel_read(udp_fd))
            resops |= SR_SELECT_READ;
        else if (!(operations & SR_SELECT_POLL))
            udp_fd->uf_flags |= UFF_SEL_READ;
    }
    if (operations & SR_SELECT_WRITE)
    {
        /* Should handle special case when the interface is down */
        resops |= SR_SELECT_WRITE;
    }
    if (operations & SR_SELECT_EXCEPTION)
    {
        printf("udp_select: not implemented for exceptions\n");
    }
    return resops;
}

PRIVATE acc_t *udp_get_data (port, offset, count, for_ioctl)
int port;
size_t offset;
size_t count;
int for_ioctl;
{
    udp_port_t *udp_port;
    udp_fd_t *udp_fd;
    int result;

    udp_port= &udp_port_table[port];

    switch(udp_port->up_state)
    {
        case UPS_SETPROTO:
            assert (for_ioctl);
            if (!count)
            {
                result= (int)offset;
                if (result<0)
                {
                    udp_port->up_state= UPS_ERROR;
                    break;
                }
                udp_port->up_state= UPS_GETCONF;
                if (udp_port->up_flags & UPF_SUSPEND)
                    udp_main(udp_port);
                return NULL;
            }
            else
            {
                struct nwio_ipopt *ipopt;
                acc_t *acc;

                assert (!offset);
                assert (count == sizeof(*ipopt));

                acc= bf_memreq(sizeof(*ipopt));
                ipopt= (struct nwio_ipopt *)ptr2acc_data(acc);
                ipopt->nwio_flags= NWIO_COPY | NWIO_EN_LOC |
                    NWIO_EN_BROAD | NWIO_REMANY | NWIO_PROTOSPEC |
                    NWIO_HDR_O_ANY | NWIO_RWDATALL;
                ipopt->nwio_proto= IPPROTO_UDP;
                return acc;
            }
        }
    }

```

```

    }
    case UPS_MAIN:
assert (!for_ioctl);
assert (udp_port->up_flags & UPF_WRITE_IP);
        if (!count)
        {
            result= (int)offset;
assert (udp_port->up_wr_pack);
            bf_afree(udp_port->up_wr_pack);
            udp_port->up_wr_pack= 0;
            if (udp_port->up_flags & UPF_WRITE_SP)
            {
                if (udp_port->up_write_fd)
                {
                    udp_fd= udp_port->up_write_fd;
                    udp_port->up_write_fd= NULL;
                    udp_fd->uf_flags &= ~UFF_WRITE_IP;
                    reply_thr_get(udp_fd, result, FALSE);
                }
                udp_port->up_flags &= ~(UPF_WRITE_SP |
                    UPF_WRITE_IP);
                if (udp_port->up_flags & UPF_MORE2WRITE)
                {
                    udp_restart_write_port(udp_port);
                }
            }
            else
                udp_port->up_flags &= ~UPF_WRITE_IP;
        }
    else
    {
        return bf_cut (udp_port->up_wr_pack, offset, count);
    }
    break;
default:
    printf("udp_get_data(%d, 0x%x, 0x%x) called but up_state= 0x%x\n",
        port, offset, count, udp_port->up_state);
    break;
}
return NULL;
}

PRIVATE int udp_put_data (fd, offset, data, for_ioctl)
int fd;
size_t offset;
acc_t *data;
int for_ioctl;
{
    udp_port_t *udp_port;
    int result;

    udp_port= &udp_port_table[fd];

    switch (udp_port->up_state)
    {
    case UPS_GETCONF:
        if (!data)
        {
            result= (int)offset;
            if (result<0)
            {
                udp_port->up_state= UPS_ERROR;
                return NW_OK;
            }
            udp_port->up_state= UPS_MAIN;
            if (udp_port->up_flags & UPF_SUSPEND)
                udp_main(udp_port);
        }
    else
    {
        struct nwio_ipconf *ipconf;

        data= bf_packIfLess(data, sizeof(*ipconf));
        ipconf= (struct nwio_ipconf *)ptr2acc_data(data);
    }
}

```



```

assert (ipconf->nwic_flags & NWIC_IPADDR_SET);
        udp_port->up_ipaddr= ipconf->nwic_ipaddr;
        bf_afree(data);
    }
    break;
case UPS_MAIN:
    assert(0);

    assert (udp_port->up_flags & UPF_READ_IP);
    if (!data)
    {
        result= (int)offset;
        compare (result, >=, 0);
        if (udp_port->up_flags & UPF_READ_SP)
        {
            udp_port->up_flags &= ~(UPF_READ_SP|
                                    UPF_READ_IP);
            read_ip_packets(udp_port);
        }
        else
            udp_port->up_flags &= ~UPF_READ_IP;
    }
    else
    {
assert (!offset);        /* This isn't a valid assertion but ip sends only
                           * whole datagrams up */
        udp_ip_arrived(fd, data, bf_bufsize(data));
    }
    break;
default:
    ip_panic((
        "udp_put_data(%d, 0x%x, %p) called but up_state= 0x%x\n",
                                fd, offset, data, udp_port->up_state ));
    }
    return NW_OK;
}

PRIVATE int udp_setopt(udp_fd)
udp_fd_t *udp_fd;
{
    udp_fd_t *fd_ptr;
    nwio_udpopt_t oldopt, newopt;
    acc_t *data;
    unsigned int new_en_flags, new_di_flags, old_en_flags, old_di_flags,
        all_flags, flags;
    unsigned long new_flags;
    int i;

    data= (*udp_fd->uf_get_userdata)(udp_fd->uf_srfd, 0,
        sizeof(nwio_udpopt_t), TRUE);

    if (!data)
        return EFAULT;

    data= bf_packIfLess(data, sizeof(nwio_udpopt_t));
assert (data->acc_length == sizeof(nwio_udpopt_t));

    newopt= *(nwio_udpopt_t *)ptr2acc_data(data);
    bf_afree(data);
    oldopt= udp_fd->uf_udpopt;

    old_en_flags= oldopt.nwuo_flags & 0xffff;
    old_di_flags= (oldopt.nwuo_flags >> 16) & 0xffff;

    new_en_flags= newopt.nwuo_flags & 0xffff;
    new_di_flags= (newopt.nwuo_flags >> 16) & 0xffff;

    if (new_en_flags & new_di_flags)
    {
        DBLOCK(1, printf("returning EBADMODE\n"));

        reply_thr_get(udp_fd, EBADMODE, TRUE);
        return NW_OK;
    }
}

```

```
/* NWUO_ACC_MASK */
if (new_di_flags & NWUO_ACC_MASK)
{
    DBLOCK(1, printf("returning EBADMODE\n"));

    reply_thr_get(udp_fd, EBADMODE, TRUE);
    return NW_OK;
    /* access modes can't be disabled */
}

if (!(new_en_flags & NWUO_ACC_MASK))
    new_en_flags |= (old_en_flags & NWUO_ACC_MASK);

/* NWUO_LOCPORT_MASK */
if (new_di_flags & NWUO_LOCPORT_MASK)
{
    DBLOCK(1, printf("returning EBADMODE\n"));

    reply_thr_get(udp_fd, EBADMODE, TRUE);
    return NW_OK;
    /* the loc ports can't be disabled */
}

if (!(new_en_flags & NWUO_LOCPORT_MASK))
{
    new_en_flags |= (old_en_flags & NWUO_LOCPORT_MASK);
    newopt.nwuo_locport= oldopt.nwuo_locport;
}
else if ((new_en_flags & NWUO_LOCPORT_MASK) == NWUO_LP_SEL)
{
    newopt.nwuo_locport= find_unused_port(udp_fd-udp_fd_table);
}
else if ((new_en_flags & NWUO_LOCPORT_MASK) == NWUO_LP_SET)
{
    if (!newopt.nwuo_locport)
    {
        DBLOCK(1, printf("returning EBADMODE\n"));

        reply_thr_get(udp_fd, EBADMODE, TRUE);
        return NW_OK;
    }
}

/* NWUO_LOCADDR_MASK */
if (!((new_en_flags | new_di_flags) & NWUO_LOCADDR_MASK))
{
    new_en_flags |= (old_en_flags & NWUO_LOCADDR_MASK);
    new_di_flags |= (old_di_flags & NWUO_LOCADDR_MASK);
}

/* NWUO_BROAD_MASK */
if (!((new_en_flags | new_di_flags) & NWUO_BROAD_MASK))
{
    new_en_flags |= (old_en_flags & NWUO_BROAD_MASK);
    new_di_flags |= (old_di_flags & NWUO_BROAD_MASK);
}

/* NWUO_REMPORT_MASK */
if (!((new_en_flags | new_di_flags) & NWUO_REMPORT_MASK))
{
    new_en_flags |= (old_en_flags & NWUO_REMPORT_MASK);
    new_di_flags |= (old_di_flags & NWUO_REMPORT_MASK);
    newopt.nwuo_rempport= oldopt.nwuo_rempport;
}

/* NWUO_REMADDR_MASK */
if (!((new_en_flags | new_di_flags) & NWUO_REMADDR_MASK))
{
    new_en_flags |= (old_en_flags & NWUO_REMADDR_MASK);
    new_di_flags |= (old_di_flags & NWUO_REMADDR_MASK);
    newopt.nwuo_remaddr= oldopt.nwuo_remaddr;
}

/* NWUO_RW_MASK */
```

```

if (!((new_en_flags | new_di_flags) & NWUO_RW_MASK))
{
    new_en_flags |= (old_en_flags & NWUO_RW_MASK);
    new_di_flags |= (old_di_flags & NWUO_RW_MASK);
}

/* NWUO_IPOPT_MASK */
if (!((new_en_flags | new_di_flags) & NWUO_IPOPT_MASK))
{
    new_en_flags |= (old_en_flags & NWUO_IPOPT_MASK);
    new_di_flags |= (old_di_flags & NWUO_IPOPT_MASK);
}

new_flags= ((unsigned long)new_di_flags << 16) | new_en_flags;
if ((new_flags & NWUO_RWDATONLY) &&
    ((new_flags & NWUO_LOCPORT_MASK) == NWUO_LP_ANY ||
    (new_flags & (NWUO_RP_ANY|NWUO_RA_ANY|NWUO_EN_IPOPT))))
{
    DBLOCK(1, printf("returning EBADMODE\n"));

    reply_thr_get(udp_fd, EBADMODE, TRUE);
    return NW_OK;
}

/* Check the access modes */
if ((new_flags & NWUO_LOCPORT_MASK) == NWUO_LP_SEL ||
    (new_flags & NWUO_LOCPORT_MASK) == NWUO_LP_SET)
{
    for (i= 0, fd_ptr= udp_fd_table; i<UDP_FD_NR; i++, fd_ptr++)
    {
        if (fd_ptr == udp_fd)
            continue;
        if (!(fd_ptr->uf_flags & UFF_INUSE))
            continue;
        if (fd_ptr->uf_port != udp_fd->uf_port)
            continue;
        flags= fd_ptr->uf_udpopt.nwuo_flags;
        if ((flags & NWUO_LOCPORT_MASK) != NWUO_LP_SEL &&
            (flags & NWUO_LOCPORT_MASK) != NWUO_LP_SET)
            continue;
        if (fd_ptr->uf_udpopt.nwuo_locport !=
            newopt.nwuo_locport)
        {
            continue;
        }
        if ((flags & NWUO_ACC_MASK) !=
            (new_flags & NWUO_ACC_MASK))
        {
            DBLOCK(1, printf(
                "address inuse: new fd= %d, old fd= %d, port= %u\n",
                udp_fd-udp_fd_table,
                fd_ptr-udp_fd_table,
                newopt.nwuo_locport));

            reply_thr_get(udp_fd, EADDRINUSE, TRUE);
            return NW_OK;
        }
    }
}

if (udp_fd->uf_flags & UFF_OPTSET)
    unhash_fd(udp_fd);

newopt.nwuo_flags= new_flags;
udp_fd->uf_udpopt= newopt;

all_flags= new_en_flags | new_di_flags;
if ((all_flags & NWUO_ACC_MASK) && (all_flags & NWUO_LOCPORT_MASK) &&
    (all_flags & NWUO_LOCADDR_MASK) &&
    (all_flags & NWUO_BROAD_MASK) &&
    (all_flags & NWUO_REMPORT_MASK) &&
    (all_flags & NWUO_REMADDR_MASK) &&
    (all_flags & NWUO_RW_MASK) &&
    (all_flags & NWUO_IPOPT_MASK))

```

```

        udp_fd->uf_flags |= UFF_OPTSET;
    else
    {
        udp_fd->uf_flags &= ~UFF_OPTSET;
    }

    if (udp_fd->uf_flags & UFF_OPTSET)
        hash_fd(udp_fd);

    reply_thr_get(udp_fd, NW_OK, TRUE);
    return NW_OK;
}

PRIVATE udpport_t find_unused_port(fd)
int fd;
{
    udpport_t port, nw_port;

    for (port= 0x8000+fd; port < 0xffff-UDP_FD_NR; port+= UDP_FD_NR)
    {
        nw_port= htons(port);
        if (is_unused_port(nw_port))
            return nw_port;
    }
    for (port= 0x8000; port < 0xffff; port++)
    {
        nw_port= htons(port);
        if (is_unused_port(nw_port))
            return nw_port;
    }
    ip_panic(( "unable to find unused port (shouldn't occur)" ));
    return 0;
}

/*
reply_thr_put
*/

PRIVATE void reply_thr_put(udp_fd, reply, for_ioctl)
udp_fd_t *udp_fd;
int reply;
int for_ioctl;
{
    int result;

    result= (*udp_fd->uf_put_userdata)(udp_fd->uf_srfd, reply,
        (acc_t *)0, for_ioctl);
    assert(result == NW_OK);
}

/*
reply_thr_get
*/

PRIVATE void reply_thr_get(udp_fd, reply, for_ioctl)
udp_fd_t *udp_fd;
int reply;
int for_ioctl;
{
    acc_t *result;
    result= (*udp_fd->uf_get_userdata)(udp_fd->uf_srfd, reply,
        (size_t)0, for_ioctl);
    assert (!result);
}

PRIVATE int is_unused_port(port)
udpport_t port;
{
    int i;
    udp_fd_t *udp_fd;

    for (i= 0, udp_fd= udp_fd_table; i<UDP_FD_NR; i++,
        udp_fd++)
    {

```

```

        if (!(udp_fd->uf_flags & UFF_OPTSET))
            continue;
        if (udp_fd->uf_udpopt.nwuo_locport == port)
            return FALSE;
    }
    return TRUE;
}

PRIVATE void read_ip_packets(udp_port)
udp_port_t *udp_port;
{
    int result;

    do
    {
        udp_port->up_flags |= UPF_READ_IP;
        result= ip_read(udp_port->up_ipfd, UDP_MAX_DATAGRAM);
        if (result == NW_SUSPEND)
        {
            udp_port->up_flags |= UPF_READ_SP;
            return;
        }
    } while(result == NW_OK);
    udp_port->up_flags &= ~UPF_READ_IP;
    } while(!(udp_port->up_flags & UPF_READ_IP));
}

PRIVATE int udp_peek (udp_fd)
udp_fd_t *udp_fd;
{
    acc_t *pack, *tmp_acc, *next_acc;
    int result;

    if (!(udp_fd->uf_flags & UFF_OPTSET))
    {
        udp_fd->uf_flags &= ~UFF_IOCTL_IP;
        reply_thr_put(udp_fd, EBADMODE, TRUE);
        return NW_OK;
    }

    if (udp_fd->uf_rdbuf_head)
    {
        if (get_time() <= udp_fd->uf_exp_tim)
        {
            pack= bf_cut(udp_fd->uf_rdbuf_head, 0,
                sizeof(udp_io_hdr_t));
            result= (*udp_fd->uf_put_userdata)(udp_fd->uf_srfd,
                (size_t)0, pack, TRUE);

            udp_fd->uf_flags &= ~UFF_IOCTL_IP;
            result= (*udp_fd->uf_put_userdata)(udp_fd->uf_srfd,
                result, (acc_t *)0, TRUE);
            assert (result == 0);
            return result;
        }
        tmp_acc= udp_fd->uf_rdbuf_head;
        while (tmp_acc)
        {
            next_acc= tmp_acc->acc_ext_link;
            bf_afree(tmp_acc);
            tmp_acc= next_acc;
        }
        udp_fd->uf_rdbuf_head= NULL;
    }
    udp_fd->uf_flags |= UFF_PEEK_IP;
    return NW_SUSPEND;
}

PRIVATE int udp_sel_read (udp_fd)
udp_fd_t *udp_fd;
{
    acc_t *pack, *tmp_acc, *next_acc;
    int result;

```

```

    if (!(udp_fd->uf_flags & UFF_OPTSET))
        return 1; /* Read will not block */

    if (udp_fd->uf_rdbuf_head)
    {
        if (get_time() <= udp_fd->uf_exp_tim)
            return 1;

        tmp_acc= udp_fd->uf_rdbuf_head;
        while (tmp_acc)
        {
            next_acc= tmp_acc->acc_ext_link;
            bf_afree(tmp_acc);
            tmp_acc= next_acc;
        }
        udp_fd->uf_rdbuf_head= NULL;
    }
    return 0;
}

PRIVATE int udp_packet2user (udp_fd)
udp_fd_t *udp_fd;
{
    acc_t *pack, *tmp_pack;
    udp_io_hdr_t *hdr;
    int result, hdr_len;
    size_t size, transf_size;

    pack= udp_fd->uf_rdbuf_head;
    udp_fd->uf_rdbuf_head= pack->acc_ext_link;

    size= bf_bufsize (pack);

    if (udp_fd->uf_udpopt.nwuo_flags & NWUO_RWDATONLY)
    {
        pack= bf_packIffLess (pack, UDP_IO_HDR_SIZE);
        assert (pack->acc_length >= UDP_IO_HDR_SIZE);

        hdr= (udp_io_hdr_t *)ptr2acc_data(pack);
#if CONF_UDP_IO_NW_BYTE_ORDER
        hdr_len= UDP_IO_HDR_SIZE+NTOHS(hdr->uih_ip_opt_len);
#else
        hdr_len= UDP_IO_HDR_SIZE+hdr->uih_ip_opt_len;
#endif

        assert (size>= hdr_len);
        size -= hdr_len;
        tmp_pack= bf_cut(pack, hdr_len, size);
        bf_afree(pack);
        pack= tmp_pack;
    }

    if (size>udp_fd->uf_rd_count)
    {
        tmp_pack= bf_cut (pack, 0, udp_fd->uf_rd_count);
        bf_afree(pack);
        pack= tmp_pack;
        transf_size= udp_fd->uf_rd_count;
    }
    else
        transf_size= size;

    result= (*udp_fd->uf_put_userdata)(udp_fd->uf_srfd,
        (size_t)0, pack, FALSE);

    if (result >= 0)
        if (size > transf_size)
            result= EPACKSIZE;
        else
            result= transf_size;

    udp_fd->uf_flags &= ~UFF_READ_IP;

```

```

    result= (*udp_fd->uf_put_userdata)(udp_fd->uf_srfd, result,
                                       (acc_t *)0, FALSE);
    assert (result == 0);

    return result;
}

PRIVATE void udp_ip_arrived(port, pack, pack_size)
int port;
acc_t *pack;
size_t pack_size;
{
    udp_port_t *udp_port;
    udp_fd_t *udp_fd, *share_fd;
    acc_t *ip_hdr_acc, *udp_acc, *ipopt_pack, *no_ipopt_pack, *tmp_acc;
    ip_hdr_t *ip_hdr;
    udp_hdr_t *udp_hdr;
    udp_io_hdr_t *udp_io_hdr;
    size_t ip_hdr_size, udp_size, data_size, opt_size;
    ipaddr_t src_addr, dst_addr, ipaddr;
    udpport_t src_port, dst_port;
    u8_t ul6[2];
    ul6_t chksum;
    unsigned long dst_type, flags;
    clock_t exp_tim;
    int i, delivered, hash;

    udp_port= &udp_port_table[port];

    ip_hdr_acc= bf_cut(pack, 0, IP_MIN_HDR_SIZE);
    ip_hdr_acc= bf_packIfLess(ip_hdr_acc, IP_MIN_HDR_SIZE);
    ip_hdr= (ip_hdr_t *)ptr2acc_data(ip_hdr_acc);
    ip_hdr_size= (ip_hdr->ih_vers_ihl & IH_IHL_MASK) << 2;
    if (ip_hdr_size != IP_MIN_HDR_SIZE)
    {
        bf_afree(ip_hdr_acc);
        ip_hdr_acc= bf_cut(pack, 0, ip_hdr_size);
        ip_hdr_acc= bf_packIfLess(ip_hdr_acc, ip_hdr_size);
        ip_hdr= (ip_hdr_t *)ptr2acc_data(ip_hdr_acc);
    }

    pack_size -= ip_hdr_size;
    if (pack_size < UDP_HDR_SIZE)
    {
        if (pack_size == 0 && ip_hdr->ih_proto == 0)
        {
            /* IP layer reports new IP address */
            ipaddr= ip_hdr->ih_src;
            udp_port->up_ipaddr= ipaddr;
            DBLOCK(1, printf("udp_ip_arrived: using address ");
                      writeIpAddr(ipaddr); printf("\n"));
        }
        else
            DBLOCK(1, printf("packet too small\n"));

        bf_afree(ip_hdr_acc);
        bf_afree(pack);
        return;
    }

    udp_acc= bf_delhead(pack, ip_hdr_size);
    pack= NULL;

    udp_acc= bf_packIfLess(udp_acc, UDP_HDR_SIZE);
    udp_hdr= (udp_hdr_t *)ptr2acc_data(udp_acc);
    udp_size= ntohs(udp_hdr->uh_length);
    if (udp_size > pack_size)
    {
        DBLOCK(1, printf("packet too large\n"));

        bf_afree(ip_hdr_acc);
        bf_afree(udp_acc);
        return;
    }

```

```

    }

    src_addr= ip_hdr->ih_src;
    dst_addr= ip_hdr->ih_dst;

    if (udp_hdr->uh_chksum)
    {
        ul6[0]= 0;
        ul6[1]= ip_hdr->ih_proto;
        chksum= pack_oneCsum(udp_acc);
        chksum= oneCsum(chksum, (ul6_t *)&src_addr, sizeof(ipaddr_t));
        chksum= oneCsum(chksum, (ul6_t *)&dst_addr, sizeof(ipaddr_t));
        chksum= oneCsum(chksum, (ul6_t *)ul6, sizeof(ul6));
        chksum= oneCsum(chksum, (ul6_t *)&udp_hdr->uh_length,
                        sizeof(udp_hdr->uh_length));
        if (~chksum & 0xffff)
        {
            DBLOCK(1, printf("checksum error in udp packet\n");
                    printf("src ip_addr= ");
                    writeIpAddr(src_addr);
                    printf(" dst ip_addr= ");
                    writeIpAddr(dst_addr);
                    printf("\n");
                    printf("packet chksum= 0x%x, sum= 0x%x\n",
                            udp_hdr->uh_chksum, chksum));

            bf_afree(ip_hdr_acc);
            bf_afree(udp_acc);
            return;
        }
    }

    exp_tim= get_time() + UDP_READ_EXP_TIME;
    src_port= udp_hdr->uh_src_port;
    dst_port= udp_hdr->uh_dst_port;

    /* Send an ICMP port unreachable if the packet could not be
     * delivered.
     */
    delivered= 0;

    if (dst_addr == udp_port->up_ipaddr)
        dst_type= NWUO_EN_LOC;
    else
    {
        dst_type= NWUO_EN_BROAD;

        /* Don't send ICMP error packets for broadcast packets */
        delivered= 1;
    }

    DBLOCK(0x20, printf("udp: got packet from ");
            writeIpAddr(src_addr);
            printf(":%u to ", ntohs(src_port));
            writeIpAddr(dst_addr);
            printf(":%u\n", ntohs(dst_port)));

    no_ipopt_pack= bf_memreq(UDP_IO_HDR_SIZE);
    udp_io_hdr= (udp_io_hdr_t *)ptr2acc_data(no_ipopt_pack);
    udp_io_hdr->uih_src_addr= src_addr;
    udp_io_hdr->uih_dst_addr= dst_addr;
    udp_io_hdr->uih_src_port= src_port;
    udp_io_hdr->uih_dst_port= dst_port;
    data_size = udp_size-UDP_HDR_SIZE;
#if CONF_UDP_IO_NW_BYTE_ORDER
    udp_io_hdr->uih_ip_opt_len= HTONS(0);
    udp_io_hdr->uih_data_len= htons(data_size);
#else
    udp_io_hdr->uih_ip_opt_len= 0;
    udp_io_hdr->uih_data_len= data_size;
#endif
    no_ipopt_pack->acc_next= bf_cut(udp_acc, UDP_HDR_SIZE, data_size);

    if (ip_hdr_size == IP_MIN_HDR_SIZE)

```



```

    {
        ipopt_pack= no_ipopt_pack;
        ipopt_pack->acc_linkC++;
    }
    else
    {
        ipopt_pack= bf_memreq(UDP_IO_HDR_SIZE);
        *(udp_io_hdr_t *)ptr2acc_data(ipopt_pack)= *udp_io_hdr;
        udp_io_hdr= (udp_io_hdr_t *)ptr2acc_data(ipopt_pack);
        opt_size = ip_hdr_size-IP_MIN_HDR_SIZE;
#if CONF_UDP_IO_NW_BYTE_ORDER
        udp_io_hdr->uih_ip_opt_len= htons(opt_size);
#else
        udp_io_hdr->uih_ip_opt_len= opt_size;
#endif

        tmp_acc= bf_cut(ip_hdr_acc, (size_t)IP_MIN_HDR_SIZE, opt_size);
        assert(tmp_acc->acc_linkC == 1);
        assert(tmp_acc->acc_next == NULL);
        ipopt_pack->acc_next= tmp_acc;

        tmp_acc->acc_next= no_ipopt_pack->acc_next;
        if (tmp_acc->acc_next)
            tmp_acc->acc_next->acc_linkC++;
    }

    hash= dst_port;
    hash ^= (hash >> 8);
    hash &= (UDP_PORT_HASH_NR-1);

    for (i= 0; i<2; i++)
    {
        share_fd= NULL;

        udp_fd= (i == 0) ? udp_port->up_port_any :
            udp_port->up_port_hash[hash];
        for (; udp_fd; udp_fd= udp_fd->uf_port_next)
        {
            if (i && udp_fd->uf_udpopt.nwuo_locport != dst_port)
                continue;

            assert(udp_fd->uf_flags & UFF_INUSE);
            assert(udp_fd->uf_flags & UFF_OPTSET);

            if (udp_fd->uf_port != udp_port)
                continue;

            flags= udp_fd->uf_udpopt.nwuo_flags;
            if (!(flags & dst_type))
                continue;

            if ((flags & NWUO_RP_SET) &&
                udp_fd->uf_udpopt.nwuo_remport != src_port)
            {
                continue;
            }

            if ((flags & NWUO_RA_SET) &&
                udp_fd->uf_udpopt.nwuo_remaddr != src_addr)
            {
                continue;
            }

            if (i)
            {
                /* Packet is considered to be delivered */
                delivered= 1;
            }

            if ((flags & NWUO_ACC_MASK) == NWUO_SHARED &&
                (!share_fd || !udp_fd->uf_rdbuf_head))
            {
                share_fd= udp_fd;
                continue;
            }
        }
    }

```

```

        if (flags & NWUO_EN_IPOPT)
            pack= ipopt_pack;
        else
            pack= no_ipopt_pack;

        pack->acc_linkC++;
        udp_rd_enqueue(udp_fd, pack, exp_tim);
        if (udp_fd->uf_flags & UFF_READ_IP)
            udp_packet2user(udp_fd);
    }

    if (share_fd)
    {
        flags= share_fd->uf_udpopt.nwuo_flags;
        if (flags & NWUO_EN_IPOPT)
            pack= ipopt_pack;
        else
            pack= no_ipopt_pack;

        pack->acc_linkC++;
        udp_rd_enqueue(share_fd, pack, exp_tim);
        if (share_fd->uf_flags & UFF_READ_IP)
            udp_packet2user(share_fd);
    }
}

if (ipopt_pack)
    bf_afree(ipopt_pack);
if (no_ipopt_pack)
    bf_afree(no_ipopt_pack);

if (!delivered)
{
    DBLOCK(0x2, printf("udp: could not deliver packet from ");
            writeIpAddr(src_addr);
            printf("%.u to ", ntohs(src_port));
            writeIpAddr(dst_addr);
            printf("%.u\n", ntohs(dst_port)));

    pack= bf_append(ip_hdr_acc, udp_acc);
    ip_hdr_acc= NULL;
    udp_acc= NULL;
    icmp_snd_unreachable(udp_port->up_ipdev, pack,
                        ICMP_PORT_UNRCH);
    return;
}

assert (ip_hdr_acc);
bf_afree(ip_hdr_acc);
assert (udp_acc);
bf_afree(udp_acc);
}

PUBLIC void udp_close(fd)
int fd;
{
    udp_fd_t *udp_fd;
    acc_t *tmp_acc, *next_acc;

    udp_fd= &udp_fd_table[fd];

    assert (udp_fd->uf_flags & UFF_INUSE);

    if (udp_fd->uf_flags & UFF_OPTSET)
        unhash_fd(udp_fd);

    udp_fd->uf_flags= UFF_EMPTY;
    tmp_acc= udp_fd->uf_rdbuf_head;
    while (tmp_acc)
    {
        next_acc= tmp_acc->acc_ext_link;
        bf_afree(tmp_acc);
        tmp_acc= next_acc;
    }
}

```

```
    }
    udp_fd->uf_rdbuf_head= NULL;
}

PUBLIC int udp_write(fd, count)
int fd;
size_t count;
{
    udp_fd_t *udp_fd;
    udp_port_t *udp_port;

    udp_fd= &udp_fd_table[fd];
    udp_port= udp_fd->uf_port;

    if (!(udp_fd->uf_flags & UFF_OPTSET))
    {
        reply_thr_get (udp_fd, EBADMODE, FALSE);
        return NW_OK;
    }

    assert (!(udp_fd->uf_flags & UFF_WRITE_IP));

    udp_fd->uf_wr_count= count;

    udp_fd->uf_flags |= UFF_WRITE_IP;

    restart_write_fd(udp_fd);

    if (udp_fd->uf_flags & UFF_WRITE_IP)
    {
        DBLOCK(1, printf("replying NW_SUSPEND\n"));

        return NW_SUSPEND;
    }
    else
    {
        return NW_OK;
    }
}

PRIVATE void restart_write_fd(udp_fd)
udp_fd_t *udp_fd;
{
    udp_port_t *udp_port;
    acc_t *pack, *ip_hdr_pack, *udp_hdr_pack, *ip_opt_pack, *user_data;
    udp_hdr_t *udp_hdr;
    udp_io_hdr_t *udp_io_hdr;
    ip_hdr_t *ip_hdr;
    size_t ip_opt_size, user_data_size;
    unsigned long flags;
    ul6_t chksum;
    u8_t ul6[2];
    int result;

    udp_port= udp_fd->uf_port;

    if (udp_port->up_flags & UPF_WRITE_IP)
    {
        udp_port->up_flags |= UPF_MORE2WRITE;
        return;
    }

    assert (udp_fd->uf_flags & UFF_WRITE_IP);
    udp_fd->uf_flags &= ~UFF_WRITE_IP;

    assert (!udp_port->up_wr_pack);

    pack= (*udp_fd->uf_get_userdata)(udp_fd->uf_srfd, 0,
        udp_fd->uf_wr_count, FALSE);
    if (!pack)
    {
        udp_fd->uf_flags &= ~UFF_WRITE_IP;
        reply_thr_get (udp_fd, EFAULT, FALSE);
        return;
    }
}
```

```
    }

    flags= udp_fd->uf_udpopt.nwuo_flags;

    ip_hdr_pack= bf_memreq(IP_MIN_HDR_SIZE);
    ip_hdr= (ip_hdr_t *)ptr2acc_data(ip_hdr_pack);

    udp_hdr_pack= bf_memreq(UDP_HDR_SIZE);
    udp_hdr= (udp_hdr_t *)ptr2acc_data(udp_hdr_pack);

    if (flags & NWUO_RWDATALL)
    {
        pack= bf_packIfLess(pack, UDP_IO_HDR_SIZE);
        udp_io_hdr= (udp_io_hdr_t *)ptr2acc_data(pack);
    }
    #if CONF_UDP_IO_NW_BYTE_ORDER
        ip_opt_size= ntohs(udp_io_hdr->uih_ip_opt_len);
    #else
        ip_opt_size= udp_io_hdr->uih_ip_opt_len;
    #endif

    if (UDP_IO_HDR_SIZE+ip_opt_size>udp_fd->uf_wr_count)
    {
        bf_afree(ip_hdr_pack);
        bf_afree(udp_hdr_pack);
        bf_afree(pack);
        reply_thr_get (udp_fd, EINVAL, FALSE);
        return;
    }
    if (ip_opt_size & 3)
    {
        bf_afree(ip_hdr_pack);
        bf_afree(udp_hdr_pack);
        bf_afree(pack);
        reply_thr_get (udp_fd, EFAULT, FALSE);
        return;
    }
    if (ip_opt_size)
        ip_opt_pack= bf_cut(pack, UDP_IO_HDR_SIZE, ip_opt_size);
    else
        ip_opt_pack= 0;
    user_data_size= udp_fd->uf_wr_count-UDP_IO_HDR_SIZE-
        ip_opt_size;
    user_data= bf_cut(pack, UDP_IO_HDR_SIZE+ip_opt_size,
        user_data_size);
    bf_afree(pack);
}
else
{
    udp_io_hdr= 0;
    ip_opt_size= 0;
    user_data_size= udp_fd->uf_wr_count;
    ip_opt_pack= 0;
    user_data= pack;
}

ip_hdr->ih_vers_ihl= (IP_MIN_HDR_SIZE+ip_opt_size) >> 2;
ip_hdr->ih_tos= UDP_TOS;
ip_hdr->ih_flags_fragoff= HTONS(UDP_IP_FLAGS);
ip_hdr->ih_ttl= IP_DEF_TTL;
ip_hdr->ih_proto= IPPROTO_UDP;
if (flags & NWUO_RA_SET)
{
    ip_hdr->ih_dst= udp_fd->uf_udpopt.nwuo_remaddr;
}
else
{
    assert (udp_io_hdr);
    ip_hdr->ih_dst= udp_io_hdr->uih_dst_addr;
}

if ((flags & NWUO_LOCPORT_MASK) != NWUO_LP_ANY)
    udp_hdr->uh_src_port= udp_fd->uf_udpopt.nwuo_locport;
else
{
    assert (udp_io_hdr);
```

```

        udp_hdr->uh_src_port= udp_io_hdr->uih_src_port;
    }

    if (flags & NWUO_RP_SET)
        udp_hdr->uh_dst_port= udp_fd->uf_udpopt.nwuo_rempport;
    else
    {
assert (udp_io_hdr);
        udp_hdr->uh_dst_port= udp_io_hdr->uih_dst_port;
    }

    udp_hdr->uh_length= htons(UDP_HDR_SIZE+user_data_size);
    udp_hdr->uh_chksum= 0;

    udp_hdr_pack->acc_next= user_data;
    chksum= pack_oneCsum(udp_hdr_pack);
    chksum= oneC_sum(chksum, (u16_t *)&udp_fd->uf_port->up_ipaddr,
        sizeof(ipaddr_t));
    chksum= oneC_sum(chksum, (u16_t *)&ip_hdr->ih_dst, sizeof(ipaddr_t));
    u16[0]= 0;
    u16[1]= IPPROTO_UDP;
    chksum= oneC_sum(chksum, (u16_t *)u16, sizeof(u16));
    chksum= oneC_sum(chksum, (u16_t *)&udp_hdr->uh_length, sizeof(u16_t));
    if (~chksum)
        chksum= ~chksum;
    udp_hdr->uh_chksum= chksum;

    if (ip_opt_pack)
    {
        ip_opt_pack= bf_packIffLess(ip_opt_pack, ip_opt_size);
        ip_opt_pack->acc_next= udp_hdr_pack;
        udp_hdr_pack= ip_opt_pack;
    }
    ip_hdr_pack->acc_next= udp_hdr_pack;

assert (!udp_port->up_wr_pack);
assert (!(udp_port->up_flags & UPF_WRITE_IP));

    udp_port->up_wr_pack= ip_hdr_pack;
    udp_port->up_flags |= UPF_WRITE_IP;
    result= ip_write(udp_port->up_ipfd, bf_bufsize(ip_hdr_pack));
    if (result == NW_SUSPEND)
    {
        udp_port->up_flags |= UPF_WRITE_SP;
        udp_fd->uf_flags |= UFF_WRITE_IP;
        udp_port->up_write_fd= udp_fd;
    }
    else if (result<0)
        reply_thr_get(udp_fd, result, FALSE);
    else
        reply_thr_get (udp_fd, udp_fd->uf_wr_count, FALSE);
}

PRIVATE u16_t pack_oneCsum(pack)
acc_t *pack;
{
    u16_t prev;
    int odd_byte;
    char *data_ptr;
    int length;
    char byte_buf[2];

    assert (pack);

    prev= 0;

    odd_byte= FALSE;
    for (; pack; pack= pack->acc_next)
    {
        data_ptr= ptr2acc_data(pack);
        length= pack->acc_length;

        if (!length)

```

```

        continue;
    if (odd_byte)
    {
        byte_buf[1]= *data_ptr;
        prev= oneC_sum(prev, (u16_t *)byte_buf, 2);
        data_ptr++;
        length--;
        odd_byte= FALSE;
    }
    if (length & 1)
    {
        odd_byte= TRUE;
        length--;
        byte_buf[0]= data_ptr[length];
    }
    if (!length)
        continue;
    prev= oneC_sum (prev, (u16_t *)data_ptr, length);
}
if (odd_byte)
{
    byte_buf[1]= 0;
    prev= oneC_sum (prev, (u16_t *)byte_buf, 1);
}
return prev;
}

PRIVATE void udp_restart_write_port(udp_port )
udp_port_t *udp_port;
{
    udp_fd_t *udp_fd;
    int i;

assert (!udp_port->up_wr_pack);
assert (!(udp_port->up_flags & (UPF_WRITE_IP|UPF_WRITE_SP)));

    while (udp_port->up_flags & UPF_MORE2WRITE)
    {
        udp_port->up_flags &= ~UPF_MORE2WRITE;

        for (i= 0, udp_fd= udp_port->up_next_fd; i<UDP_FD_NR;
             i++, udp_fd++)
        {
            if (udp_fd == &udp_fd_table[UDP_FD_NR])
                udp_fd= udp_fd_table;

            if (!(udp_fd->uf_flags & UFF_INUSE))
                continue;
            if (!(udp_fd->uf_flags & UFF_WRITE_IP))
                continue;
            if (udp_fd->uf_port != udp_port)
                continue;
            restart_write_fd(udp_fd);
            if (udp_port->up_flags & UPF_WRITE_IP)
            {
                udp_port->up_next_fd= udp_fd+1;
                udp_port->up_flags |= UPF_MORE2WRITE;
                return;
            }
        }
    }
}

PUBLIC int udp_cancel(fd, which_operation)
int fd;
int which_operation;
{
    udp_fd_t *udp_fd;

    DBLOCK(0x10, printf("udp_cancel(%d,%d)\n", fd, which_operation));

    udp_fd= &udp_fd_table[fd];

    switch (which_operation)

```

```

    {
        case SR_CANCEL_READ:
assert (udp_fd->uf_flags & UFF_READ_IP);
        udp_fd->uf_flags &= ~UFF_READ_IP;
        reply_thr_put(udp_fd, EINTR, FALSE);
        break;
        case SR_CANCEL_WRITE:
assert (udp_fd->uf_flags & UFF_WRITE_IP);
        udp_fd->uf_flags &= ~UFF_WRITE_IP;
        if (udp_fd->uf_port->up_write_fd == udp_fd)
            udp_fd->uf_port->up_write_fd= NULL;
        reply_thr_get(udp_fd, EINTR, FALSE);
        break;
        case SR_CANCEL_IOCTL:
assert (udp_fd->uf_flags & UFF_IOCTL_IP);
        udp_fd->uf_flags &= ~UFF_IOCTL_IP;
        udp_fd->uf_flags &= ~UFF_PEEK_IP;
        reply_thr_get(udp_fd, EINTR, TRUE);
        break;
        default:
            ip_panic(( "got unknown cancel request" ));
    }
    return NW_OK;
}

PRIVATE void udp_buffree (priority)
int priority;
{
    int i;
    udp_fd_t *udp_fd;
    acc_t *tmp_acc;

    if (priority == UDP_PRI_FDBUFS_EXTRA)
    {
        for (i=0, udp_fd= udp_fd_table; i<UDP_FD_NR; i++, udp_fd++)
        {
            while (udp_fd->uf_rdbuf_head &&
                    udp_fd->uf_rdbuf_head->acc_ext_link)
            {
                tmp_acc= udp_fd->uf_rdbuf_head;
                udp_fd->uf_rdbuf_head= tmp_acc->acc_ext_link;
                bf_afree(tmp_acc);
            }
        }

        if (priority == UDP_PRI_FDBUFS)
        {
            for (i=0, udp_fd= udp_fd_table; i<UDP_FD_NR; i++, udp_fd++)
            {
                while (udp_fd->uf_rdbuf_head)
                {
                    tmp_acc= udp_fd->uf_rdbuf_head;
                    udp_fd->uf_rdbuf_head= tmp_acc->acc_ext_link;
                    bf_afree(tmp_acc);
                }
            }
        }
    }
}

PRIVATE void udp_rd_enqueue(udp_fd, pack, exp_tim)
udp_fd_t *udp_fd;
acc_t *pack;
clock_t exp_tim;
{
    acc_t *tmp_acc;
    int result;

    if (pack->acc_linkC != 1)
    {
        tmp_acc= bf_dupacc(pack);
        bf_afree(pack);
        pack= tmp_acc;
    }
}

```

```

pack->acc_ext_link= NULL;
if (udp_fd->uf_rdbuf_head == NULL)
{
    udp_fd->uf_exp_tim= exp_tim;
    udp_fd->uf_rdbuf_head= pack;
}
else
    udp_fd->uf_rdbuf_tail->acc_ext_link= pack;
udp_fd->uf_rdbuf_tail= pack;

if (udp_fd->uf_flags & UFF_PEEK_IP)
{
    pack= bf_cut(udp_fd->uf_rdbuf_head, 0,
                sizeof(udp_io_hdr_t));
    result= (*udp_fd->uf_put_userdata)(udp_fd->uf_srfd,
                                     (size_t)0, pack, TRUE);

    udp_fd->uf_flags &= ~UFF_IOCTL_IP;
    udp_fd->uf_flags &= ~UFF_PEEK_IP;
    result= (*udp_fd->uf_put_userdata)(udp_fd->uf_srfd,
                                     result, (acc_t *)0, TRUE);
    assert (result == 0);
}

if (udp_fd->uf_flags & UFF_SEL_READ)
{
    udp_fd->uf_flags &= ~UFF_SEL_READ;
    if (udp_fd->uf_select_res)
        udp_fd->uf_select_res(udp_fd->uf_srfd, SR_SELECT_READ);
    else
        printf("udp_rd_enqueue: no select_res\n");
}
}

```

```

PRIVATE void hash_fd(udp_fd)
udp_fd_t *udp_fd;
{
    udp_port_t *udp_port;
    int hash;

    udp_port= udp_fd->uf_port;
    if ((udp_fd->uf_udpopt.nwuo_flags & NWUO_LOCPORT_MASK) ==
        NWUO_LP_ANY)
    {
        udp_fd->uf_port_next= udp_port->up_port_any;
        udp_port->up_port_any= udp_fd;
    }
    else
    {
        hash= udp_fd->uf_udpopt.nwuo_locport;
        hash ^= (hash >> 8);
        hash &= (UDP_PORT_HASH_NR-1);

        udp_fd->uf_port_next= udp_port->up_port_hash[hash];
        udp_port->up_port_hash[hash]= udp_fd;
    }
}

```

```

PRIVATE void unhash_fd(udp_fd)
udp_fd_t *udp_fd;
{
    udp_port_t *udp_port;
    udp_fd_t *prev, *curr, **udp_fd_p;
    int hash;

    udp_port= udp_fd->uf_port;
    if ((udp_fd->uf_udpopt.nwuo_flags & NWUO_LOCPORT_MASK) ==
        NWUO_LP_ANY)
    {
        udp_fd_p= &udp_port->up_port_any;
    }
    else
    {
        hash= udp_fd->uf_udpopt.nwuo_locport;

```



```
        hash ^= (hash >> 8);
        hash &= (UDP_PORT_HASH_NR-1);

        udp_fd_p= &udp_port->up_port_hash[hash];
    }
    for (prev= NULL, curr= *udp_fd_p; curr;
        prev= curr, curr= curr->uf_port_next)
    {
        if (curr == udp_fd)
            break;
    }
    assert(curr);
    if (prev)
        prev->uf_port_next= curr->uf_port_next;
    else
        *udp_fd_p= curr->uf_port_next;
}

#ifdef BUF_CONSISTENCY_CHECK
PRIVATE void udp_bufcheck()
{
    int i;
    udp_port_t *udp_port;
    udp_fd_t *udp_fd;
    acc_t *tmp_acc;

    for (i= 0, udp_port= udp_port_table; i<udp_conf_nr; i++, udp_port++)
    {
        if (udp_port->up_wr_pack)
            bf_check_acc(udp_port->up_wr_pack);
    }

    for (i= 0, udp_fd= udp_fd_table; i<UDP_FD_NR; i++, udp_fd++)
    {
        for (tmp_acc= udp_fd->uf_rdbuf_head; tmp_acc;
            tmp_acc= tmp_acc->acc_ext_link)
        {
            bf_check_acc(tmp_acc);
        }
    }
}
#endif

/*
 * $PchId: udp.c,v 1.25 2005/06/28 14:14:44 philip Exp $
 */
```

```
/*
udp.h

Copyright 1995 Philip Homburg
*/

#ifndef UDP_H
#define UDP_H

#define UDP_DEF_OPT          NWUO_NOFLAGS
#define UDP_MAX_DATAGRAM    40000 /* 8192 */
#define UDP_READ_EXP_TIME   (10L * HZ)
#define UDP_TOS              0
#define UDP_IP_FLAGS        0

#define UDP0      0

struct acc;

void udp_prep ARGS(( void ));
void udp_init ARGS(( void ));
int udp_open ARGS(( int port, int srfd,
    get_userdata_t get_userdata, put_userdata_t put_userdata,
    put_pkt_t put_pkt, select_res_t select_res ));
int udp_ioctl ARGS(( int fd, ioreq_t req ));
int udp_read ARGS(( int fd, size_t count ));
int udp_write ARGS(( int fd, size_t count ));
void udp_close ARGS(( int fd ));
int udp_cancel ARGS(( int fd, int which_operation ));

#endif /* UDP_H */

/*
 * $PchId: udp.h,v 1.9 2005/06/28 14:12:05 philip Exp $
 */
```

```

/*
generic/udp_int.h

Created:      March 2001 by Philip Homburg <philip@f-mnx.phicoh.com>

Some internals of the UDP module
*/

#define UDP_FD_NR          (4*IP_PORT_MAX)
#define UDP_PORT_HASH_NR   16          /* Must be a power of 2 */

typedef struct udp_port
{
    int up_flags;
    int up_state;
    int up_ipfd;
    int up_ipdev;
    acc_t *up_wr_pack;
    ipaddr_t up_ipaddr;
    struct udp_fd *up_next_fd;
    struct udp_fd *up_write_fd;
    struct udp_fd *up_port_any;
    struct udp_fd *up_port_hash[UDP_PORT_HASH_NR];
} udp_port_t;

#define UPF_EMPTY          0x0
#define UPF_WRITE_IP       0x1
#define UPF_WRITE_SP       0x2
#define UPF_READ_IP        0x4
#define UPF_READ_SP        0x8
#define UPF_SUSPEND        0x10
#define UPF_MORE2WRITE     0x20

#define UPS_EMPTY          0
#define UPS_SETPROTO        1
#define UPS_GETCONF        2
#define UPS_MAIN            3
#define UPS_ERROR          4

typedef struct udp_fd
{
    int uf_flags;
    udp_port_t *uf_port;
    ioreq_t uf_ioreq;
    int uf_srfd;
    nwio_udpopt_t uf_udpopt;
    get_userdata_t uf_get_userdata;
    put_userdata_t uf_put_userdata;
    select_res_t uf_select_res;
    acc_t *uf_rdbuf_head;
    acc_t *uf_rdbuf_tail;
    size_t uf_rd_count;
    size_t uf_wr_count;
    clock_t uf_exp_tim;
    struct udp_fd *uf_port_next;
} udp_fd_t;

#define UFF_EMPTY          0x0
#define UFF_INUSE          0x1
#define UFF_IOCTL_IP       0x2
#define UFF_READ_IP        0x4
#define UFF_WRITE_IP       0x8
#define UFF_OPTSET         0x10
#define UFF_PEEK_IP        0x20
#define UFF_SEL_READ       0x40
#define UFF_SEL_WRITE      0x80

EXTERN udp_port_t *udp_port_table;
EXTERN udp_fd_t udp_fd_table[UDP_FD_NR];

/*
 * $PchId: udp_int.h,v 1.4 2004/08/03 11:12:01 philip Exp $
 */

```

```
/*      queryparam() - allow program parameters to be queried
 *
 *      Author: Kees J. Bot
 *      21 Apr 1994
 */
#define nil 0
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <minix3/queryparam.h>

#if EXAMPLE
struct stat st[2];

struct export_param_list ex_st_list[] = {
    QP_VARIABLE(st),
    QP_ARRAY(st),
    QP_FIELD(st_dev, struct stat),
    QP_FIELD(st_ino, struct stat),
    ...
    QP_END()
};

struct buf { block_t b_blocknr; ... } *buf;
size_t nr_bufs;

struct export_param_list ex_buf_list[] =
    QP_VECTOR(buf, buf, nr_bufs),
    QP_FIELD(b_blocknr),
    ...
    QP_END()
};

struct export_params ex_st = { ex_st_list, 0 };
struct export_params ex_buf = { ex_buf_list, 0 };
#endif

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

static int isvar(int c)
{
    return between('a', c, 'z') || between('A', c, 'Z')
           || between('0', c, '9') || c == '_';
}

static struct export_params *params;

void qp_export(struct export_params *ex_params)
{
    /* Add a set of exported parameters. */

    if (ex_params->next == nil) {
        ex_params->next = params;
        params = ex_params;
    }
}

int queryparam(int qgetc(void), void **poffset, size_t *psize)
{
    char *prefix;
    struct export_params *ep;
    struct export_param_list *epl;
    size_t offset = 0;
    size_t size = -1;
    size_t n;
    static size_t retval;
    int c, firstc;

    firstc = c = (*qgetc)();
    if (c == '&' || c == '$') c = (*qgetc)();
    if (!isvar(c)) goto fail;

    if ((ep = params) == nil) goto fail;
    epl = ep->list;
```

```

while (c != 0 && c != ',') {
    prefix= "x";
    n= 0;

    for (;;) {
        while (epl->name == nil) {
            if ((ep= ep->next) == nil) goto fail;
            epl= ep->list;
        }
        if (strncmp(prefix, epl->name, n) == 0) {
            prefix= epl->name;
            while (prefix[n] != 0 && c == prefix[n]) {
                n++;
                c= (*qgetc());
            }
        }
        if (prefix[n] == 0 && (!isvar(c) || prefix[0] == '[')) {
            /* Got a match. */
            break;
        }
        epl++;
    }

    if (prefix[0] == '[') {
        /* Array reference. */
        size_t idx= 0, cnt= 1, max= size / epl->size;

        while (between('0', c, '9')) {
            idx= idx * 10 + (c - '0');
            if (idx > max) goto fail;
            c= (*qgetc());
        }
        if (c == ':') {
            cnt= 0;
            while (between('0', (c= (*qgetc())), '9')) {
                cnt= cnt * 10 + (c - '0');
            }
        }
        if (c != ']') goto fail;
        if (idx + cnt > max) cnt= max - idx;
        offset+= idx * epl->size;
        size= cnt * epl->size;
        c= (*qgetc());
    } else
    if (epl->size == -1) {
        /* Vector. */
        offset= (size_t) * (void **) epl->offset;
        size= (* (size_t *) epl[1].offset) * epl[1].size;
    } else {
        /* Variable or struct field. */
        offset+= (size_t) epl->offset;
        if ((size_t) epl->offset > size) goto fail;
        size-= (size_t) epl->offset;
        if (size < epl->size) goto fail;
        size= epl->size;
    }
}
if (firstc == '&' || firstc == '$') {
    retval= firstc == '&' ? offset : size;
    offset= (size_t) &retval;
    size= sizeof(retval);
}
if (c != 0 && c != ',') goto fail;
*poffset= (void *) offset;
*psize= size;
return c != 0;

fail:
while (c != 0 && c != ',') c= (*qgetc());
*poffset= nil;
*psize= 0;
return c != 0;
}

```

```
/*  
 * $PchId: queryparam.c,v 1.1 2005/06/28 14:30:56 philip Exp $  
 */
```

```
/*      queryparam.h - query program parameters      Author: Kees J. Bot
*                                              22 Apr 1994
*/
#ifndef _MINIX__QUERYPARAM_H
#define _MINIX__QUERYPARAM_H

#include <ansi.h>

typedef size_t _mnx_size_t;

struct export_param_list {
    char      *name;          /* "variable", "[", ".field", or NULL. */
    void      *offset;        /* Address of a variable or field offset. */
    size_t    size;           /* Size of the resulting object. */
};

struct export_params {
    struct export_param_list *list; /* List of exported parameters. */
    struct export_params     *next; /* Link several sets of parameters. */
};

#ifdef __STDC__
#define qp_stringize(var)      #var
#define qp_dotstringize(var)  "." #var
#else
#define qp_stringize(var)      "var"
#define qp_dotstringize(var)  ".var"
#endif
#define QP_VARIABLE(var)      { qp_stringize(var), &(var), sizeof(var) }
#define QP_ARRAY(var)         { "[", 0, sizeof((var)[0]) }
#define QP_VECTOR(var,ptr,len) { qp_stringize(var), &(ptr), -1 }, \
                                {"[", &(len), sizeof(*(ptr)) }
#define QP_FIELD(field, type) { qp_dotstringize(field), \
                                (void *)offsetof(type, field), \
                                sizeof(((type *)0)->field) }
#define QP_END()              { 0, 0, 0 }

void qp_export_ARGS((struct export_params *_ex_params));
int queryparam_ARGS((int (*_qgetc) _ARGS((void)), void **_paddress,
                    _mnx_size_t *_psize));
_mnx_size_t paramvalue_ARGS((char **_value, void *_address,
                             _mnx_size_t _size));
#endif /* _MINIX__QUERYPARAM_H */

/* $PchId: queryparam.h,v 1.1 2005/06/28 14:31:26 philip Exp $ */
```

```
# Makefile for the init program (INIT)
SERVER = init

# directories
u = /usr
i = $u/include
s = $i/sys
h = $i/minix
k = $u/src/kernel

# programs, flags, etc.
CC =      exec cc
CFLAGS = -I$i -O -D_MINIX -D_POSIX_SOURCE
LDFLAGS = -i

OBJ =     init.o

# build local binary
all build:      $(SERVER)
$(SERVER):      $(OBJ)
                $(CC) $(CFLAGS) -o $@ $(LDFLAGS) $(OBJ) -lsysutil
                install -S 8k $@

# install with other servers
install:      /usr/sbin/$(SERVER)
/usr/sbin/$(SERVER):      $(SERVER)
                install -o root -cs $? $@

# clean up local files
clean:
                rm -f $(SERVER) *.o *.bak

depend:
                /usr/bin/mkdep "$(CC) -E $(CPPFLAGS)" *.c > .depend

# Include generated dependencies.
include .depend
```



```

/* This process is the father (mother) of all Minix user processes.  When
 * Minix comes up, this is process number 2, and has a pid of 1.  It
 * executes the /etc/rc shell file, and then reads the /etc/ttytab file to
 * determine which terminals need a login process.
 *
 * If the files /usr/adm/wtmp and /etc/utmp exist and are writable, init
 * (with help from login) will maintain login accounting.  Sending a
 * signal 1 (SIGHUP) to init will cause it to rescan /etc/ttytab and start
 * up new shell processes if necessary.  It will not, however, kill off
 * login processes for lines that have been turned off; do this manually.
 * Signal 15 (SIGTERM) makes init stop spawning new processes, this is
 * used by shutdown and friends when they are about to close the system
 * down.
 */

#include <minix/type.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <sys/svrctl.h>
#include <ttyent.h>
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <signal.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>
#include <unistd.h>
#include <utmp.h>

/* Command to execute as a response to the three finger salute. */
char *REBOOT_CMD[] = { "shutdown", "now", "CTRL-ALT-DEL", NULL };

/* Associated fake ttytab entry. */
struct ttyent TT_REBOOT = { "console", "-", REBOOT_CMD, NULL };

char PATH_UTMP[] = "/etc/utmp";          /* current logins */
char PATH_WTMP[] = "/usr/adm/wtmp";      /* login/logout history */

#define PIDSLOTS          32              /* first this many ttys can be on */

struct slotent {
    int errct;                          /* error count */
    pid_t pid;                          /* pid of login process for this tty line */
};

#define ERRCT_DISABLE     10              /* disable after this many errors */
#define NO_PID            0              /* pid value indicating no process */

struct slotent slots[PIDSLOTS]; /* init table of ttys and pids */

int gothup = 0;                          /* flag, showing signal 1 was received */
int gotabrt = 0;                          /* flag, showing signal 6 was received */
int spawn = 1;                            /* flag, spawn processes only when set */

void tell(int fd, char *s);
void report(int fd, char *label);
void wtmp(int type, int linenr, char *line, pid_t pid);
void startup(int linenr, struct ttyent *ttyp);
int execute(char **cmd);
void onhup(int sig);
void onterm(int sig);
void onabrt(int sig);

int main(void)
{
    pid_t pid;                          /* pid of child process */
    int fd;                             /* generally useful */
    int linenr;                          /* loop variable */
    int check;                           /* check if a new process must be spawned */
    struct slotent *slotp;               /* slots[] pointer */
    struct ttyent *ttyp;                 /* ttytab entry */
    struct sigaction sa;

```

```

struct stat stb;

if (fstat(0, &stb) < 0) {
    /* Open standard input, output & error. */
    (void) open("/dev/null", O_RDONLY);
    (void) open("/dev/log", O_WRONLY);
    dup(1);
}

sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;

/* Hangup: Reexamine /etc/ttytab for newly enabled terminal lines. */
sa.sa_handler = onhup;
sigaction(SIGHUP, &sa, NULL);

/* Terminate: Stop spawning login processes, shutdown is near. */
sa.sa_handler = onterm;
sigaction(SIGTERM, &sa, NULL);

/* Abort: Sent by the kernel on CTRL-ALT-DEL; shut the system down. */
sa.sa_handler = onabrt;
sigaction(SIGABRT, &sa, NULL);

/* Execute the /etc/rc file. */
if ((pid = fork()) != 0) {
    /* Parent just waits. */
    while (wait(NULL) != pid) {
        if (gotabrt) reboot(RBT_HALT);
    }
} else {
    #if ! SYS_GETKENV
        struct sysgetenv sysgetenv;
    #endif

    char bootopts[16];
    static char *rc_command[] = { "sh", "/etc/rc", NULL, NULL, NULL };
    char **rcp = rc_command + 2;

    /* Get the boot options from the boot environment. */
    sysgetenv.key = "bootopts";
    sysgetenv.keylen = 8+1;
    sysgetenv.val = bootopts;
    sysgetenv.vallen = sizeof(bootopts);
    if (svrctl(MMGETPARAM, &sysgetenv) == 0) *rcp++ = bootopts;
    *rcp = "start";

    execute(rc_command);
    report(2, "sh/etc/rc");
    _exit(1); /* impossible, we hope */
}

/* Clear /etc/utmp if it exists. */
if ((fd = open(PATH_UTMP, O_WRONLY | O_TRUNC)) >= 0) close(fd);

/* Log system reboot. */
wtmp(BOOT_TIME, 0, NULL, 0);

/* Main loop. If login processes have already been started up, wait for one
 * to terminate, or for a HUP signal to arrive. Start up new login processes
 * for all ttys which don't have them. Note that wait() also returns when
 * somebody's orphan dies, in which case ignore it. If the TERM signal is
 * sent then stop spawning processes, shutdown time is near.
 */

check = 1;
while (1) {
    while ((pid = waitpid(-1, NULL, check ? WNOHANG : 0)) > 0) {
        /* Search to see which line terminated. */
        for (linenr = 0; linenr < PIDSLOTS; linenr++) {
            slotp = &slots[linenr];
            if (slotp->pid == pid) {
                /* Record process exiting. */
                wtmp(DEAD_PROCESS, linenr, NULL, pid);
                slotp->pid = NO_PID;
            }
        }
    }
}

```

```

        }
        }
    }

    /* If a signal 1 (SIGHUP) is received, simply reset error counts. */
    if (gothup) {
        gothup = 0;
        for (linenr = 0; linenr < PIDSLOTS; linenr++) {
            slots[linenr].errct = 0;
        }
        check = 1;
    }

    /* Shut down on signal 6 (SIGABRT). */
    if (gotabrt) {
        gotabrt = 0;
        startup(0, &TT_REBOOT);
    }

    if (spawn && check) {
        /* See which lines need a login process started up. */
        for (linenr = 0; linenr < PIDSLOTS; linenr++) {
            slotp = &slots[linenr];
            if ((ttyp = getttyent()) == NULL) break;

            if (ttyp->ty_getty != NULL
                && ttyp->ty_getty[0] != NULL
                && slotp->pid == NO_PID
                && slotp->errct < ERRCT_DISABLE)
            {
                startup(linenr, ttyp);
            }
        }
        endttyent();
    }
    check = 0;
}

void onhup(int sig)
{
    gothup = 1;
    spawn = 1;
}

void onterm(int sig)
{
    spawn = 0;
}

void onabrt(int sig)
{
    static int count;

    if (++count == 2) reboot(RBT_HALT);
    gotabrt = 1;
}

void startup(int linenr, struct ttyent *ttyp)
{
    /* Fork off a process for the indicated line. */

    struct slotent *slotp;           /* pointer to ttyslot */
    pid_t pid;                       /* new pid */
    int err[2];                      /* error reporting pipe */
    char line[32];                   /* tty device name */
    int status;

    slotp = &slots[linenr];

    /* Error channel for between fork and exec. */
    if (pipe(err) < 0) err[0] = err[1] = -1;

```

```
if ((pid = fork()) == -1 ) {
    report(2, "fork()");
    sleep(10);
    return;
}

if (pid == 0) {
    /* Child */
    close(err[0]);
    fcntl(err[1], F_SETFD, fcntl(err[1], F_GETFD) | FD_CLOEXEC);

    /* A new session. */
    setsid();

    /* Construct device name. */
    strcpy(line, "/dev/");
    strncat(line, tty->ty_name, sizeof(line) - 6);

    /* Open the line for standard input and output. */
    close(0);
    close(1);
    if (open(line, O_RDWR) < 0 || dup(0) < 0) {
        write(err[1], &errno, sizeof(errno));
        _exit(1);
    }

    if (tty->ty_init != NULL && tty->ty_init[0] != NULL) {
        /* Execute a command to initialize the terminal line. */

        if ((pid = fork()) == -1) {
            report(2, "fork()");
            errno = 0;
            write(err[1], &errno, sizeof(errno));
            _exit(1);
        }

        if (pid == 0) {
            alarm(10);
            execute(tty->ty_init);
            report(2, tty->ty_init[0]);
            _exit(1);
        }

        while (waitpid(pid, &status, 0) != pid) {}
        if (status != 0) {
            tell(2, "init: ");
            tell(2, tty->ty_name);
            tell(2, ": ");
            tell(2, tty->ty_init[0]);
            tell(2, ": bad exit status\n");
            errno = 0;
            write(err[1], &errno, sizeof(errno));
            _exit(1);
        }
    }

    /* Redirect standard error too. */
    dup2(0, 2);

    /* Execute the getty process. */
    execute(tty->ty_getty);

    /* Oops, disaster strikes. */
    fcntl(2, F_SETFL, fcntl(2, F_GETFL) | O_NONBLOCK);
    if (linenr != 0) report(2, tty->ty_getty[0]);
    write(err[1], &errno, sizeof(errno));
    _exit(1);
}

/* Parent */
if (tty != &TT_REBOOT) slot->pid = pid;

close(err[1]);
if (read(err[0], &errno, sizeof(errno)) != 0) {
```

```

    /* If an errno value goes down the error pipe: Problems. */

    switch (errno) {
    case ENOENT:
    case ENODEV:
    case ENXIO:
        /* Device nonexistent, no driver, or no minor device. */
        slotp->errct = ERRCT_DISABLE;
        close(err[0]);
        return;
    case 0:
        /* Error already reported. */
        break;
    default:
        /* Any other error on the line. */
        report(2, ttyp->ty_name);
    }
    close(err[0]);

    if (++slotp->errct >= ERRCT_DISABLE) {
        tell(2, "init: ");
        tell(2, ttyp->ty_name);
        tell(2, ": excessive errors, shutting down\n");
    } else {
        sleep(5);
    }
    return;
}
close(err[0]);

if (ttyp != &TT_REBOOT) wtmp(LOGIN_PROCESS, linenr, ttyp->ty_name, pid);
slotp->errct = 0;
}

int execute(char **cmd)
{
    /* Execute a command with a path search along /sbin:/bin:/usr/sbin:/usr/bin.
    */
    static char *nullenv[] = { NULL };
    char command[128];
    char *path[] = { "/sbin", "/bin", "/usr/sbin", "/usr/bin" };
    int i;

    if (cmd[0][0] == '/') {
        /* A full path. */
        return execve(cmd[0], cmd, nullenv);
    }

    /* Path search. */
    for (i = 0; i < 4; i++) {
        if (strlen(path[i]) + 1 + strlen(cmd[0]) + 1 > sizeof(command)) {
            errno = ENAMETOOLONG;
            return -1;
        }
        strcpy(command, path[i]);
        strcat(command, "/");
        strcat(command, cmd[0]);
        execve(command, cmd, nullenv);
        if (errno != ENOENT) break;
    }
    return -1;
}

void wtmp(type, linenr, line, pid)
int type; /* type of entry */
int linenr; /* line number in ttytab */
char *line; /* tty name (only good on login) */
pid_t pid; /* pid of process */
{
    /* Log an event into the UTMP and WTMP files. */

    struct utmp utmp; /* UTMP/WTMP User Accounting */
    int fd;

```

```

/* Clear the utmp record. */
memset((void *) &utmp, 0, sizeof(utmp));

/* Fill in utmp. */
switch (type) {
case BOOT_TIME:
    /* Make a special reboot record. */
    strcpy(utmp.ut_name, "reboot");
    strcpy(utmp.ut_line, "~");
    break;

case LOGIN_PROCESS:
    /* A new login, fill in line name. */
    strncpy(utmp.ut_line, line, sizeof(utmp.ut_line));
    break;

case DEAD_PROCESS:
    /* A logout. Use the current utmp entry, but make sure it is a
     * user process exiting, and not getty or login giving up.
     */
    if ((fd = open(PATH_UTMP, O_RDONLY)) < 0) {
        if (errno != ENOENT) report(2, PATH_UTMP);
        return;
    }
    if (lseek(fd, (off_t) (linenr+1) * sizeof(utmp), SEEK_SET) == -1
        || read(fd, &utmp, sizeof(utmp)) == -1
    ) {
        report(2, PATH_UTMP);
        close(fd);
        return;
    }
    close(fd);
    if (utmp.ut_type != USER_PROCESS) return;
    strncpy(utmp.ut_name, "", sizeof(utmp.ut_name));
    break;
}

/* Finish new utmp entry. */
utmp.ut_pid = pid;
utmp.ut_type = type;
utmp.ut_time = time((time_t *) 0);

switch (type) {
case LOGIN_PROCESS:
case DEAD_PROCESS:
    /* Write new entry to utmp. */
    if ((fd = open(PATH_UTMP, O_WRONLY)) < 0
        || lseek(fd, (off_t) (linenr+1) * sizeof(utmp), SEEK_SET) == -1
        || write(fd, &utmp, sizeof(utmp)) == -1
    ) {
        if (errno != ENOENT) report(2, PATH_UTMP);
    }
    if (fd != -1) close(fd);
    break;
}

switch (type) {
case BOOT_TIME:
case DEAD_PROCESS:
    /* Add new wtmp entry. */
    if ((fd = open(PATH_WTMP, O_WRONLY | O_APPEND)) < 0
        || write(fd, &utmp, sizeof(utmp)) == -1
    ) {
        if (errno != ENOENT) report(2, PATH_WTMP);
    }
    if (fd != -1) close(fd);
    break;
}
}

void tell(fd, s)
int fd;
char *s;
{

```

```
        write(fd, s, strlen(s));
    }

void report(fd, label)
int fd;
char *label;
{
    int err = errno;

    tell(fd, "init: ");
    tell(fd, label);
    tell(fd, ": ");
    tell(fd, strerror(err));
    tell(fd, "\n");
    errno= err;
}
```

```
# Makefile for Information Server (IS)
SERVER = is

# directories
u = /usr
i = $u/include
s = $i/sys
m = $i/minix
b = $i/ibm
k = $u/src/kernel
p = $u/src/servers/pm
f = $u/src/servers/fs

# programs, flags, etc.
CC =      exec cc
CFLAGS = -I$i
LDFLAGS = -i
LIBS = -lsys -lsysutil

OBJ = main.o dmp.o dmp_kernel.o dmp_pm.o dmp_fs.o dmp_rs.o dmp_ds.o

# build local binary
all build:      $(SERVER)
$(SERVER):      $(OBJ)
                $(CC) -o $@ $(LDFLAGS) $(OBJ) $(LIBS)
#              install -S 256w $@

# install with other servers
install:      $(SERVER)
                install -o root -c $? /sbin/$(SERVER)

# clean up local files
clean:
                rm -f $(SERVER) *.o *.bak

depend:
                /usr/bin/mkdep "$(CC) -E $(CPPFLAGS)" *.c > .depend

# Include generated dependencies.
include .depend
```



```

/* This file contains information dump procedures. During the initialization
 * of the Information Service 'known' function keys are registered at the TTY
 * server in order to receive a notification if one is pressed. Here, the
 * corresponding dump procedure is called.
 *
 * The entry points into this file are
 *   handle_fkey:      handle a function key pressed notification
 */

#include "inc.h"

/* Define hooks for the debugging dumps. This table maps function keys
 * onto a specific dump and provides a description for it.
 */
#define NHOOKS 18

struct hook_entry {
    int key;
    void (*function)(void);
    char *name;
} hooks[NHOOKS] = {
    { F1,   proctab_dmp, "Kernel process table" },
    { F2,   memmap_dmp, "Process memory maps" },
    { F3,   image_dmp, "System image" },
    { F4,   privileges_dmp, "Process privileges" },
    { F5,   monparams_dmp, "Boot monitor parameters" },
    { F6,   irqtab_dmp, "IRQ hooks and policies" },
    { F7,   kmessages_dmp, "Kernel messages" },
    { F9,   sched_dmp, "Scheduling queues" },
    { F10,  kenv_dmp, "Kernel parameters" },
    { F11,  timing_dmp, "Timing details (if enabled)" },
    { SF1,  mproc_dmp, "Process manager process table" },
    { SF2,  sigaction_dmp, "Signals" },
    { SF3,  fproc_dmp, "Filesystem process table" },
    { SF4,  dtab_dmp, "Device/Driver mapping" },
    { SF5,  mapping_dmp, "Print key mappings" },
    { SF6,  rproc_dmp, "Reincarnation server process table" },
    { SF7,  holes_dmp, "Memory free list" },
    { SF8,  data_store_dmp, "Data store contents" },
};

/*=====
 *                               handle_fkey                               *
 *=====*/
#define pressed(k) ((F1<=(k)&&(k)<=F12 && bit_isset(m->FKEY_FKEYS,((k)-F1+1)))\
    || (SF1<=(k) && (k)<=SF12 && bit_isset(m->FKEY_SFKEYS, ((k)-SF1+1))))
PUBLIC int do_fkey_pressed(m)
message *m;                               /* notification message */
{
    int s, h;

    /* The notification message does not convey any information, other
     * than that some function keys have been pressed. Ask TTY for details.
     */
    m->m_type = FKEY_CONTROL;
    m->FKEY_REQUEST = FKEY_EVENTS;
    if (OK != (s=sendrec(TTY_PROC_NR, m)))
        report("IS", "warning, sendrec to TTY failed", s);

    /* Now check which keys were pressed: F1-F12, SF1-SF12. */
    for(h=0; h < NHOOKS; h++)
        if(pressed(hooks[h].key))
            hooks[h].function();

    /* Don't send a reply message. */
    return(EDONTREPLY);
}

/*=====
 *                               key_name                               *
 *=====*/
PRIVATE char *key_name(int key)
{
    static char name[15];

```

```
    if(key >= F1 && key <= F12)
        sprintf(name, " F%d", key - F1 + 1);
    else if(key >= SF1 && key <= SF12)
        sprintf(name, "Shift+F%d", key - SF1 + 1);
    else
        sprintf(name, "?");
    return name;
}

/*=====
 *                               mapping_dmp                               *
 *=====*/
PUBLIC void mapping_dmp(void)
{
    int h;

    printf("Function key mappings for debug dumps in IS server.\n");
    printf("    Key  Description\n");
    printf("-----");
    printf("-----\n");

    for(h=0; h < NHOOKS; h++)
        printf(" %10s. %s\n", key_name(hooks[h].key), hooks[h].name);
    printf("\n");
}
```

```
/* This file contains procedures to dump DS data structures.
 *
 * The entry points into this file are
 *   data_store_dmp:    display DS data store contents
 *
 * Created:
 *   Oct 18, 2005:      by Jorrit N. Herder
 */

#include "inc.h"
#include "../ds/store.h"

PUBLIC struct data_store store[NR_DS_KEYS];

FORWARD _PROTOTYPE( char *s_flags_str, (int flags)                );

/*=====
 *                               data_store_dmp                               *
 *=====*/
PUBLIC void data_store_dmp()
{
    struct data_store *dsp;
    int i,j, n=0;
    static int prev_i=0;

    printf("Data Store (DS) contents dump\n");

    getsysinfo(DS_PROC_NR, SI_DATA_STORE, store);

    printf("-slot- -key- -flags- -val_l1- -val_l2-\n");

    for (i=prev_i; i<NR_DS_KEYS; i++) {
        dsp = &store[i];
        if (! dsp->ds_flags & DS_IN_USE) continue;
        if (++n > 22) break;
        printf("%3d %8d %s [%8d] [%8d]\n",
                i, dsp->ds_key,
                s_flags_str(dsp->ds_flags),
                dsp->ds_val_l1,
                dsp->ds_val_l2
        );
    }
    if (i >= NR_DS_KEYS) i = 0;
    else printf("--more--\r");
    prev_i = i;
}

PRIVATE char *s_flags_str(int flags)
{
    static char str[5];
    str[0] = (flags & DS_IN_USE) ? 'U' : '-';
    str[1] = (flags & DS_PUBLIC) ? 'P' : '-';
    str[2] = '-';
    str[3] = '\0';

    return(str);
}
```

```

/* This file contains procedures to dump to FS' data structures.
 *
 * The entry points into this file are
 *   dtab_dump:      display device <-> driver mappings
 *   fproc_dump:     display FS process table
 *
 * Created:
 *   Oct 01, 2004:    by Jorrit N. Herder
 */

#include "inc.h"
#include "../fs/const.h"
#include "../fs/fproc.h"
#include <minix/dmap.h>

PUBLIC struct fproc fproc[NR_PROCS];
PUBLIC struct dmap dmap[NR_DEVICES];

/*=====
 *                               fproc_dmp                               *
 *=====*/
PUBLIC void fproc_dmp()
{
    struct fproc *fp;
    int i, n=0;
    static int prev_i;

    getsysinfo(FS_PROC_NR, SI_PROC_TAB, fproc);

    printf("File System (FS) process table dump\n");
    printf("-nr- -pid- -tty- -umask- --uid-- --gid-- -ldr- -sus-rev-proc- -cloexec-\n");
    for (i=prev_i; i<NR_PROCS; i++) {
        fp = &fproc[i];
        if (fp->fp_pid <= 0) continue;
        if (++n > 22) break;
        printf("%3d %4d %2d/%d 0x%05x %2d(%d) %2d(%d) %3d %3d %3d %4d 0x%05x\n",
            i, fp->fp_pid,
            ((fp->fp_tty>>MAJOR)&BYTE), ((fp->fp_tty>>MINOR)&BYTE),
            fp->fp_umask,
            fp->fp_realuid, fp->fp_effuid, fp->fp_realgid, fp->fp_effgid,
            fp->fp_sesldr,
            fp->fp_suspended, fp->fp_revived, fp->fp_task,
            fp->fp_cloexec
        );
    }
    if (i >= NR_PROCS) i = 0;
    else printf("--more--\r");
    prev_i = i;
}

/*=====
 *                               dmap_flags                               *
 *=====*/
PRIVATE char * dmap_flags(int flags)
{
    static char fl[10];
    strcpy(fl, "---");
    if(flags & DMAP_MUTABLE) fl[0] = 'M';
    if(flags & DMAP_BUSY) fl[1] = 'S';
    if(flags & DMAP_BABY) fl[2] = 'B';
    return fl;
}

/*=====
 *                               dtab_dmp                               *
 *=====*/
PUBLIC void dtab_dmp()
{
    int i;

    getsysinfo(FS_PROC_NR, SI_DMAP_TAB, dmap);

    printf("File System (FS) device <-> driver mappings\n");
    printf("Major Driver ept Flags\n");

```

```
printf("-----\n");  
for (i=0; i<NR_DEVICES; i++) {  
    if (dmap[i].dmap_driver == NONE) continue;  
    printf("%5d %10d %s\n",  
        i, dmap[i].dmap_driver, dmap_flags(dmap[i].dmap_flags));  
}  
}
```

```

/* Debugging dump procedures for the kernel. */

#include "inc.h"
#include <timers.h>
#include <ibm/interrupt.h>
#include <minix/endpoint.h>
#include "../kernel/const.h"
#include "../kernel/config.h"
#include "../kernel/debug.h"
#include "../kernel/type.h"
#include "../kernel/proc.h"
#include "../kernel/ipc.h"

#define click_to_round_k(n) \
    ((unsigned) (((unsigned long) (n) << CLICK_SHIFT) + 512) / 1024))

/* Declare some local dump procedures. */
FORWARD _PROTOTYPE( char *proc_name, (int proc_nr) );
FORWARD _PROTOTYPE( char *s_traps_str, (int flags) );
FORWARD _PROTOTYPE( char *s_flags_str, (int flags) );
FORWARD _PROTOTYPE( char *p_rts_flags_str, (int flags) );

/* Some global data that is shared among several dumping procedures.
 * Note that the process table copy has the same name as in the kernel
 * so that most macros and definitions from proc.h also apply here.
 */
PUBLIC struct proc proc[NR_TASKS + NR_PROCS];
PUBLIC struct priv priv[NR_SYS_PROCS];
PUBLIC struct boot_image image[NR_BOOT_PROCS];

/*=====
 *
 * timing_dmp
 *=====*/
PUBLIC void timing_dmp()
{
    #if ! DEBUG_TIME_LOCKS
        printf("Enable the DEBUG_TIME_LOCKS definition in src/kernel/config.h\n");
    #else
        static struct lock_timingdata timingdata[TIMING_CATEGORIES];
        int r, c, f, skipped = 0, printed = 0, maxlines = 23, x = 0;
        static int offsetlines = 0;

        if ((r = sys_getlocktimings(&timingdata[0])) != OK) {
            report("IS", "warning: couldn't get copy of lock timings", r);
            return;
        }

        for(c = 0; c < TIMING_CATEGORIES; c++) {
            int b;
            if (!timingdata[c].lock_timings_range[0] || !timingdata[c].binsize)
                continue;
            x = printf("%-*s: misses %lu, resets %lu, measurements %lu: ",
                TIMING_NAME, timingdata[c].names,
                timingdata[c].misses,
                timingdata[c].resets,
                timingdata[c].measurements);
            for(b = 0; b < TIMING_POINTS; b++) {
                int w;
                if (!timingdata[c].lock_timings[b])
                    continue;
                x += (w = printf("%5d:%5d", timingdata[c].lock_timings_range[0] +
                    b*timingdata[c].binsize,
                    timingdata[c].lock_timings[b]));
                if (x + w >= 80) { printf("\n"); x = 0; }
            }
            if (x > 0) printf("\n");
        }
    #endif
}

/*=====
 *
 * kmessages_dmp
 *=====*/
PUBLIC void kmessages_dmp()

```

```

{
    struct kmessages kmess;          /* get copy of kernel messages */
    char print_buf[KMESS_BUF_SIZE+1]; /* this one is used to print */
    int start;                       /* calculate start of messages */
    int r;

    /* Try to get a copy of the kernel messages. */
    if ((r = sys_getkmessages(&kmess)) != OK) {
        report("IS", "warning: couldn't get copy of kmessages", r);
        return;
    }

    /* Try to print the kernel messages. First determine start and copy the
     * buffer into a print-buffer. This is done because the messages in the
     * copy may wrap (the kernel buffer is circular).
     */
    start = ((kmess.km_next + KMESS_BUF_SIZE) - kmess.km_size) % KMESS_BUF_SIZE;
    r = 0;
    while (kmess.km_size > 0) {
        print_buf[r] = kmess.km_buf[(start+r) % KMESS_BUF_SIZE];
        r ++;
        kmess.km_size --;
    }
    print_buf[r] = 0;                /* make sure it terminates */
    printf("Dump of all messages generated by the kernel.\n\n");
    printf("%s", print_buf);         /* print the messages */
}

/*=====
 *                               monparams_dmp                               *
 *=====*/
PUBLIC void monparams_dmp()
{
    char val[1024];
    char *e;
    int r;

    /* Try to get a copy of the boot monitor parameters. */
    if ((r = sys_getmonparams(val, sizeof(val))) != OK) {
        report("IS", "warning: couldn't get copy of monitor params", r);
        return;
    }

    /* Append new lines to the result. */
    e = val;
    do {
        e += strlen(e);
        *e++ = '\n';
    } while (*e != 0);

    /* Finally, print the result. */
    printf("Dump of kernel environment strings set by boot monitor.\n");
    printf("\n%s\n", val);
}

/*=====
 *                               irqtab_dmp                               *
 *=====*/
PUBLIC void irqtab_dmp()
{
    int i,r;
    struct irq_hook irq_hooks[NR_IRQ_HOOKS];
    int irq_actids[NR_IRQ_VECTORS];
    struct irq_hook *e; /* irq tab entry */
    char *irq[] = {
        "clock",          /* 00 */
        "keyboard",       /* 01 */
        "cascade",        /* 02 */
        "rs232",          /* 03 */
        "rs232",          /* 04 */
        "NIC(eth)",       /* 05 */
        "floppy",         /* 06 */
        "printer",        /* 07 */
        "",               /* 08 */
    }

```

```

    " ", /* 09 */
    " ", /* 10 */
    " ", /* 11 */
    " ", /* 12 */
    " ", /* 13 */
    "at_wini_0", /* 14 */
    "at_wini_1", /* 15 */
};

if ((r = sys_getirqhooks(irq_hooks)) != OK) {
    report("IS", "warning: couldn't get copy of irq hooks", r);
    return;
}
if ((r = sys_getirqactids(irq_actids)) != OK) {
    report("IS", "warning: couldn't get copy of irq mask", r);
    return;
}

#if 0
printf("irq_actids:");
for (i= 0; i<NR_IRQ_VECTORS; i++)
    printf(" [%d]=0x%08x", i, irq_actids[i]);
printf("\n");
#endif

printf("IRQ policies dump shows use of kernel's IRQ hooks.\n");
printf("-h.id- -proc.nr- -IRQ vector (nr.)- -policy- -notify id-\n");
for (i=0; i<NR_IRQ_HOOKS; i++) {
    e = &irq_hooks[i];
    printf("%3d", i);
    if (e->proc_nr_e==NONE) {
        printf(" <unused>\n");
        continue;
    }
    printf("%10d ", e->proc_nr_e);
    printf(" %9.9s(%02d)", irq[e->irq], e->irq);
    printf(" %s", (e->policy & IRQ_REENABLE) ? "reenable" : " - ");
    printf(" %d", e->notify_id);
    if (irq_actids[e->irq] & (1 << i))
        printf("masked");
    printf("\n");
}
printf("\n");
}

/*=====
*                                     image_dmp                                     *
*=====*/
PUBLIC void image_dmp()
{
    int m, i, j, r;
    struct boot_image *ip;
    static char ipc_to[BITCHUNK_BITS*2];

    if ((r = sys_getimage(image)) != OK) {
        report("IS", "warning: couldn't get copy of image table", r);
        return;
    }
    printf("Image table dump showing all processes included in system image.\n");
    printf("----name---nr--flags--traps--sq-----pc--stack--ipc_to[0]-----\n");
    for (m=0; m<NR_BOOT_PROCS; m++) {
        ip = &image[m];
        for (i=j=0; i < BITCHUNK_BITS; i++, j++) {
            ipc_to[j] = (ip->ipc_to & (1<<i)) ? '1' : '0';
            if (i % 8 == 7) ipc_to[++j] = ' ';
        }
        ipc_to[j] = '\0';
        printf("%8s %4d %s %s %3d %7lu %7lu %s\n",
            ip->proc_name, ip->proc_nr,
            s_flags_str(ip->flags), s_traps_str(ip->trap_mask),
            ip->priority, (long)ip->initial_pc, ip->stksize, ipc_to);
    }
    printf("\n");
}

```



```

/*=====
*
*                                sched_dmp
*=====*/
PUBLIC void sched_dmp()
{
    struct proc *rdy_head[NR_SCHED_QUEUES];
    struct kinfo kinfo;
    register struct proc *rp;
    vir_bytes ptr_diff;
    int r;

    /* First obtain a scheduling information. */
    if ((r = sys_getschedinfo(proc, rdy_head)) != OK) {
        report("IS", "warning: couldn't get copy of process table", r);
        return;
    }
    /* Then obtain kernel addresses to correct pointer information. */
    if ((r = sys_getkinfo(&kinfo)) != OK) {
        report("IS", "warning: couldn't get kernel addresses", r);
        return;
    }

    /* Update all pointers. Nasty pointer algorithmic ... */
    ptr_diff = (vir_bytes) proc - (vir_bytes) kinfo.proc_addr;
    for (r=0; r<NR_SCHED_QUEUES; r++)
        if (rdy_head[r] != NIL_PROC)
            rdy_head[r] =
                (struct proc *)((vir_bytes) rdy_head[r] + ptr_diff);
    for (rp=BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++)
        if (rp->p_nextready != NIL_PROC)
            rp->p_nextready =
                (struct proc *)((vir_bytes) rp->p_nextready + ptr_diff);

    /* Now show scheduling queues. */
    printf("Dumping scheduling queues.\n");

    for (r=0; r<NR_SCHED_QUEUES; r++) {
        rp = rdy_head[r];
        if (!rp) continue;
        printf("%2d: ", r);
        while (rp != NIL_PROC) {
            printf("%3d ", rp->p_nr);
            rp = rp->p_nextready;
        }
        printf("\n");
    }
    printf("\n");
}

/*=====
*
*                                kenv_dmp
*=====*/
PUBLIC void kenv_dmp()
{
    struct kinfo kinfo;
    struct machine machine;
    int r;
    if ((r = sys_getkinfo(&kinfo)) != OK) {
        report("IS", "warning: couldn't get copy of kernel info struct", r);
        return;
    }
    if ((r = sys_getmachine(&machine)) != OK) {
        report("IS", "warning: couldn't get copy of kernel machine struct", r);
        return;
    }

    printf("Dump of kinfo and machine structures.\n\n");
    printf("Machine structure:\n");
    printf("- pc_at:   %3d\n", machine.pc_at);
    printf("- ps_mca:   %3d\n", machine.ps_mca);
    printf("- processor: %3d\n", machine.processor);
    printf("- protected: %3d\n", machine.protected);
    printf("- vdu_ega:  %3d\n", machine.vdu_ega);
}

```

```

printf("-vdu_vga: %3d\n\n", machine.vdu_vga);
printf("Kernel info structure:\n");
printf("-code_base: %5u\n", kinfo.code_base);
printf("-code_size: %5u\n", kinfo.code_size);
printf("-data_base: %5u\n", kinfo.data_base);
printf("-data_size: %5u\n", kinfo.data_size);
printf("-proc_addr: %5u\n", kinfo.proc_addr);
printf("-kmem_base: %5u\n", kinfo.kmem_base);
printf("-kmem_size: %5u\n", kinfo.kmem_size);
printf("-bootdev_base: %5u\n", kinfo.bootdev_base);
printf("-bootdev_size: %5u\n", kinfo.bootdev_size);
printf("-ramdev_base: %5u\n", kinfo.ramdev_base);
printf("-ramdev_size: %5u\n", kinfo.ramdev_size);
printf("-params_base: %5u\n", kinfo.params_base);
printf("-params_size: %5u\n", kinfo.params_size);
printf("-nr_procs: %3u\n", kinfo.nr_procs);
printf("-nr_tasks: %3u\n", kinfo.nr_tasks);
printf("-release: %.6s\n", kinfo.release);
printf("-version: %.6s\n", kinfo.version);
#if DEBUG_LOCK_CHECK
printf("-relocking: %d\n", kinfo.relocking);
#endif
printf("\n");
}

PRIVATE char *s_flags_str(int flags)
{
    static char str[10];
    str[0] = (flags & PREEMPTIBLE) ? 'P' : '-';
    str[1] = '-';
    str[2] = (flags & BILLABLE) ? 'B' : '-';
    str[3] = (flags & SYS_PROC) ? 'S' : '-';
    str[4] = '-';
    str[5] = '\0';

    return str;
}

PRIVATE char *s_traps_str(int flags)
{
    static char str[10];
    str[0] = (flags & (1 << ECHO)) ? 'E' : '-';
    str[1] = (flags & (1 << SEND)) ? 'S' : '-';
    str[2] = (flags & (1 << RECEIVE)) ? 'R' : '-';
    str[3] = (flags & (1 << SENDREC)) ? 'B' : '-';
    str[4] = (flags & (1 << NOTIFY)) ? 'N' : '-';
    str[5] = '\0';

    return str;
}

/*=====
*                                     privileges_dmp                                     *
*=====*/
PUBLIC void privileges_dmp()
{
    register struct proc *rp;
    static struct proc *oldrp = BEG_PROC_ADDR;
    register struct priv *sp;
    static char ipc_to[NR_SYS_PROCS + 1 + NR_SYS_PROCS/8];
    int r, i, j, n = 0;

    /* First obtain a fresh copy of the current process and system table. */
    if ((r = sys_getprivtab(priv)) != OK) {
        report("IS", "warning: couldn't get copy of system privileges table", r);
        return;
    }
    if ((r = sys_getproctab(proc)) != OK) {
        report("IS", "warning: couldn't get copy of process table", r);
        return;
    }

    printf("\n--nr-id-name---- -flags- -traps- -ipc_to mask----- \n");

```

```

for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
    if (isempty(rp)) continue;
    if (++n > 23) break;
    if (proc_nr(rp) == IDLE)          printf("(%2d ", proc_nr(rp));
    else if (proc_nr(rp) < 0)         printf("[%2d]", proc_nr(rp));
    else                             printf(" %2d ", proc_nr(rp));
    r = -1;
    for (sp = &priv[0]; sp < &priv[NR_SYS_PROCS]; sp++)
        if (sp->s_proc_nr == rp->p_nr) { r++; break; }
    if (r == -1 && ! (rp->p_rts_flags & SLOT_FREE)) {
        sp = &priv[USER_PRIV_ID];
    }
    printf("(%02u) %-7.7s %s %s ",
           sp->s_id, rp->p_name,
           s_flags_str(sp->s_flags), s_traps_str(sp->s_trap_mask)
    );
    for (i=j=0; i < NR_SYS_PROCS; i++, j++) {
        ipc_to[j] = get_sys_bit(sp->s_ipc_to, i) ? '1' : '0';
        if (i % 8 == 7) ipc_to[++j] = ' ';
    }
    ipc_to[j] = '\0';

    printf(" %s\n", ipc_to);
}
if (rp == END_PROC_ADDR) rp = BEG_PROC_ADDR; else printf("--more--\r");
oldrp = rp;
}

/*=====
*                                     sendmask_dmp                                     *
*=====*/
PUBLIC void sendmask_dmp()
{
    register struct proc *rp;
    static struct proc *oldrp = BEG_PROC_ADDR;
    int r, i, j, n = 0;

    /* First obtain a fresh copy of the current process table. */
    if ((r = sys_getproctab(proc)) != OK) {
        report("IS", "warning: couldn't get copy of process table", r);
        return;
    }

    printf("\n\n");
    printf("Sendmask dump for process table. User processes (*) don't have [].");
    printf("\n");
    printf("The rows of bits indicate to which processes each process may send.");
    printf("\n\n");

    #if DEAD_CODE
    printf(" ");
    for (j=proc_nr(BEG_PROC_ADDR); j< INIT_PROC_NR+1; j++) {
        printf("%3d", j);
    }
    printf(" *\n");

    for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
        if (isempty(rp)) continue;
        if (++n > 20) break;

        printf("%8s ", rp->p_name);
        if (proc_nr(rp) == IDLE)          printf("(%2d ", proc_nr(rp));
        else if (proc_nr(rp) < 0)         printf("[%2d]", proc_nr(rp));
        else                             printf(" %2d ", proc_nr(rp));

        for (j=proc_nr(BEG_PROC_ADDR); j<INIT_PROC_NR+2; j++) {
            if (isallowed(rp->p_sendmask, j)) printf(" 1");
            else                             printf(" 0");
        }
        printf("\n");
    }
    if (rp == END_PROC_ADDR) { printf("\n"); rp = BEG_PROC_ADDR; }
    else printf("--more--\r");
}

```

```

    oldrp = rp;
#endif
}

PRIVATE char *p_rts_flags_str(int flags)
{
    static char str[10];
    str[0] = (flags & NO_PRIORITY) ? 's' : '-';
    str[1] = (flags & SENDING) ? 'S' : '-';
    str[2] = (flags & RECEIVING) ? 'R' : '-';
    str[3] = (flags & SINGALED) ? 'I' : '-';
    str[4] = (flags & SIG_PENDING) ? 'P' : '-';
    str[5] = (flags & P_STOP) ? 'T' : '-';
    str[6] = (flags & NO_PRIV) ? 'p' : '-';
    str[7] = '\0';

    return str;
}

/*=====
*
*                                proctab_dmp
*=====*/
#ifdef (CHIP == INTEL)
PUBLIC void proctab_dmp()
{
    /* Proc table dump */

    register struct proc *rp;
    static struct proc *oldrp = BEG_PROC_ADDR;
    int r, n = 0;
    phys_clicks text, data, size;

    /* First obtain a fresh copy of the current process table. */
    if ((r = sys_getproctab(proc)) != OK) {
        report("IS", "warning: couldn't get copy of process table", r);
        return;
    }

    printf("\n-nr-----gen---endpoint-name--- -prior-quant- -user----sys-----size-rts flags-\n");

    for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
        if (isemptyp(rp)) continue;
        if (++n > 23) break;
        text = rp->p_memmap[T].mem_phys;
        data = rp->p_memmap[D].mem_phys;
        size = rp->p_memmap[T].mem_len
            + ((rp->p_memmap[S].mem_phys + rp->p_memmap[S].mem_len) - data);
        if (proc_nr(rp) == IDLE) printf("(%2d", proc_nr(rp));
        else if (proc_nr(rp) < 0) printf("[%2d]", proc_nr(rp));
        else printf("%2d", proc_nr(rp));
        printf(" %5d %10d", _ENDPOINT_G(rp->p_endpoint), rp->p_endpoint);
        printf(" %-8.8s %02u/%02u %02d/%02u %6lu%6lu %6uK %s",
            rp->p_name,
            rp->p_priority, rp->p_max_priority,
            rp->p_ticks_left, rp->p_quantum_size,
            rp->p_user_time, rp->p_sys_time,
            click_to_round_k(size),
            p_rts_flags_str(rp->p_rts_flags));
        if (rp->p_rts_flags & (SENDING|RECEIVING)) {
            printf(" %-7.7s", proc_name(_ENDPOINT_P(rp->p_getfrom_e)));
        }
        printf("\n");
    }
    if (rp == END_PROC_ADDR) rp = BEG_PROC_ADDR; else printf("--more--\r");
    oldrp = rp;
}
#endif

/* (CHIP == INTEL) */

/*=====
*
*                                memmap_dmp
*=====*/
PUBLIC void memmap_dmp()
{
    register struct proc *rp;

```

```

static struct proc *oldrp = proc;
int r, n = 0;
phys_clicks size;

/* First obtain a fresh copy of the current process table. */
if ((r = sys_getproctab(proc)) != OK) {
    report("IS", "warning: couldn't get copy of process table", r);
    return;
}

printf("\n-nr/name-----pc---sp-----text-----data-----stack-----size-\n");
for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
    if (isemptyp(rp)) continue;
    if (++n > 23) break;
    size = rp->p_memmap[T].mem_len
        + ((rp->p_memmap[S].mem_phys + rp->p_memmap[S].mem_len)
           - rp->p_memmap[D].mem_phys);
    printf("%3d %-7.7s%7lx%7lx %4x %4x %4x %4x %4x %4x %4x %4x %4x %5uK\n",
        proc_nr(rp),
        rp->p_name,
        (unsigned long) rp->p_reg.pc,
        (unsigned long) rp->p_reg.sp,
        rp->p_memmap[T].mem_vir, rp->p_memmap[T].mem_phys, rp->p_memmap[T].mem_len
    ,
        rp->p_memmap[D].mem_vir, rp->p_memmap[D].mem_phys, rp->p_memmap[D].mem_len
    ,
        rp->p_memmap[S].mem_vir, rp->p_memmap[S].mem_phys, rp->p_memmap[S].mem_len
    ,
        click_to_round_k(size));
}
if (rp == END_PROC_ADDR) rp = proc;
else printf("--more--\r");
oldrp = rp;
}

/*=====*
*                                     proc_name                                     *
*=====*/
PRIVATE char *proc_name(proc_nr)
int proc_nr;
{
    if (proc_nr == ANY) return "ANY";
    return cproc_addr(proc_nr)->p_name;
}

```

```

/* This file contains procedures to dump to PM' data structures.
 *
 * The entry points into this file are
 *   mproc_dmp:          display PM process table
 *
 * Created:
 *   May 11, 2005:      by Jorrit N. Herder
 */

#include "inc.h"
#include "../pm/mproc.h"
#include <timers.h>
#include <minix/config.h>
#include <minix/type.h>

PUBLIC struct mproc mproc[NR_PROCS];

/*=====
 *                               mproc_dmp
 *=====*/
PRIVATE char *flags_str(int flags)
{
    static char str[10];
    str[0] = (flags & WAITING) ? 'W' : '-';
    str[1] = (flags & ZOMBIE) ? 'Z' : '-';
    str[2] = (flags & PAUSED) ? 'P' : '-';
    str[3] = (flags & ALARM_ON) ? 'A' : '-';
    str[4] = (flags & TRACED) ? 'T' : '-';
    str[5] = (flags & STOPPED) ? 'S' : '-';
    str[6] = (flags & SIGSUSPENDED) ? 'U' : '-';
    str[7] = (flags & REPLY) ? 'R' : '-';
    str[8] = (flags & ONSWAP) ? 'O' : '-';
    str[9] = (flags & SWAPIN) ? 'I' : '-';
    str[10] = (flags & DONT_SWAP) ? 'D' : '-';
    str[11] = (flags & PRIV_PROC) ? 'P' : '-';
    str[12] = '\0';

    return str;
}

PUBLIC void mproc_dmp()
{
    struct mproc *mp;
    int i, n=0;
    static int prev_i = 0;

    printf("Process manager (PM) process table dump\n");

    getsysinfo(PM_PROC_NR, SI_PROC_TAB, mproc);

    printf("-process- -nr-prnt- -pid/ppid/grp- -uid--gid- -nice- -flags-----\n");
    for (i=prev_i; i<NR_PROCS; i++) {
        mp = &mproc[i];
        if (mp->mp_pid == 0 && i != PM_PROC_NR) continue;
        if (++n > 22) break;
        printf("%8s %4d%4d %4d%4d%4d  ",
            mp->mp_name, i, mp->mp_parent, mp->mp_pid, mproc[mp->mp_parent].mp_pid, m
p->mp_procgrp);
        printf("%d(%d) %d(%d) ",
            mp->mp_realuid, mp->mp_effuid, mp->mp_realgid, mp->mp_effgid);
        printf(" %3d %s ",
            mp->mp_nice, flags_str(mp->mp_flags));
        printf("\n");
    }
    if (i >= NR_PROCS) i = 0;
    else printf("---more---\r");
    prev_i = i;
}

/*=====
 *                               sigaction_dmp
 *=====*/
PUBLIC void sigaction_dmp()
{

```

```

struct mproc *mp;
int i, n=0;
static int prev_i = 0;
clock_t uptime;

printf("Process manager (PM) signal action dump\n");

getsysinfo(PM_PROC_NR, SI_PROC_TAB, mproc);
getuptime(&uptime);

printf("-process- -nr- --ignore- --catch- --block- -tomess- --pending- --alarm- --\n");
for (i=prev_i; i<NR_PROCS; i++) {
    mp = &mproc[i];
    if (mp->mp_pid == 0 && i != PM_PROC_NR) continue;
    if (++n > 22) break;
    printf("%8s %3d ", mp->mp_name, i);
    printf(" 0x%06x 0x%06x 0x%06x 0x%06x ",
           mp->mp_ignore, mp->mp_catch, mp->mp_sigmask, mp->mp_sig2mess);
    printf("0x%06x ", mp->mp_sigpending);
    if (mp->mp_flags & ALARM_ON) printf("%8u", mp->mp_timer.tmr_exp_time-uptime);
    else printf("    -");
    printf("\n");
}
if (i >= NR_PROCS) i = 0;
else printf("--more--\r");
prev_i = i;
}

/*=====
*                                     holes_dmp                                     *
*=====*/
PUBLIC void holes_dmp(void)
{
    static struct pm_mem_info pmi;
    int h;
    int largest_bytes = 0, total_bytes = 0;

    if(getsysinfo(PM_PROC_NR, SI_MEM_ALLOC, &pmi) != OK) {
        printf("Obtaining memory hole list failed.\n");
        return;
    }
    printf("Available memory stats\n");

    for(h = 0; h < _NR_HOLES; h++) {
        if(pmi.pmi_holes[h].h_base && pmi.pmi_holes[h].h_len) {
            int bytes;
            bytes = (pmi.pmi_holes[h].h_len << CLICK_SHIFT);
            printf("%08lx: %6d kB\n",
                   pmi.pmi_holes[h].h_base << CLICK_SHIFT, bytes / 1024);
            if(bytes > largest_bytes) largest_bytes = bytes;
            total_bytes += bytes;
        }
    }
    printf("\n"
           "Total memory free:   %7d kB\n"
           "Largest chunk:          %7d kB\n"
           "Uncontiguous rest:      %7d kB (%d%% of total free)\n"
           "Memory high watermark: %7d kB\n",
           total_bytes/1024,
           largest_bytes/1024,
           (total_bytes-largest_bytes)/1024,
           100*(total_bytes/100-largest_bytes/100)/total_bytes,
           (pmi.pmi_hi_watermark/1024 << CLICK_SHIFT));

    return;
}

```

```

/* This file contains procedures to dump RS data structures.
 *
 * The entry points into this file are
 *   rproc_dump:      display RS system process table
 *
 * Created:
 *   Oct 03, 2005:      by Jorrit N. Herder
 */

#include "inc.h"
#include "../rs/manager.h"

PUBLIC struct rproc rproc[NR_SYS_PROCS];

FORWARD _PROTOTYPE( char *s_flags_str, (int flags)
                    );

/*=====
 *                               rproc_dmp
 *=====*/
PUBLIC void rproc_dmp()
{
    struct rproc *rp;
    int i,j, n=0;
    static int prev_i=0;

    getsysinfo(RS_PROC_NR, SI_PROC_TAB, rproc);

    printf( "Reincarnation Server (RS) system process table dump\n" );
    printf( "-----proc---pid--flag--dev--T---checked-----alive--starts--backoff--command (argc)--\n" );
    for (i=prev_i; i<NR_SYS_PROCS; i++) {
        rp = &rproc[i];
        if (! rp->r_flags & RS_IN_USE) continue;
        if (++n > 22) break;
        printf( "%9d %5d %s %3d/%2d %3u %8u %8u %4dx %3d %s (%d)",
                rp->r_proc_nr_e, rp->r_pid,
                s_flags_str(rp->r_flags),
                rp->r_dev_nr, rp->r_dev_style,
                rp->r_period,
                rp->r_check_tm, rp->r_alive_tm,
                rp->r_restarts, rp->r_backoff,
                rp->r_cmd,
                rp->r_argc
            );
        printf("\n");
    }
    if (i >= NR_SYS_PROCS) i = 0;
    else printf( "--more--\r" );
    prev_i = i;
}

PRIVATE char *s_flags_str(int flags)
{
    static char str[5];
    str[0] = (flags & RS_IN_USE)      ? 'U' : '-';
    str[1] = (flags & RS_EXITING)     ? 'E' : '-';
    str[2] = '-';
    str[3] = '\0';

    return(str);
}

```



```
/* Global variables. */

/* Parameters needed to keep diagnostics at IS. */
#define DIAG_BUF_SIZE 1024
extern char diag_buf[DIAG_BUF_SIZE];    /* buffer for messages */
extern int diag_next;                   /* next index to be written */
extern int diag_size;                   /* size of all messages */

/* Flag to indicate system-wide panic. */
extern int sys_panic;                  /* if set, shutdown can be done */

/* The parameters of the call are kept here. */
extern message m_in;                   /* the input message itself */
extern message m_out;                  /* the output message used for reply */
extern int who_e;                      /* caller's proc number */
extern int callnr;                     /* system call number */
extern int dont_reply;                 /* normally 0; set to 1 to inhibit reply */
```

```
/* Header file for the system information server.
 *
 * Created:
 *   Jul 13, 2004           by Jorrit N. Herder
 */

#define _SYSTEM            1    /* get OK and negative error codes */
#define _MINIX             1    /* tell headers to include MINIX stuff */

#include <ansi.h>
#include <sys/types.h>
#include <limits.h>
#include <errno.h>

#include <minix/callnr.h>
#include <minix/config.h>
#include <minix/type.h>
#include <minix/const.h>
#include <minix/com.h>
#include <minix/syslib.h>
#include <minix/sysutil.h>
#include <minix/keymap.h>
#include <minix/bitmap.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>

#include "proto.h"
#include "glo.h"
```

```
/* Header file for the system information server.
 *
 * Created:
 *   Jul 13, 2004      by Jorrit N. Herder
 */

#define _SYSTEM          1    /* get OK and negative error codes */
#define _MINIX           1    /* tell headers to include MINIX stuff */

#include <ansi.h>
#include <sys/types.h>
#include <limits.h>
#include <errno.h>

#include <minix/callnr.h>
#include <minix/config.h>
#include <minix/type.h>
#include <minix/const.h>
#include <minix/com.h>
#include <minix/syslib.h>
#include <minix/sysutil.h>
#include <minix/keymap.h>
#include <minix/bitmap.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>

#include "proto.h"
#include "glo.h"
```

```

/* System Information Service.
 * This service handles the various debugging dumps, such as the process
 * table, so that these no longer directly touch kernel memory. Instead, the
 * system task is asked to copy some table in local memory.
 *
 * Created:
 *   Apr 29, 2004          by Jorrit N. Herder
 */

#include "inc.h"

/* Set debugging level to 0, 1, or 2 to see no, some, all debug output. */
#define DEBUG_LEVEL      1
#define DPRINTF          if (DEBUG_LEVEL > 0) printf

/* Allocate space for the global variables. */
message m_in;           /* the input message itself */
message m_out;          /* the output message used for reply */
int who_e;              /* caller's proc number */
int callnr;             /* system call number */

extern int errno;       /* error number set by system library */

/* Declare some local functions. */
FORWARD _PROTOTYPE(void init_server, (int argc, char **argv)      );
FORWARD _PROTOTYPE(void sig_handler, (void)                      );
FORWARD _PROTOTYPE(void exit_server, (void)                      );
FORWARD _PROTOTYPE(void get_work, (void)                         );
FORWARD _PROTOTYPE(void reply, (int whom, int result)             );

/*=====
 *                               main                               *
 *=====*/
PUBLIC int main(int argc, char **argv)
{
    /* This is the main routine of this service. The main loop consists of
     * three major activities: getting new work, processing the work, and
     * sending the reply. The loop never terminates, unless a panic occurs.
     */
    int result;
    sigset_t sigset;

    /* Initialize the server, then go to work. */
    init_server(argc, argv);

    /* Main loop - get work and do it, forever. */
    while (TRUE) {

        /* Wait for incoming message, sets 'callnr' and 'who'. */
        get_work();

        switch (callnr) {
        case SYS_SIG:
            printf("got SYS_SIG message\n");
            sigset = m_in.NOTIFY_ARG;
            for (result=0; result< _NSIG; result++) {
                if (sigismember(&sigset, result))
                    printf("signal %d found\n", result);
            }
            continue;
        case PROC_EVENT:
            sig_handler();
            continue;
        case FKEY_PRESSED:
            result = do_fkey_pressed(&m_in);
            break;
        case DEV_PING:
            notify(m_in.m_source);
            continue;
        default:
            report("IS", "warning, got illegal request from:", m_in.m_source);
            result = EINVAL;
        }
    }
}

```

```

    /* Finally send reply message, unless disabled. */
    if (result != EDONTREPLY) {
        reply(who_e, result);
    }
}
return(OK); /* shouldn't come here */
}

/*=====
*                               init_server                               *
*=====*/
PRIVATE void init_server(int argc, char **argv)
{
    /* Initialize the information service. */
    int fkeys, sfkeys;
    int i, s;
    struct sigaction sigact;

    /* Install signal handler. Ask PM to transform signal into message. */
    sigact.sa_handler = SIG_MESS;
    sigact.sa_mask = ~0; /* block all other signals */
    sigact.sa_flags = 0; /* default behaviour */
    if (sigaction(SIGTERM, &sigact, NULL) < 0)
        report("IS", "warning, sigaction() failed", errno);

    /* Set key mappings. IS takes all of F1-F12 and Shift+F1-F6. */
    fkeys = sfkeys = 0;
    for (i=1; i<=12; i++) bit_set(fkeys, i);
    for (i=1; i<= 8; i++) bit_set(sfkeys, i);
    if ((s=fkey_map(&fkeys, &sfkeys)) != OK)
        report("IS", "warning, fkey_map failed:", s);
}

/*=====
*                               sig_handler                               *
*=====*/
PRIVATE void sig_handler()
{
    sigset_t sigset;
    int sig;

    /* Try to obtain signal set from PM. */
    if (getsigset(&sigset) != 0) return;

    /* Check for known signals. */
    if (sigismember(&sigset, SIGTERM)) {
        exit_server();
    }
}

/*=====
*                               exit_server                               *
*=====*/
PRIVATE void exit_server()
{
    /* Shut down the information service. */
    int fkeys, sfkeys;
    int i, s;

    /* Release the function key mappings requested in init_server().
     * IS took all of F1-F12 and Shift+F1-F6.
     */
    fkeys = sfkeys = 0;
    for (i=1; i<=12; i++) bit_set(fkeys, i);
    for (i=1; i<= 7; i++) bit_set(sfkeys, i);
    if ((s=fkey_unmap(&fkeys, &sfkeys)) != OK)
        report("IS", "warning, unfkey_map failed:", s);

    /* Done. Now exit. */
    exit(0);
}

/*=====
*                               get_work                               *
*=====*/

```

```

*=====*/
PRIVATE void get_work()
{
    int status = 0;
    status = receive(ANY, &m_in);  /* this blocks until message arrives */
    if (OK != status)
        panic("IS", "failed to receive message!", status);
    who_e = m_in.m_source;        /* message arrived! set sender */
    callnr = m_in.m_type;        /* set function call number */
}

/*=====*
*                               *
*=====*/
PRIVATE void reply(who, result)
int who;                        /* destination */
int result;                     /* report result to replyee */
{
    int send_status;
    m_out.m_type = result;      /* build reply message */
    send_status = send(who, &m_out); /* send the message */
    if (OK != send_status)
        panic("IS", "unable to send reply!", send_status);
}

```

```
/* Function prototypes. */

/* main.c */
_PROTOTYPE( int main, (int argc, char **argv) ) ;

/* dmp.c */
_PROTOTYPE( int do_fkey_pressed, (message *m) ) ;
_PROTOTYPE( void mapping_dmp, (void) ) ;

/* dmp_kernel.c */
_PROTOTYPE( void proctab_dmp, (void) ) ;
_PROTOTYPE( void memmap_dmp, (void) ) ;
_PROTOTYPE( void privileges_dmp, (void) ) ;
_PROTOTYPE( void sendmask_dmp, (void) ) ;
_PROTOTYPE( void image_dmp, (void) ) ;
_PROTOTYPE( void irqtab_dmp, (void) ) ;
_PROTOTYPE( void kmessages_dmp, (void) ) ;
_PROTOTYPE( void sched_dmp, (void) ) ;
_PROTOTYPE( void monparams_dmp, (void) ) ;
_PROTOTYPE( void kenv_dmp, (void) ) ;
_PROTOTYPE( void timing_dmp, (void) ) ;

/* dmp_pm.c */
_PROTOTYPE( void mproc_dmp, (void) ) ;
_PROTOTYPE( void sigaction_dmp, (void) ) ;
_PROTOTYPE( void holes_dmp, (void) ) ;

/* dmp_fs.c */
_PROTOTYPE( void dtab_dmp, (void) ) ;
_PROTOTYPE( void fproc_dmp, (void) ) ;

/* dmp_rs.c */
_PROTOTYPE( void rproc_dmp, (void) ) ;

/* dmp_ds.c */
_PROTOTYPE( void data_store_dmp, (void) ) ;
```

```
# Makefile for Process Manager (PM)
SERVER = pm

# directories
u = /usr
i = $u/include
s = $i/sys
h = $i/minix
k = $u/src/kernel

# programs, flags, etc.
CC =      exec cc
CFLAGS = -I$i
LDFLAGS = -i

OBJ =      main.o forkexit.o break.o exec.o time.o timers.o \
           signal.o alloc.o utility.o table.o trace.o getset.o misc.o

# build local binary
all build: $(SERVER)
$(SERVER): $(OBJ)
           $(CC) -o $@ $(LDFLAGS) $(OBJ) -lsys -lsysutil -ltimers
           install -S 256w $@

# install with other servers
install:      /usr/sbin/$(SERVER)
/usr/sbin/$(SERVER): $(SERVER)
           install -o root -cs $? $@

# clean up local files
clean:
           rm -f $(SERVER) *.o *.bak

depend:
           /usr/bin/mkdep "$(CC) -E $(CPPFLAGS)" *.c > .depend

# Include generated dependencies.
include .depend
```



```

/* This file is concerned with allocating and freeing arbitrary-size blocks of
 * physical memory on behalf of the FORK and EXEC system calls. The key data
 * structure used is the hole table, which maintains a list of holes in memory.
 * It is kept sorted in order of increasing memory address. The addresses
 * it contains refers to physical memory, starting at absolute address 0
 * (i.e., they are not relative to the start of PM). During system
 * initialization, that part of memory containing the interrupt vectors,
 * kernel, and PM are "allocated" to mark them as not available and to
 * remove them from the hole list.
 *
 * The entry points into this file are:
 *   alloc_mem: allocate a given sized chunk of memory
 *   free_mem:  release a previously allocated chunk of memory
 *   mem_init:  initialize the tables when PM start up
 *   max_hole:  returns the largest hole currently available
 *   mem_holes_copy: for outsiders who want a copy of the hole-list
 */

#include "pm.h"
#include <minix/com.h>
#include <minix/callnr.h>
#include <minix/type.h>
#include <minix/config.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include "mproc.h"
#include "../kernel/const.h"
#include "../kernel/config.h"
#include "../kernel/type.h"

#define NIL_HOLE (struct hole *) 0

PRIVATE struct hole hole[_NR_HOLES];
PRIVATE u32_t high_watermark = 0;

PRIVATE struct hole *hole_head; /* pointer to first hole */
PRIVATE struct hole *free_slots; /* ptr to list of unused table slots */
#if ENABLE_SWAP
PRIVATE int swap_fd = -1; /* file descriptor of open swap file/device */
PRIVATE u32_t swap_offset; /* offset to start of swap area on swap file */
PRIVATE phys_clicks swap_base; /* memory offset chosen as swap base */
PRIVATE phys_clicks swap_maxsize; /* maximum amount of swap "memory" possible */
PRIVATE struct mproc *in_queue; /* queue of processes wanting to swap in */
PRIVATE struct mproc *outswap = &mproc[0]; /* outswap candidate? */
#else /* ! ENABLE_SWAP */
#define swap_base ((phys_clicks) -1)
#endif /* ENABLE_SWAP */

FORWARD _PROTOTYPE( void del_slot, (struct hole *prev_ptr, struct hole *hp) );
FORWARD _PROTOTYPE( void merge, (struct hole *hp) );
#if ENABLE_SWAP
FORWARD _PROTOTYPE( int swap_out, (void) );
#else
#define swap_out() (0)
#endif

/*=====
 *                               alloc_mem                               *
 *=====*/
PUBLIC phys_clicks alloc_mem(clicks)
phys_clicks clicks; /* amount of memory requested */
{
/* Allocate a block of memory from the free list using first fit. The block
 * consists of a sequence of contiguous bytes, whose length in clicks is
 * given by 'clicks'. A pointer to the block is returned. The block is
 * always on a click boundary. This procedure is called when memory is
 * needed for FORK or EXEC. Swap other processes out if needed.
 */
register struct hole *hp, *prev_ptr;
phys_clicks old_base;

do {
    prev_ptr = NIL_HOLE;

```

```

    hp = hole_head;
    while (hp != NIL_HOLE && hp->h_base < swap_base) {
        if (hp->h_len >= clicks) {
            /* We found a hole that is big enough. Use it. */
            old_base = hp->h_base; /* remember where it started */
            hp->h_base += clicks; /* bite a piece off */
            hp->h_len -= clicks; /* ditto */

            /* Remember new high watermark of used memory. */
            if (hp->h_base > high_watermark)
                high_watermark = hp->h_base;

            /* Delete the hole if used up completely. */
            if (hp->h_len == 0) del_slot(prev_ptr, hp);

            /* Return the start address of the acquired block. */
            return(old_base);
        }

        prev_ptr = hp;
        hp = hp->h_next;
    }
} while (swap_out()); /* try to swap some other process out */
return(NO_MEM);
}

/*=====
*                                     free_mem                                     *
*=====*/
PUBLIC void free_mem(base, clicks)
phys_clicks base; /* base address of block to free */
phys_clicks clicks; /* number of clicks to free */
{
    /* Return a block of free memory to the hole list. The parameters tell where
    * the block starts in physical memory and how big it is. The block is added
    * to the hole list. If it is contiguous with an existing hole on either end,
    * it is merged with the hole or holes.
    */
    register struct hole *hp, *new_ptr, *prev_ptr;

    if (clicks == 0) return;
    if ( (new_ptr = free_slots) == NIL_HOLE)
        panic(__FILE__, "hole table full", NO_NUM);
    new_ptr->h_base = base;
    new_ptr->h_len = clicks;
    free_slots = new_ptr->h_next;
    hp = hole_head;

    /* If this block's address is numerically less than the lowest hole currently
    * available, or if no holes are currently available, put this hole on the
    * front of the hole list.
    */
    if (hp == NIL_HOLE || base <= hp->h_base) {
        /* Block to be freed goes on front of the hole list. */
        new_ptr->h_next = hp;
        hole_head = new_ptr;
        merge(new_ptr);
        return;
    }

    /* Block to be returned does not go on front of hole list. */
    prev_ptr = NIL_HOLE;
    while (hp != NIL_HOLE && base > hp->h_base) {
        prev_ptr = hp;
        hp = hp->h_next;
    }

    /* We found where it goes. Insert block after 'prev_ptr'. */
    new_ptr->h_next = prev_ptr->h_next;
    prev_ptr->h_next = new_ptr;
    merge(prev_ptr); /* sequence is 'prev_ptr', 'new_ptr', 'hp' */
}

/*=====

```

```

*                                     del_slot                                     *
*=====*/
PRIVATE void del_slot(prev_ptr, hp)
/* pointer to hole entry just ahead of 'hp' */
register struct hole *prev_ptr;
/* pointer to hole entry to be removed */
register struct hole *hp;
{
/* Remove an entry from the hole list. This procedure is called when a
 * request to allocate memory removes a hole in its entirety, thus reducing
 * the numbers of holes in memory, and requiring the elimination of one
 * entry in the hole list.
 */
    if (hp == hole_head)
        hole_head = hp->h_next;
    else
        prev_ptr->h_next = hp->h_next;

    hp->h_next = free_slots;
    hp->h_base = hp->h_len = 0;
    free_slots = hp;
}

/*=====*
*                                     merge                                     *
*=====*/
PRIVATE void merge(hp)
register struct hole *hp;          /* ptr to hole to merge with its successors */
{
/* Check for contiguous holes and merge any found. Contiguous holes can occur
 * when a block of memory is freed, and it happens to abut another hole on
 * either or both ends. The pointer 'hp' points to the first of a series of
 * three holes that can potentially all be merged together.
 */
    register struct hole *next_ptr;

    /* If 'hp' points to the last hole, no merging is possible. If it does not,
     * try to absorb its successor into it and free the successor's table entry.
     */
    if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
    if (hp->h_base + hp->h_len == next_ptr->h_base) {
        hp->h_len += next_ptr->h_len; /* first one gets second one's mem */
        del_slot(hp, next_ptr);
    } else {
        hp = next_ptr;
    }

    /* If 'hp' now points to the last hole, return; otherwise, try to absorb its
     * successor into it.
     */
    if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
    if (hp->h_base + hp->h_len == next_ptr->h_base) {
        hp->h_len += next_ptr->h_len;
        del_slot(hp, next_ptr);
    }
}

/*=====*
*                                     mem_init                                    *
*=====*/
PUBLIC void mem_init(chunks, free)
struct memory *chunks;          /* list of free memory chunks */
phys_clicks *free;             /* memory size summaries */
{
/* Initialize hole lists. There are two lists: 'hole_head' points to a linked
 * list of all the holes (unused memory) in the system; 'free_slots' points to
 * a linked list of table entries that are not in use. Initially, the former
 * list has one entry for each chunk of physical memory, and the second
 * list links together the remaining table slots. As memory becomes more
 * fragmented in the course of time (i.e., the initial big holes break up into
 * smaller holes), new table slots are needed to represent them. These slots
 * are taken from the list headed by 'free_slots'.
 */
    int i;

```

```

register struct hole *hp;

/* Put all holes on the free list. */
for (hp = &hole[0]; hp < &hole[_NR_HOLES]; hp++) {
    hp->h_next = hp + 1;
    hp->h_base = hp->h_len = 0;
}
hole[_NR_HOLES-1].h_next = NIL_HOLE;
hole_head = NIL_HOLE;
free_slots = &hole[0];

/* Use the chunks of physical memory to allocate holes. */
*free = 0;
for (i=NR_MEMS-1; i>=0; i--) {
    if (chunks[i].size > 0) {
        free_mem(chunks[i].base, chunks[i].size);
        *free += chunks[i].size;
#if ENABLE_SWAP
        if (swap_base < chunks[i].base + chunks[i].size)
            swap_base = chunks[i].base + chunks[i].size;
#endif
    }
}

#if ENABLE_SWAP
/* The swap area is represented as a hole above and separate of regular
 * memory. A hole at the size of the swap file is allocated on "swapon".
 */
swap_base++; /* make separate */
swap_maxsize = 0 - swap_base; /* maximum we can possibly use */
#endif
}

/*=====
 *                               mem_holes_copy                               *
 *=====*/
PUBLIC int mem_holes_copy(struct hole *holecopies, size_t *bytes, u32_t *hi)
{
    if(*bytes < sizeof(hole)) return ENOSPC;
    memcpy(holecopies, hole, sizeof(hole));
    *bytes = sizeof(hole);
    *hi = high_watermark;
    return OK;
}

#if ENABLE_SWAP
/*=====
 *                               swap_on                               *
 *=====*/
PUBLIC int swap_on(file, offset, size)
char *file; /* file to swap on */
u32_t offset, size; /* area on swap file to use */
{
    /* Turn swapping on. */

    if (swap_fd != -1) return(EBUSY); /* already have swap? */

    tell_fs(CHDIR, who_e, FALSE, 0); /* be like the caller for open() */
    if ((swap_fd = open(file, O_RDWR)) < 0) return(-errno);
    swap_offset = offset;
    size >>= CLICK_SHIFT;
    if (size > swap_maxsize) size = swap_maxsize;
    if (size > 0) free_mem(swap_base, (phys_clicks) size);
    return(OK);
}

/*=====
 *                               swap_off                               *
 *=====*/
PUBLIC int swap_off()
{
    /* Turn swapping off. */
    struct mproc *rmp;
    struct hole *hp, *prev_ptr;

```

```

if (swap_fd == -1) return(OK);          /* can't turn off what isn't on */

/* Put all swapped out processes on the inswap queue and swap in. */
for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
    if (rmp->mp_flags & ONSWAP) swap_inqueue(rmp);
}
swap_in();

/* All in memory? */
for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
    if (rmp->mp_flags & ONSWAP) return(ENOMEM);
}

/* Yes. Remove the swap hole and close the swap file descriptor. */
for (hp = hole_head; hp != NIL_HOLE; prev_ptr = hp, hp = hp->h_next) {
    if (hp->h_base >= swap_base) {
        del_slot(prev_ptr, hp);
        hp = hole_head;
    }
}
close(swap_fd);
swap_fd = -1;
return(OK);
}

/*=====
*                               swap_inqueue                               *
*=====*/
PUBLIC void swap_inqueue(rmp)
register struct mproc *rmp;          /* process to add to the queue */
{
    /* Put a swapped out process on the queue of processes to be swapped in. This
     * happens when such a process gets a signal, or if a reply message must be
     * sent, like when a process doing a wait() has a child that exits.
     */
    struct mproc **pmp;

    if (rmp->mp_flags & SWAPIN) return; /* already queued */

    for (pmp = &in_queue; *pmp != NULL; pmp = &(*pmp)->mp_swapq) {}
    *pmp = rmp;
    rmp->mp_swapq = NULL;
    rmp->mp_flags |= SWAPIN;
}

/*=====
*                               swap_in                               *
*=====*/
PUBLIC void swap_in()
{
    /* Try to swap in a process on the inswap queue. We want to send it a message,
     * interrupt it, or something.
     */
    struct mproc **pmp, *rmp;
    phys_clicks old_base, new_base, size;
    off_t off;
    int proc_nr;

    pmp = &in_queue;
    while ((rmp = *pmp) != NULL) {
        proc_nr = (rmp - mproc);
        size = rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len
            - rmp->mp_seg[D].mem_vir;

        if (!(rmp->mp_flags & SWAPIN)) {
            /* Guess it got killed. (Queue is cleaned here.) */
            *pmp = rmp->mp_swapq;
            continue;
        } else
        if ((new_base = alloc_mem(size)) == NO_MEM) {
            /* No memory for this one, try the next. */
            pmp = &rmp->mp_swapq;
        }
    }
}

```

```

    } else {
        /* We've found memory. Update map and swap in. */
        old_base = rmp->mp_seg[D].mem_phys;
        rmp->mp_seg[D].mem_phys = new_base;
        rmp->mp_seg[S].mem_phys = rmp->mp_seg[D].mem_phys +
            (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
        sys_newmap(rmp->mp_endpoint, rmp->mp_seg);
        off = swap_offset + ((off_t) (old_base-swap_base)<<CLICK_SHIFT);
        lseek(swap_fd, off, SEEK_SET);
        rw_seg(0, swap_fd, rmp->mp_endpoint, D, (phys_bytes)size << CLICK_SHIFT);
        free_mem(old_base, size);
        rmp->mp_flags &= ~(ONSWAP|SWAPIN);
        *pmp = rmp->mp_swapq;
        check_pending(rmp);      /* a signal may have waked this one */
    }
}
}

/*=====
 *                               swap_out                               *
 *=====*/
PRIVATE int swap_out()
{
    /* Try to find a process that can be swapped out. Candidates are those blocked
    * on a system call that PM handles, like wait(), pause() or sigsuspend().
    */
    struct mproc *rmp;
    struct hole *hp, *prev_ptr;
    phys_clicks old_base, new_base, size;
    off_t off;
    int proc_nr;

    rmp = outswap;
    do {
        if (++rmp == &mproc[NR_PROCS]) rmp = &mproc[0];

        /* A candidate? */
        if (!(rmp->mp_flags & (PAUSED | WAITING | SIGSUSPENDED))) continue;

        /* Already on swap or otherwise to be avoided? */
        if (rmp->mp_flags & (DONT_SWAP | TRACED | REPLY | ONSWAP)) continue;

        /* Got one, find a swap hole and swap it out. */
        proc_nr = (rmp - mproc);
        size = rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len
            - rmp->mp_seg[D].mem_vir;

        prev_ptr = NIL_HOLE;
        for (hp = hole_head; hp != NIL_HOLE; prev_ptr = hp, hp = hp->h_next) {
            if (hp->h_base >= swap_base && hp->h_len >= size) break;
        }
        if (hp == NIL_HOLE) continue;      /* oops, not enough swapspace */
        new_base = hp->h_base;
        hp->h_base += size;
        hp->h_len -= size;
        if (hp->h_len == 0) del_slot(prev_ptr, hp);

        off = swap_offset + ((off_t) (new_base - swap_base) << CLICK_SHIFT);
        lseek(swap_fd, off, SEEK_SET);
        rw_seg(1, swap_fd, rmp->mp_endpoint, D, (phys_bytes)size << CLICK_SHIFT);
        old_base = rmp->mp_seg[D].mem_phys;
        rmp->mp_seg[D].mem_phys = new_base;
        rmp->mp_seg[S].mem_phys = rmp->mp_seg[D].mem_phys +
            (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
        sys_newmap(rmp->mp_endpoint, rmp->mp_seg);
        free_mem(old_base, size);
        rmp->mp_flags |= ONSWAP;

        outswap = rmp;      /* next time start here */
        return(TRUE);
    } while (rmp != outswap);

    return(FALSE);      /* no candidate found */
}

```

```
#endif /* SWAP */
```

```

/* The MINIX model of memory allocation reserves a fixed amount of memory for
 * the combined text, data, and stack segments. The amount used for a child
 * process created by FORK is the same as the parent had. If the child does
 * an EXEC later, the new size is taken from the header of the file EXEC'ed.
 *
 * The layout in memory consists of the text segment, followed by the data
 * segment, followed by a gap (unused memory), followed by the stack segment.
 * The data segment grows upward and the stack grows downward, so each can
 * take memory from the gap. If they meet, the process must be killed. The
 * procedures in this file deal with the growth of the data and stack segments.
 *
 * The entry points into this file are:
 *   do_brk:      BRK/SBRK system calls to grow or shrink the data segment
 *   adjust:      see if a proposed segment adjustment is allowed
 *   size_ok:     see if the segment sizes are feasible (i86 only)
 */

#include "pm.h"
#include <signal.h>
#include "mproc.h"
#include "param.h"

#define DATA_CHANGED      1    /* flag value when data segment size changed */
#define STACK_CHANGED      2    /* flag value when stack size changed */

/*=====
 *                               do_brk                               *
 *=====*/
PUBLIC int do_brk()
{
    /* Perform the brk(addr) system call.
     *
     * The call is complicated by the fact that on some machines (e.g., 8088),
     * the stack pointer can grow beyond the base of the stack segment without
     * anybody noticing it.
     * The parameter, 'addr' is the new virtual address in D space.
     */

    register struct mproc *rmp;
    int r;
    vir_bytes v, new_sp;
    vir_clicks new_clicks;

    rmp = mp;
    v = (vir_bytes) m_in.addr;
    new_clicks = (vir_clicks) ( ((long) v + CLICK_SIZE - 1) >> CLICK_SHIFT);
    if (new_clicks < rmp->mp_seg[D].mem_vir) {
        rmp->mp_reply.reply_ptr = (char *) -1;
        return(ENOMEM);
    }
    new_clicks -= rmp->mp_seg[D].mem_vir;
    if ((r=get_stack_ptr(who_e, &new_sp)) != OK) /* ask kernel for sp value */
        panic(__FILE__, "couldn't get stack pointer", r);
    r = adjust(rmp, new_clicks, new_sp);
    rmp->mp_reply.reply_ptr = (r == OK ? m_in.addr : (char *) -1);
    return(r); /* return new address or -1 */
}

/*=====
 *                               adjust                               *
 *=====*/
PUBLIC int adjust(rmp, data_clicks, sp)
register struct mproc *rmp; /* whose memory is being adjusted? */
vir_clicks data_clicks; /* how big is data segment to become? */
vir_bytes sp; /* new value of sp */
{
    /* See if data and stack segments can coexist, adjusting them if need be.
     * Memory is never allocated or freed. Instead it is added or removed from the
     * gap between data segment and stack segment. If the gap size becomes
     * negative, the adjustment of data or stack fails and ENOMEM is returned.
     */

    register struct mem_map *mem_sp, *mem_dp;
    vir_clicks sp_click, gap_base, lower, old_clicks;

```



```

int changed, r, ft;
long base_of_stack, delta;    /* longs avoid certain problems */

mem_dp = &rmp->mp_seg[D];      /* pointer to data segment map */
mem_sp = &rmp->mp_seg[S];      /* pointer to stack segment map */
changed = 0;                  /* set when either segment changed */

/* See if stack size has gone negative (i.e., sp too close to 0xFFFF...) */
base_of_stack = (long) mem_sp->mem_vir + (long) mem_sp->mem_len;
sp_click = sp >> CLICK_SHIFT; /* click containing sp */
if (sp_click >= base_of_stack)
{
    return(ENOMEM); /* sp too high */
}

/* Compute size of gap between stack and data segments. */
delta = (long) mem_sp->mem_vir - (long) sp_click;
lower = (delta > 0 ? sp_click : mem_sp->mem_vir);

/* Add a safety margin for future stack growth. Impossible to do right. */
#define SAFETY_BYTES (384 * sizeof(char *))
#define SAFETY_CLICKS ((SAFETY_BYTES + CLICK_SIZE - 1) / CLICK_SIZE)
gap_base = mem_dp->mem_vir + data_clicks + SAFETY_CLICKS;
if (lower < gap_base)
{
    return(ENOMEM); /* data and stack collided */
}

/* Update data length (but not data origin) on behalf of brk() system call. */
old_clicks = mem_dp->mem_len;
if (data_clicks != mem_dp->mem_len) {
    mem_dp->mem_len = data_clicks;
    changed |= DATA_CHANGED;
}

/* Update stack length and origin due to change in stack pointer. */
if (delta > 0) {
    mem_sp->mem_vir -= delta;
    mem_sp->mem_phys -= delta;
    mem_sp->mem_len += delta;
    changed |= STACK_CHANGED;
}

/* Do the new data and stack segment sizes fit in the address space? */
ft = (rmp->mp_flags & SEPARATE);
#if CHIP == INTEL && _WORD_SIZE == 2
r = size_ok(ft, rmp->mp_seg[T].mem_len, rmp->mp_seg[D].mem_len,
    rmp->mp_seg[S].mem_len, rmp->mp_seg[D].mem_vir, rmp->mp_seg[S].mem_vir);
#else
r = (rmp->mp_seg[D].mem_vir + rmp->mp_seg[D].mem_len >
    rmp->mp_seg[S].mem_vir) ? ENOMEM : OK;
#endif
if (r == OK) {
    int r2;
    if (changed && (r2=sys_newmap(rmp->mp_endpoint, rmp->mp_seg)) != OK)
        panic(__FILE__, "couldn't sys_newmap in adjust", r2);
    return(OK);
}

/* New sizes don't fit or require too many page/segment registers. Restore.*/
if (changed & DATA_CHANGED) mem_dp->mem_len = old_clicks;
if (changed & STACK_CHANGED) {
    mem_sp->mem_vir += delta;
    mem_sp->mem_phys += delta;
    mem_sp->mem_len -= delta;
}
return(ENOMEM);
}

#if CHIP == INTEL && _WORD_SIZE == 2
/*=====
*                                     size_ok                                     *
*=====*/
PUBLIC int size_ok(file_type, tc, dc, sc, dvir, s_vir)

```

```
int file_type;          /* SEPARATE or 0 */
vir_clicks tc;          /* text size in clicks */
vir_clicks dc;          /* data size in clicks */
vir_clicks sc;          /* stack size in clicks */
vir_clicks dvir;        /* virtual address for start of data seg */
vir_clicks s_vir;       /* virtual address for start of stack seg */
{
/* Check to see if the sizes are feasible and enough segmentation registers
 * exist.  On a machine with eight 8K pages, text, data, stack sizes of
 * (32K, 16K, 16K) will fit, but (33K, 17K, 13K) will not, even though the
 * former is bigger (64K) than the latter (63K).  Even on the 8088 this test
 * is needed, since the data and stack may not exceed 4096 clicks.
 * Note this is not used for 32-bit Intel Minix, the test is done in-line.
 */

    int pt, pd, ps;      /* segment sizes in pages */

    pt = ( (tc << CLICK_SHIFT) + PAGE_SIZE - 1)/PAGE_SIZE;
    pd = ( (dc << CLICK_SHIFT) + PAGE_SIZE - 1)/PAGE_SIZE;
    ps = ( (sc << CLICK_SHIFT) + PAGE_SIZE - 1)/PAGE_SIZE;

    if (file_type == SEPARATE) {
        if (pt > MAX_PAGES || pd + ps > MAX_PAGES) return(ENOMEM);
    } else {
        if (pt + pd + ps > MAX_PAGES) return(ENOMEM);
    }

    if (dvir + dc > s_vir) return(ENOMEM);

    return(OK);
}
#endif
```

```
/* Constants used by the Process Manager. */

#define NO_MEM ((phys_clicks) 0) /* returned by alloc_mem() with mem is up */

#if (CHIP == INTEL && _WORD_SIZE == 2)
/* These definitions are used in size_ok and are not needed for 386.
 * The 386 segment granularity is 1 for segments smaller than 1M and 4096
 * above that.
 */
#define PAGE_SIZE          16 /* how many bytes in a page (s.b.HCLICK_SIZE)*/
#define MAX_PAGES          4096 /* how many pages in the virtual addr space */
#endif

#define NR_PIDS            30000 /* process ids range from 0 to NR_PIDS-1.
 * (magic constant: some old applications use
 * a 'short' instead of pid_t.)
 */

#define PM_PID              0 /* PM's process id number */
#define INIT_PID            1 /* INIT's process id number */

#define DUMPED              0200 /* bit set in status when core dumped */
```

```

/* This file handles the EXEC system call.  It performs the work as follows:
 *   - see if the permissions allow the file to be executed
 *   - read the header and extract the sizes
 *   - fetch the initial args and environment from the user space
 *   - allocate the memory for the new process
 *   - copy the initial stack from PM to the process
 *   - read in the text and data segments and copy to the process
 *   - take care of setuid and setgid bits
 *   - fix up 'mproc' table
 *   - tell kernel about EXEC
 *   - save offset to initial argc (for ps)
 *
 * The entry points into this file are:
 *   do_exec:    perform the EXEC system call
 *   exec_newmem: allocate new memory map for a process that tries to exec
 *   do_execrestart: finish the special exec call for RS
 *   exec_restart: finish a regular exec call
 *   find_share: find a process whose text segment can be shared
 */

#include "pm.h"
#include <sys/stat.h>
#include <minix/callnr.h>
#include <minix/endpoint.h>
#include <minix/com.h>
#include <a.out.h>
#include <signal.h>
#include <string.h>
#include "mproc.h"
#include "param.h"

FORWARD _PROTOTYPE( int new_mem, (struct mproc *rmp, struct mproc *sh_mp,
    vir_bytes text_bytes, vir_bytes data_bytes, vir_bytes bss_bytes,
    vir_bytes stk_bytes, phys_bytes tot_bytes) );

#define ESCRIPT (-2000) /* Returned by read_header for a #! script. */
#define PTRSIZE sizeof(char *) /* Size of pointers in argv[] and envp[]. */

/*=====
 *                               do_exec
 *=====*/
PUBLIC int do_exec()
{
    int r;

    /* Save parameters */
    mp->mp_exec_path= m_in.exec_name;
    mp->mp_exec_path_len= m_in.exec_len;
    mp->mp_exec_frame= m_in.stack_ptr;
    mp->mp_exec_frame_len= m_in.stack_bytes;

    /* Forward call to FS */
    if (mp->mp_fs_call != PM_IDLE)
    {
        panic(__FILE__, "do_exec: not idle", mp->mp_fs_call);
    }
    mp->mp_fs_call= PM_EXEC;
    r= notify(FS_PROC_NR);
    if (r != OK)
        panic(__FILE__, "do_getset: unable to notify FS", r);

    /* Do not reply */
    return SUSPEND;
}

/*=====
 *                               exec_newmem
 *=====*/
PUBLIC int exec_newmem()
{
    int r, proc_e, proc_n, allow_setuid;
    vir_bytes stack_top;
    vir_clicks tc, dc, sc, totc, dvir, s_vir;

```

```

struct mproc *rmp, *sh_mp;
char *ptr;
struct exec_newmem args;

if (who_e != FS_PROC_NR && who_e != RS_PROC_NR)
    return EPERM;

proc_e= m_in.EXC_NM_PROC;
if (pm_isokendpt(proc_e, &proc_n) != OK)
{
    panic(__FILE__, "exec_newmem: got bad endpoint",
          proc_e);
}
rmp= &mproc[proc_n];

ptr= m_in.EXC_NM_PTR;
r= sys_datacopy(who_e, (vir_bytes)ptr,
                SELF, (vir_bytes)&args, sizeof(args));
if (r != OK)
    panic(__FILE__, "exec_newmem: sys_datacopy failed", r);

/* Check to see if segment sizes are feasible. */
tc = ((unsigned long) args.text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
dc = (args.data_bytes+args.bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
totc = (args.tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
sc = (args.args_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
if (dc >= totc) return(ENOEXEC); /* stack must be at least 1 click */

dvir = (args.sep_id ? 0 : tc);
s_vir = dvir + (totc - sc);
#if (CHIP == INTEL && _WORD_SIZE == 2)
    r = size_ok(*ft, tc, dc, sc, dvir, s_vir);
#else
    r = (dvir + dc > s_vir) ? ENOMEM : OK;
#endif
if (r != OK)
    return r;

/* Can the process' text be shared with that of one already running? */
sh_mp = find_share(rmp, args.st_ino, args.st_dev, args.st_ctime);

/* Allocate new memory and release old memory. Fix map and tell
 * kernel.
 */
r = new_mem(rmp, sh_mp, args.text_bytes, args.data_bytes,
            args.bss_bytes, args.args_bytes, args.tot_bytes);
if (r != OK) return(r);

rmp->mp_flags |= PARTIAL_EXEC; /* Kill process if something goes
                                * wrong after this point.
                                */

/* Save file identification to allow it to be shared. */
rmp->mp_ino = args.st_ino;
rmp->mp_dev = args.st_dev;
rmp->mp_ctime = args.st_ctime;

stack_top= ((vir_bytes)rmp->mp_seg[S].mem_vir << CLICK_SHIFT) +
            ((vir_bytes)rmp->mp_seg[S].mem_len << CLICK_SHIFT);

/* Save offset to initial argc (for ps) */
rmp->mp_procars = stack_top - args.args_bytes;

/* set/clear separate I&D flag */
if (args.sep_id)
    rmp->mp_flags |= SEPARATE;
else
    rmp->mp_flags &= ~SEPARATE;

allow_setuid= 0; /* Do not allow setuid execution */
if ((rmp->mp_flags & TRACED) == 0) {
    /* Okay, setuid execution is allowed */
    allow_setuid= 1;
    rmp->mp_effuid = args.new_uid;
}

```

```

        rmp->mp_effgid = args.new_gid;
    }

    /* System will save command line for debugging, ps(1) output, etc. */
    strncpy(rmp->mp_name, args.progname, PROC_NAME_LEN-1);
    rmp->mp_name[PROC_NAME_LEN-1] = '\0';

    mp->mp_reply.reply_res2= stack_top;
    mp->mp_reply.reply_res3= 0;
    if (!sh_mp) /* Load text if sh_mp = NULL */
        mp->mp_reply.reply_res3 |= EXC_NM_RF_LOAD_TEXT;
    if (allow_setuid)
        mp->mp_reply.reply_res3 |= EXC_NM_RF_ALLOW_SETUID;

    return OK;
}

/*=====
 *                               do_execrestart                               *
 *=====*/
PUBLIC int do_execrestart()
{
    int proc_e, proc_n, result;
    struct mproc *rmp;

    if (who_e != RS_PROC_NR)
        return EPERM;

    proc_e= m_in.EXC_RS_PROC;
    if (pm_isokendpt(proc_e, &proc_n) != OK)
    {
        panic(__FILE__, "do_execrestart: got bad endpoint",
              proc_e);
    }
    rmp= &mproc[proc_n];
    result= m_in.EXC_RS_RESULT;

    exec_restart(rmp, result);

    return OK;
}

/*=====
 *                               exec_restart                               *
 *=====*/
PUBLIC void exec_restart(rmp, result)
struct mproc *rmp;
int result;
{
    int r, sn;
    vir_bytes pc;
    char *new_sp;

    if (result != OK)
    {
        if (rmp->mp_flags & PARTIAL_EXEC)
        {
            printf("partial exec; killing process\n");

            /* Use SIGILL signal that something went wrong */
            rmp->mp_sigstatus = SIGILL;
            pm_exit(rmp, 0, FALSE /*!for_trace*/);
            return;
        }
        setreply(rmp-mproc, result);
        return;
    }

    rmp->mp_flags &= ~PARTIAL_EXEC;

    /* Fix 'mproc' fields, tell kernel that exec is done, reset caught
     * sigs.

```

```

    */
    for (sn = 1; sn <= _NSIG; sn++) {
        if (sigismember(&rmp->mp_catch, sn)) {
            sigdelset(&rmp->mp_catch, sn);
            rmp->mp_sigact[sn].sa_handler = SIG_DFL;
            sigemptyset(&rmp->mp_sigact[sn].sa_mask);
        }
    }

    new_sp= (char *)rmp->mp_procargs;
    pc= 0; /* for now */
    r= sys_exec(rmp->mp_endpoint, new_sp, rmp->mp_name, pc);
    if (r != OK) panic(__FILE__, "sys_exec failed", r);

    /* Cause a signal if this process is traced. */
    if (rmp->mp_flags & TRACED) check_sig(rmp->mp_pid, SIGTRAP);
}

/*=====
*
* find_share
*=====*/
PUBLIC struct mproc *find_share(mp_ign, ino, dev, ctime)
struct mproc *mp_ign; /* process that should not be looked at */
ino_t ino; /* parameters that uniquely identify a file */
dev_t dev;
time_t ctime;
{
    /* Look for a process that is the file <ino, dev, ctime> in execution. Don't
    * accidentally "find" mp_ign, because it is the process on whose behalf this
    * call is made.
    */
    struct mproc *sh_mp;
    for (sh_mp = &mproc[0]; sh_mp < &mproc[NR_PROCS]; sh_mp++) {

        if (!(sh_mp->mp_flags & SEPARATE)) continue;
        if (sh_mp == mp_ign) continue;
        if (sh_mp->mp_ino != ino) continue;
        if (sh_mp->mp_dev != dev) continue;
        if (sh_mp->mp_ctime != ctime) continue;
        return sh_mp;
    }
    return(NULL);
}

/*=====
*
* new_mem
*=====*/
PRIVATE int new_mem(rmp, sh_mp, text_bytes, data_bytes,
    bss_bytes, stk_bytes, tot_bytes)
struct mproc *rmp; /* process to get a new memory map */
struct mproc *sh_mp; /* text can be shared with this process */
vir_bytes text_bytes; /* text segment size in bytes */
vir_bytes data_bytes; /* size of initialized data in bytes */
vir_bytes bss_bytes; /* size of bss in bytes */
vir_bytes stk_bytes; /* size of initial stack segment in bytes */
phys_bytes tot_bytes; /* total memory to allocate, including gap */
{
    /* Allocate new memory and release the old memory. Change the map and report
    * the new map to the kernel. Zero the new core image's bss, gap and stack.
    */

    vir_clicks text_clicks, data_clicks, gap_clicks, stack_clicks, tot_clicks;
    phys_clicks new_base;
    phys_bytes bytes, base, bss_offset;
    int s, r2;

    /* No need to allocate text if it can be shared. */
    if (sh_mp != NULL) text_bytes = 0;

    /* Allow the old data to be swapped out to make room. (Which is really a
    * waste of time, because we are going to throw it away anyway.)
    */
    rmp->mp_flags |= WAITING;

```

```

/* Acquire the new memory. Each of the 4 parts: text, (data+bss), gap,
 * and stack occupies an integral number of clicks, starting at click
 * boundary. The data and bss parts are run together with no space.
 */
text_clicks = ((unsigned long) text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
data_clicks = (data_bytes + bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
stack_clicks = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
tot_clicks = (tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
gap_clicks = tot_clicks - data_clicks - stack_clicks;
if ( (int) gap_clicks < 0) return(ENOMEM);

/* Try to allocate memory for the new process. */
new_base = alloc_mem(text_clicks + tot_clicks);
if (new_base == NO_MEM) return(ENOMEM);

/* We've got memory for the new core image. Release the old one. */
if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL) {
    /* No other process shares the text segment, so free it. */
    free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
}
/* Free the data and stack segments. */
free_mem(rmp->mp_seg[D].mem_phys,
    rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);

/* We have now passed the point of no return. The old core image has been
 * forever lost, memory for a new core image has been allocated. Set up
 * and report new map.
 */
if (sh_mp != NULL) {
    /* Share the text segment. */
    rmp->mp_seg[T] = sh_mp->mp_seg[T];
} else {
    rmp->mp_seg[T].mem_phys = new_base;
    rmp->mp_seg[T].mem_vir = 0;
    rmp->mp_seg[T].mem_len = text_clicks;

    if (text_clicks > 0)
    {
        /* Zero the last click of the text segment. Otherwise the
         * part of that click may remain unchanged.
         */
        base = (phys_bytes)(new_base+text_clicks-1) << CLICK_SHIFT;
        if ((s= sys_memset(0, base, CLICK_SIZE)) != OK)
            panic(__FILE__, "new_mem: sys_memset failed", s);
    }
}
rmp->mp_seg[D].mem_phys = new_base + text_clicks;
rmp->mp_seg[D].mem_vir = 0;
rmp->mp_seg[D].mem_len = data_clicks;
rmp->mp_seg[S].mem_phys = rmp->mp_seg[D].mem_phys + data_clicks + gap_clicks;
rmp->mp_seg[S].mem_vir = rmp->mp_seg[D].mem_vir + data_clicks + gap_clicks;
rmp->mp_seg[S].mem_len = stack_clicks;

#if (CHIP == M68000)
    rmp->mp_seg[T].mem_vir = 0;
    rmp->mp_seg[D].mem_vir = rmp->mp_seg[T].mem_len;
    rmp->mp_seg[S].mem_vir = rmp->mp_seg[D].mem_vir
        + rmp->mp_seg[D].mem_len + gap_clicks;
#endif

if ((r2=sys_newmap(rmp->mp_endpoint, rmp->mp_seg)) != OK) {
    /* report new map to the kernel */
    panic(__FILE__, "sys_newmap failed", r2);
}

/* The old memory may have been swapped out, but the new memory is real. */
rmp->mp_flags &= ~(WAITING|ONSWAP|SWAPIN);

/* Zero the bss, gap, and stack segment. */
bytes = (phys_bytes)(data_clicks + gap_clicks + stack_clicks) << CLICK_SHIFT;
base = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
bss_offset = (data_bytes >> CLICK_SHIFT) << CLICK_SHIFT;
base += bss_offset;

```



```
bytes -= bss_offset;

if ((s=sys_memset(0, base, bytes)) != OK) {
    panic(__FILE__, "new_mem can't zero", s);
}

return(OK);
}
```

```

/* This file deals with creating processes (via FORK) and deleting them (via
 * EXIT/WAIT). When a process forks, a new slot in the 'mproc' table is
 * allocated for it, and a copy of the parent's core image is made for the
 * child. Then the kernel and file system are informed. A process is removed
 * from the 'mproc' table when two events have occurred: (1) it has exited or
 * been killed by a signal, and (2) the parent has done a WAIT. If the process
 * exits first, it continues to occupy a slot until the parent does a WAIT.
 *
 * The entry points into this file are:
 *   do_fork:      perform the FORK system call
 *   do_pm_exit:   perform the EXIT system call (by calling pm_exit())
 *   pm_exit:      actually do the exiting
 *   do_wait:      perform the WAITPID or WAIT system call
 *   tell_parent:  tell parent about the death of a child
 */

#include "pm.h"
#include <sys/wait.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <sys/resource.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

#define LAST_FEW          2    /* last few slots reserved for superuser */

FORWARD _PROTOTYPE (void cleanup, (register struct mproc *child) );

/*=====
 *
 * do_fork
 *=====*/
PUBLIC int do_fork()
{
    /* The process pointed to by 'mp' has forked. Create a child process. */
    register struct mproc *rmp; /* pointer to parent */
    register struct mproc *rmc; /* pointer to child */
    int child_nr, s;
    phys_clicks prog_clicks, child_base;
    phys_bytes prog_bytes, parent_abs, child_abs; /* Intel only */
    pid_t new_pid;
    static int next_child;
    int n = 0, r;

    /* If tables might fill up during FORK, don't even start since recovery half
     * way through is such a nuisance.
     */
    rmp = mp;
    if ((procs_in_use == NR_PROCS) ||
        (procs_in_use >= NR_PROCS-LAST_FEW && rmp->mp_effuid != 0))
    {
        printf("PM: warning, process table is full!\n");
        return(EAGAIN);
    }

    /* Determine how much memory to allocate. Only the data and stack need to
     * be copied, because the text segment is either shared or of zero length.
     */
    prog_clicks = (phys_clicks) rmp->mp_seg[S].mem_len;
    prog_clicks += (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
    prog_bytes = (phys_bytes) prog_clicks << CLICK_SHIFT;
    if ( (child_base = alloc_mem(prog_clicks)) == NO_MEM) return(ENOMEM);

    /* Create a copy of the parent's core image for the child. */
    child_abs = (phys_bytes) child_base << CLICK_SHIFT;
    parent_abs = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
    s = sys_abscopy(parent_abs, child_abs, prog_bytes);
    if (s < 0) panic(__FILE__, "do_fork can't copy", s);

    /* Find a slot in 'mproc' for the child process. A slot must exist. */
    do {
        next_child = (next_child+1) % NR_PROCS;
        n++;
    } while((mproc[next_child].mp_flags & IN_USE) && n <= NR_PROCS);

```

```

if(n > NR_PROCS)
    panic(__FILE__, "do_fork can't find child slot", NO_NUM);
if(next_child < 0 || next_child >= NR_PROCS
|| (mproc[next_child].mp_flags & IN_USE))
    panic(__FILE__, "do_fork finds wrong child slot", next_child);

rmc = &mproc[next_child];
/* Set up the child and its memory map; copy its 'mproc' slot from parent. */
child_nr = (int)(rmc - mproc); /* slot number of the child */
procs_in_use++;
*rmc = *rmp; /* copy parent's process slot to child's */
rmc->mp_parent = who_p; /* record child's parent */
/* inherit only these flags */
rmc->mp_flags &= (IN_USE|SEPARATE|PRIV_PROC|DONT_SWAP);
rmc->mp_child_utime = 0; /* reset administration */
rmc->mp_child_stime = 0; /* reset administration */

/* A separate I&D child keeps the parents text segment. The data and stack
 * segments must refer to the new copy.
 */
if (!(rmc->mp_flags & SEPARATE)) rmc->mp_seg[T].mem_phys = child_base;
rmc->mp_seg[D].mem_phys = child_base;
rmc->mp_seg[S].mem_phys = rmc->mp_seg[D].mem_phys +
    (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
rmc->mp_exitstatus = 0;
rmc->mp_sigstatus = 0;

/* Find a free pid for the child and put it in the table. */
new_pid = get_free_pid();
rmc->mp_pid = new_pid; /* assign pid to child */

/* Tell kernel and file system about the (now successful) FORK. */
if((r=sys_fork(who_e, child_nr, &rmc->mp_endpoint, rmc->mp_seg)) != OK) {
    panic(__FILE__, "do_fork can't sys_fork", r);
}

if (rmc->mp_fs_call != PM_IDLE)
    panic("pm", "do_fork: not idle", rmc->mp_fs_call);
rmc->mp_fs_call = PM_FORK;
r = notify(FS_PROC_NR);
if (r != OK) panic("pm", "do_fork: unable to notify FS", r);

/* Do not reply until FS is ready to process the fork
 * request
 */
return SUSPEND;
}

/*=====
 * do_fork_nb
 *=====*/
PUBLIC int do_fork_nb()
{
/* The process pointed to by 'mp' has forked. Create a child process. */
register struct mproc *rmp; /* pointer to parent */
register struct mproc *rmc; /* pointer to child */
int child_nr, s;
phys_clicks prog_clicks, child_base;
phys_bytes prog_bytes, parent_abs, child_abs; /* Intel only */
pid_t new_pid;
static int next_child;
int n = 0, r;

/* Only system processes are allowed to use fork_nb */
if (!(mp->mp_flags & PRIV_PROC))
    return EPERM;

/* If tables might fill up during FORK, don't even start since recovery half
 * way through is such a nuisance.
 */
rmp = mp;
if ((procs_in_use == NR_PROCS) ||
    (procs_in_use >= NR_PROCS-LAST_FEW && rmp->mp_effuid != 0))
{

```

```

    printf("PM: warning, process table is full!\n");
    return(EAGAIN);
}

/* Determine how much memory to allocate. Only the data and stack need to
 * be copied, because the text segment is either shared or of zero length.
 */
prog_clicks = (phys_clicks) rmp->mp_seg[S].mem_len;
prog_clicks += (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
prog_bytes = (phys_bytes) prog_clicks << CLICK_SHIFT;
if ( (child_base = alloc_mem(prog_clicks)) == NO_MEM) return(ENOMEM);

/* Create a copy of the parent's core image for the child. */
child_abs = (phys_bytes) child_base << CLICK_SHIFT;
parent_abs = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
s = sys_abscopy(parent_abs, child_abs, prog_bytes);
if (s < 0) panic(__FILE__, "do_fork can't copy", s);

/* Find a slot in 'mproc' for the child process. A slot must exist. */
do {
    next_child = (next_child+1) % NR_PROCS;
    n++;
} while((mproc[next_child].mp_flags & IN_USE) && n <= NR_PROCS);
if(n > NR_PROCS)
    panic(__FILE__, "do_fork can't find child slot", NO_NUM);
if(next_child < 0 || next_child >= NR_PROCS
|| (mproc[next_child].mp_flags & IN_USE))
    panic(__FILE__, "do_fork finds wrong child slot", next_child);

rmc = &mproc[next_child];
/* Set up the child and its memory map; copy its 'mproc' slot from parent. */
child_nr = (int)(rmc - mproc); /* slot number of the child */
procs_in_use++;
*rmc = *rmp; /* copy parent's process slot to child's */
rmc->mp_parent = who_p; /* record child's parent */
/* inherit only these flags */
rmc->mp_flags &= (IN_USE|SEPARATE|PRIV_PROC|DONT_SWAP);
rmc->mp_child_utime = 0; /* reset administration */
rmc->mp_child_stime = 0; /* reset administration */

/* A separate I&D child keeps the parents text segment. The data and stack
 * segments must refer to the new copy.
 */
if (!(rmc->mp_flags & SEPARATE)) rmc->mp_seg[T].mem_phys = child_base;
rmc->mp_seg[D].mem_phys = child_base;
rmc->mp_seg[S].mem_phys = rmc->mp_seg[D].mem_phys +
    (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
rmc->mp_exitstatus = 0;
rmc->mp_sigstatus = 0;

/* Find a free pid for the child and put it in the table. */
new_pid = get_free_pid();
rmc->mp_pid = new_pid; /* assign pid to child */

/* Tell kernel and file system about the (now successful) FORK. */
if((r=sys_fork(who_e, child_nr, &rmc->mp_endpoint, rmc->mp_seg)) != OK) {
    panic(__FILE__, "do_fork can't sys_fork", r);
}

if (rmc->mp_fs_call != PM_IDLE)
    panic("pm", "do_fork: not idle", rmc->mp_fs_call);
rmc->mp_fs_call= PM_FORK_NB;
r= notify(FS_PROC_NR);
if (r != OK) panic("pm", "do_fork: unable to notify FS", r);

/* Wakeup the newly created process */
setreply(rmc-mproc, OK);

return rmc->mp_pid;
}

/*=====
 *
 * do_pm_exit
 *=====*/

```

```

PUBLIC int do_pm_exit()
{
    /* Perform the exit(status) system call. The real work is done by pm_exit(),
     * which is also called when a process is killed by a signal.
     */
    pm_exit(mp, m_in.status, FALSE /*!for_trace*/);
    return(SUSPEND);          /* can't communicate from beyond the grave */
}

/*=====
 *                               pm_exit                               *
 *=====*/
PUBLIC void pm_exit(rmp, exit_status, for_trace)
register struct mproc *rmp;    /* pointer to the process to be terminated */
int exit_status;              /* the process' exit status (for parent) */
int for_trace;
{
    /* A process is done. Release most of the process' possessions. If its
     * parent is waiting, release the rest, else keep the process slot and
     * become a zombie.
     */
    register int proc_nr, proc_nr_e;
    int parent_waiting, right_child, r;
    pid_t pidarg, procgrp;
    struct mproc *p_mp;
    clock_t t[5];

    proc_nr = (int) (rmp - mproc);          /* get process slot number */
    proc_nr_e = rmp->mp_endpoint;

    /* Remember a session leader's process group. */
    procgrp = (rmp->mp_pid == mp->mp_procgrp) ? mp->mp_procgrp : 0;

    /* If the exited process has a timer pending, kill it. */
    if (rmp->mp_flags & ALARM_ON) set_alarm(proc_nr_e, (unsigned) 0);

    /* Do accounting: fetch usage times and accumulate at parent. */
    if((r=sys_times(proc_nr_e, t)) != OK)
        panic(__FILE__, "pm_exit: sys_times failed", r);

    p_mp = &mproc[rmp->mp_parent];          /* process' parent */
    p_mp->mp_child_utime += t[0] + rmp->mp_child_utime; /* add user time */
    p_mp->mp_child_stime += t[1] + rmp->mp_child_stime; /* add system time */

    /* Tell the kernel the process is no longer runnable to prevent it from
     * being scheduled in between the following steps. Then tell FS that it
     * the process has exited and finally, clean up the process at the kernel.
     * This order is important so that FS can tell drivers to cancel requests
     * such as copying to/ from the exiting process, before it is gone.
     */
    sys_nice(proc_nr_e, PRIO_STOP);          /* stop the process */

    if (proc_nr_e == INIT_PROC_NR)
    {
        printf("PM: INIT died\n");
        return;
    }
    else
    if (proc_nr_e != FS_PROC_NR)              /* if it is not FS that is exiting.. */
    {
        /* Tell FS about the exiting process. */
        if (rmp->mp_fs_call != PM_IDLE)
            panic(__FILE__, "pm_exit: not idle", rmp->mp_fs_call);
        rmp->mp_fs_call = (for_trace ? PM_EXIT_TR : PM_EXIT);
        r = notify(FS_PROC_NR);
        if (r != OK) panic(__FILE__, "pm_exit: unable to notify FS", r);

        if (rmp->mp_flags & PRIV_PROC)
        {
            /* destroy system processes without waiting for FS */
            if((r = sys_exit(rmp->mp_endpoint)) != OK)
                panic(__FILE__, "pm_exit: sys_exit failed", r);
        }
    }
}

```

```

else
{
    printf("PM: FS died\n");
    return;
}

/* Pending reply messages for the dead process cannot be delivered. */
rmp->mp_flags &= ~REPLY;

/* Keep the process around until FS is finished with it. */

rmp->mp_exitstatus = (char) exit_status;
pidarg = p_mp->mp_wpid;          /* who's being waited for? */
parent_waiting = p_mp->mp_flags & WAITING;
right_child =                    /* child meets one of the 3 tests? */
    (pidarg == -1 || pidarg == rmp->mp_pid || -pidarg == rmp->mp_procgrp);

if (parent_waiting && right_child) {
    tell_parent(rmp);          /* tell parent */
} else {
    rmp->mp_flags &= (IN_USE|PRIV_PROC);
    rmp->mp_flags |= ZOMBIE;    /* parent not waiting, zombify child */
    sig_proc(p_mp, SIGCHLD);   /* send parent a "child died" signal */
}

/* If the process has children, disinherit them. INIT is the new parent. */
for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
    if (rmp->mp_flags & IN_USE && rmp->mp_parent == proc_nr) {
        /* 'rmp' now points to a child to be disinherited. */
        rmp->mp_parent = INIT_PROC_NR;
        parent_waiting = mproc[INIT_PROC_NR].mp_flags & WAITING;
        if (parent_waiting && (rmp->mp_flags & ZOMBIE))
            cleanup(rmp);
    }
}

/* Send a hangup to the process' process group if it was a session leader. */
if (procgrp != 0) check_sig(-procgrp, SIGHUP);
}

/*=====
*                                     do_waitpid                                     *
*=====*/
PUBLIC int do_waitpid()
{
    /* A process wants to wait for a child to terminate. If a child is already
    * waiting, go clean it up and let this WAIT call terminate. Otherwise,
    * really wait.
    * A process calling WAIT never gets a reply in the usual way at the end
    * of the main loop (unless WNOHANG is set or no qualifying child exists).
    * If a child has already exited, the routine cleanup() sends the reply
    * to awaken the caller.
    * Both WAIT and WAITPID are handled by this code.
    */
    register struct mproc *rp;
    int pidarg, options, children;

    /* Set internal variables, depending on whether this is WAIT or WAITPID. */
    pidarg = (call_nr == WAIT ? -1 : m_in.pid);    /* 1st param of waitpid */
    options = (call_nr == WAIT ? 0 : m_in.sig_nr); /* 3rd param of waitpid */
    if (pidarg == 0) pidarg = -mp->mp_procgrp;    /* pidarg < 0 ==> proc grp */

    /* Is there a child waiting to be collected? At this point, pidarg != 0:
    * pidarg > 0 means pidarg is pid of a specific process to wait for
    * pidarg == -1 means wait for any child
    * pidarg < -1 means wait for any child whose process group = -pidarg
    */
    children = 0;
    for (rp = &mproc[0]; rp < &mproc[NR_PROCS]; rp++) {
        if ( (rp->mp_flags & IN_USE) && rp->mp_parent == who_p) {
            /* The value of pidarg determines which children qualify. */
            if (pidarg > 0 && pidarg != rp->mp_pid) continue;
            if (pidarg < -1 && -pidarg != rp->mp_procgrp) continue;

```

```

        children++;
        /* this child is acceptable */
        if (rp->mp_flags & ZOMBIE) {
            /* This child meets the pid test and has exited. */
            tell_parent(rp); /* this child has already exited */
            if (rp->mp_fs_call == PM_IDLE)
                real_cleanup(rp);
            return(SUSPEND);
        }
        if ((rp->mp_flags & STOPPED) && rp->mp_sigstatus) {
            /* This child meets the pid test and is being traced.*/
            mp->mp_reply.reply_res2 = 0177|(rp->mp_sigstatus << 8);
            rp->mp_sigstatus = 0;
            return(rp->mp_pid);
        }
    }
}

/* No qualifying child has exited. Wait for one, unless none exists. */
if (children > 0) {
    /* At least 1 child meets the pid test exists, but has not exited. */
    if (options & WNOHANG) return(0); /* parent does not want to wait */
    mp->mp_flags |= WAITING; /* parent wants to wait */
    mp->mp_wpid = (pid_t) pidarg; /* save pid for later */
    return(SUSPEND); /* do not reply, let it wait */
} else {
    /* No child even meets the pid test. Return error immediately. */
    return(ECHILD); /* no - parent has no children */
}
}

/*=====
 *                                *
 *                                *
 *=====*/
PRIVATE void cleanup(child)
register struct mproc *child; /* tells which process is exiting */
{
    /* Finish off the exit of a process. The process has exited or been killed
    * by a signal, and its parent is waiting.
    */

    if (child->mp_fs_call != PM_IDLE)
        panic(__FILE__, "cleanup: notidle", child->mp_fs_call);

    tell_parent(child);
    real_cleanup(child);
}

/*=====
 *                                *
 *                                *
 *=====*/
PUBLIC void tell_parent(child)
register struct mproc *child; /* tells which process is exiting */
{
    int exitstatus, mp_parent;
    struct mproc *parent;

    mp_parent = child->mp_parent;
    if (mp_parent <= 0)
        panic(__FILE__, "tell_parent: bad value in mp_parent", mp_parent);
    parent = &mproc[mp_parent];

    /* Wake up the parent by sending the reply message. */
    exitstatus = (child->mp_exitstatus << 8) | (child->mp_sigstatus & 0377);
    parent->mp_reply.reply_res2 = exitstatus;
    setreply(child->mp_parent, child->mp_pid);
    parent->mp_flags &= ~WAITING; /* parent no longer waiting */
    child->mp_flags &= ~ZOMBIE; /* avoid informing parent twice */
}

/*=====
 *                                *
 *                                *
 *=====*/
PUBLIC void real_cleanup(rmp)

```

```
register struct mproc *rmp;      /* tells which process is exiting */
{
    /* Release the process table entry and reinitialize some field. */
    rmp->mp_pid = 0;
    rmp->mp_flags = 0;
    rmp->mp_child_utime = 0;
    rmp->mp_child_stime = 0;
    procs_in_use--;
}
```



```

/* This file handles the 4 system calls that get and set uids and gids.
 * It also handles getpid(), setsid(), and getpgrp(). The code for each
 * one is so tiny that it hardly seemed worthwhile to make each a separate
 * function.
 */

#include "pm.h"
#include <minix/callnr.h>
#include <minix/endpoint.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

/*=====
 *
 * do_getset
 *=====*/

PUBLIC int do_getset()
{
/* Handle GETUID, GETGID, GETPID, GETPGRP, SETUID, SETGID, SETSID. The four
 * GETs and SETSID return their primary results in 'r'. GETUID, GETGID, and
 * GETPID also return secondary results (the effective IDs, or the parent
 * process ID) in 'reply_res2', which is returned to the user.
 */

    register struct mproc *rmp = mp;
    int r, proc;

    switch(call_nr) {
        case GETUID:
            r = rmp->mp_realuid;
            rmp->mp_reply.reply_res2 = rmp->mp_effuid;
            break;

        case GETGID:
            r = rmp->mp_realgid;
            rmp->mp_reply.reply_res2 = rmp->mp_effgid;
            break;

        case GETPID:
            r = mproc[who_p].mp_pid;
            rmp->mp_reply.reply_res2 = mproc[rmp->mp_parent].mp_pid;
            if(pm_isokendpt(m_in.endpt, &proc) == OK && proc >= 0)
                rmp->mp_reply.reply_res3 = mproc[proc].mp_pid;
            break;

        case SETEUID:
        case SETUID:
            if (rmp->mp_realuid != (uid_t) m_in.usr_id &&
                rmp->mp_effuid != SUPER_USER)
                return(EPERM);
            if(call_nr == SETUID) rmp->mp_realuid = (uid_t) m_in.usr_id;
            rmp->mp_effuid = (uid_t) m_in.usr_id;

            if (rmp->mp_fs_call != PM_IDLE)
            {
                panic(__FILE__, "do_getset: not idle",
                    rmp->mp_fs_call);
            }
            rmp->mp_fs_call= PM_SETUID;
            r= notify(FS_PROC_NR);
            if (r != OK)
                panic(__FILE__, "do_getset: unable to notify FS", r);

            /* Do not reply until FS is ready to process the setuid
             * request
             */
            r= SUSPEND;
            break;

        case SETEGID:
        case SETGID:
            if (rmp->mp_realgid != (gid_t) m_in.grp_id &&
                rmp->mp_effuid != SUPER_USER)
                return(EPERM);
    }

```

```
    if(call_nr == SETGID) rmp->mp_realgid = (gid_t) m_in.grp_id;
    rmp->mp_effgid = (gid_t) m_in.grp_id;

    if (rmp->mp_fs_call != PM_IDLE)
    {
        panic(__FILE__, "do_getset: not idle",
              rmp->mp_fs_call);
    }
    rmp->mp_fs_call= PM_SETGID;
    r= notify(FS_PROC_NR);
    if (r != OK)
        panic(__FILE__, "do_getset: unable to notify FS", r);

    /* Do not reply until FS is ready to process the setgid
     * request
     */
    r= SUSPEND;
    break;

case SETSID:
    if (rmp->mp_procgrp == rmp->mp_pid) return(EPERM);
    rmp->mp_procgrp = rmp->mp_pid;

    if (rmp->mp_fs_call != PM_IDLE)
    {
        panic(__FILE__, "do_getset: not idle",
              rmp->mp_fs_call);
    }
    rmp->mp_fs_call= PM_SETSID;
    r= notify(FS_PROC_NR);
    if (r != OK)
        panic(__FILE__, "do_getset: unable to notify FS", r);

    /* Do not reply until FS is ready to process the setsid
     * request
     */
    r= SUSPEND;
    break;

case GETPGRP:
    r = rmp->mp_procgrp;
    break;

default:
    r = EINVAL;
    break;
}
return(r);
}
```

```
/* EXTERN should be extern except in table.c */
#ifdef _TABLE
#undef EXTERN
#define EXTERN
#endif

/* Global variables. */
EXTERN struct mproc *mp;          /* ptr to 'mproc' slot of current process */
EXTERN int procs_in_use;          /* how many processes are marked as IN_USE */
EXTERN char monitor_params[128*sizeof(char *)]; /* boot monitor parameters */
EXTERN struct kinfo kinfo;        /* kernel information */

/* Misc.c */
extern struct utsname uts_val; /* uname info */

/* The parameters of the call are kept here. */
EXTERN message m_in;            /* the incoming message itself is kept here. */
EXTERN int who_p, who_e;         /* caller's proc number, endpoint */
EXTERN int call_nr;              /* system call number */

extern _PROTOTYPE (int (*call_vec[]), (void) ); /* system call handlers */
extern char core_name[];         /* file name where core images are produced */
EXTERN sigset_t core_sset;       /* which signals cause core images */
EXTERN sigset_t ign_sset;        /* which signals are by default ignored */

EXTERN time_t boottime;          /* time when the system was booted (for
                                   * reporting to FS)
                                   */
EXTERN int report_reboot;        /* During reboot to report to FS that we are
                                   * rebooting.
                                   */

EXTERN int abort_flag;
EXTERN char monitor_code[256];
```

```

/* This file contains the main program of the process manager and some related
 * procedures. When MINIX starts up, the kernel runs for a little while,
 * initializing itself and its tasks, and then it runs PM and FS. Both PM
 * and FS initialize themselves as far as they can. PM asks the kernel for
 * all free memory and starts serving requests.
 *
 * The entry points into this file are:
 *   main:      starts PM running
 *   setreply:  set the reply to be sent to process making an PM system call
 */

#include "pm.h"
#include <minix/keymap.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <minix/endpoint.h>
#include <minix/minlib.h>
#include <signal.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/resource.h>
#include <sys/utsname.h>
#include <string.h>
#include "mproc.h"
#include "param.h"

#include "../kernel/const.h"
#include "../kernel/config.h"
#include "../kernel/type.h"
#include "../kernel/proc.h"

FORWARD _PROTOTYPE( void get_work, (void) ) ;
FORWARD _PROTOTYPE( void pm_init, (void) ) ;
FORWARD _PROTOTYPE( int get_nice_value, (int queue) ) ;
FORWARD _PROTOTYPE( void get_mem_chunks, (struct memory *mem_chunks) ) ;
FORWARD _PROTOTYPE( void patch_mem_chunks, (struct memory *mem_chunks,
      struct mem_map *map_ptr) ) ;
FORWARD _PROTOTYPE( void do_x86_vm, (struct memory mem_chunks[NR_MEMS]) ) ;
FORWARD _PROTOTYPE( void send_work, (void) ) ;
FORWARD _PROTOTYPE( void handle_fs_reply, (message *m_ptr) ) ;

#define click_to_round_k(n) \
    ((unsigned) (((unsigned long) (n) << CLICK_SHIFT) + 512) / 1024))

/*=====
 *
 *                               main
 *=====*/
PUBLIC int main()
{
    /* Main routine of the process manager. */
    int result, s, proc_nr;
    struct mproc *rmp;
    sigset_t sigset;

    pm_init();                /* initialize process manager tables */

    /* This is PM's main loop- get work and do it, forever and forever. */
    while (TRUE) {
        get_work();           /* wait for an PM system call */

        /* Check for system notifications first. Special cases. */
        switch(call_nr)
        {
            case SYN_ALARM:
                pm_expire_timers(m_in.NOTIFY_TIMESTAMP);
                result = SUSPEND;          /* don't reply */
                break;
            case SYS_SIG:
                /* signals pending */
                sigset = m_in.NOTIFY_ARG;
                if (sigismember(&sigset, SIGKSIG)) {
                    (void) ksig_pending();
                }
                result = SUSPEND;          /* don't reply */
                break;
        }
    }
}

```

```

case PM_GET_WORK:
    if (who_e == FS_PROC_NR)
    {
        send_work();
        result= SUSPEND;                /* don't reply */
    }
    else
        result= ENOSYS;
    break;
case PM_EXIT_REPLY:
case PM_REBOOT_REPLY:
case PM_EXEC_REPLY:
case PM_CORE_REPLY:
case PM_EXIT_REPLY_TR:
    if (who_e == FS_PROC_NR)
    {
        handle_fs_reply(&m_in);
        result= SUSPEND;                /* don't reply */
    }
    else
        result= ENOSYS;
    break;
case ALLOCMEM:
    result= do_allocmem();
    break;
case FORK_NB:
    result= do_fork_nb();
    break;
case EXEC_NEWMEM:
    result= exec_newmem();
    break;
case EXEC_RESTART:
    result= do_execrestart();
    break;
case PROCSTAT:
    result= do_procstat();
    break;
case GETPROCNR:
    result= do_getprocnr();
    break;
default:
    /* Else, if the system call number is valid, perform the
     * call.
     */
    if ((unsigned) call_nr >= NCALLS) {
        result = ENOSYS;
    } else {
        result = (*call_vec[call_nr])();
    }
    break;
}

/* Send the results back to the user to indicate completion. */
if (result != SUSPEND) setreply(who_p, result);

swap_in();                /* maybe a process can be swapped in? */

/* Send out all pending reply messages, including the answer to
 * the call just made above. The processes must not be swapped out.
 */
for (proc_nr=0, rmp=mproc; proc_nr < NR_PROCS; proc_nr++, rmp++) {
    /* In the meantime, the process may have been killed by a
     * signal (e.g. if a lethal pending signal was unblocked)
     * without the PM realizing it. If the slot is no longer in
     * use or just a zombie, don't try to reply.
     */
    if ((rmp->mp_flags & (REPLY | ONSWAP | IN_USE | ZOMBIE)) ==
        (REPLY | IN_USE)) {
        if ((s=send(rmp->mp_endpoint, &rmp->mp_reply)) != OK) {
            printf("PM can't reply to %d (%s)\n",
                rmp->mp_endpoint, rmp->mp_name);
            panic(__FILE__, "PM can't reply", NO_NUM);
        }
        rmp->mp_flags &= ~REPLY;
    }
}

```

```

    }
}
return(OK);
}

/*=====
 *                               get_work                               *
 *=====*/
PRIVATE void get_work()
{
/* Wait for the next message and extract useful information from it. */
if (receive(ANY, &m_in) != OK)
    panic(__FILE__, "PM receive error", NO_NUM);
who_e = m_in.m_source; /* who sent the message */
if(pm_isokendpt(who_e, &who_p) != OK)
    panic(__FILE__, "PM got message from invalid endpoint", who_e);
call_nr = m_in.m_type; /* system call number */

/* Process slot of caller. Misuse PM's own process slot if the kernel is
 * calling. This can happen in case of synchronous alarms (CLOCK) or or
 * event like pending kernel signals (SYSTEM).
 */
mp = &mproc[who_p < 0 ? PM_PROC_NR : who_p];
if(who_p >= 0 && mp->mp_endpoint != who_e) {
    panic(__FILE__, "PM endpoint number out of sync with source",
        mp->mp_endpoint);
}
}

/*=====
 *                               setreply                               *
 *=====*/
PUBLIC void setreply(proc_nr, result)
int proc_nr; /* process to reply to */
int result; /* result of call (usually OK or error #) */
{
/* Fill in a reply message to be sent later to a user process. System calls
 * may occasionally fill in other fields, this is only for the main return
 * value, and for setting the "must send reply" flag.
 */
register struct mproc *rmp = &mproc[proc_nr];

if(proc_nr < 0 || proc_nr >= NR_PROCS)
    panic(__FILE__, "setreply arg out of range", proc_nr);

rmp->mp_reply.reply_res = result;
rmp->mp_flags |= REPLY; /* reply pending */

if (rmp->mp_flags & ONSWAP)
    swap_inqueue(rmp); /* must swap this process back in */
}

/*=====
 *                               pm_init                               *
 *=====*/
PRIVATE void pm_init()
{
/* Initialize the process manager.
 * Memory use info is collected from the boot monitor, the kernel, and
 * all processes compiled into the system image. Initially this information
 * is put into an array mem_chunks. Elements of mem_chunks are struct memory,
 * and hold base, size pairs in units of clicks. This array is small, there
 * should be no more than 8 chunks. After the array of chunks has been built
 * the contents are used to initialize the hole list. Space for the hole list
 * is reserved as an array with twice as many elements as the maximum number
 * of processes allowed. It is managed as a linked list, and elements of the
 * array are struct hole, which, in addition to storage for a base and size in
 * click units also contain space for a link, a pointer to another element.
 */
int s;
static struct boot_image image[NR_BOOT_PROCS];
register struct boot_image *ip;
static char core_sigs[] = { SIGQUIT, SIGILL, SIGTRAP, SIGABRT,

```

```

        SIGEMT, SIGFPE, SIGUSR1, SIGSEGV, SIGUSR2 };
static char ign_sigs[] = { SIGCHLD, SIGWINCH, SIGCONT };
static char mess_sigs[] = { SIGTERM, SIGHUP, SIGABRT, SIGQUIT };
register struct mproc *rmp;
register int i;
register char *sig_ptr;
phys_clicks total_clicks, minix_clicks, free_clicks;
message mess;
struct mem_map mem_map[NR_LOCAL_SEGS];
struct memory mem_chunks[NR_MEMS];

/* Initialize process table, including timers. */
for (rmp=&mproc[0]; rmp<&mproc[NR_PROCS]; rmp++) {
    tmr_inittimer(&rmp->mp_timer);

    rmp->mp_fs_call= PM_IDLE;
    rmp->mp_fs_call2= PM_IDLE;
}

/* Build the set of signals which cause core dumps, and the set of signals
 * that are by default ignored.
 */
sigemptyset(&core_sset);
for (sig_ptr = core_sigs; sig_ptr < core_sigs+sizeof(core_sigs); sig_ptr++)
    sigaddset(&core_sset, *sig_ptr);
sigemptyset(&ign_sset);
for (sig_ptr = ign_sigs; sig_ptr < ign_sigs+sizeof(ign_sigs); sig_ptr++)
    sigaddset(&ign_sset, *sig_ptr);

/* Obtain a copy of the boot monitor parameters and the kernel info struct.
 * Parse the list of free memory chunks. This list is what the boot monitor
 * reported, but it must be corrected for the kernel and system processes.
 */
if ((s=sys_getmonparams(monitor_params, sizeof(monitor_params))) != OK)
    panic(__FILE__, "get monitor params failed", s);
get_mem_chunks(mem_chunks);
if ((s=sys_getkinfo(&kinfo)) != OK)
    panic(__FILE__, "get kernel info failed", s);

/* Get the memory map of the kernel to see how much memory it uses. */
if ((s=get_mem_map(SYSTASK, mem_map)) != OK)
    panic(__FILE__, "couldn't get memory map of SYSTASK", s);
minix_clicks = (mem_map[S].mem_phys+mem_map[S].mem_len)-mem_map[T].mem_phys;
patch_mem_chunks(mem_chunks, mem_map);

/* Initialize PM's process table. Request a copy of the system image table
 * that is defined at the kernel level to see which slots to fill in.
 */
if (OK != (s=sys_getimage(image)))
    panic(__FILE__, "couldn't get image table: %d\n", s);
procs_in_use = 0;
printf("Building process table:");
for (ip = &image[0]; ip < &image[NR_BOOT_PROCS]; ip++) {
    if (ip->proc_nr >= 0) {
        procs_in_use += 1;

        /* Set process details found in the image table. */
        rmp = &mproc[ip->proc_nr];
        strncpy(rmp->mp_name, ip->proc_name, PROC_NAME_LEN);
        rmp->mp_parent = RS_PROC_NR;
        rmp->mp_nice = get_nice_value(ip->priority);
        sigemptyset(&rmp->mp_sig2mess);
        sigemptyset(&rmp->mp_ignore);
        sigemptyset(&rmp->mp_sigmask);
        sigemptyset(&rmp->mp_catch);
        if (ip->proc_nr == INIT_PROC_NR) {
            rmp->mp_procgrp = rmp->mp_pid = INIT_PID;
            rmp->mp_flags |= IN_USE;
        }
        else {
            rmp->mp_pid = get_free_pid();
            rmp->mp_flags |= IN_USE | DONT_SWAP | PRIV_PROC;
            for (sig_ptr = mess_sigs;
                sig_ptr < mess_sigs+sizeof(mess_sigs);

```

```

        sig_ptr++)
        sigaddset(&rmp->mp_sig2mess, *sig_ptr);
    }

    /* Get kernel endpoint identifier. */
    rmp->mp_endpoint = ip->endpoint;

    /* Get memory map for this process from the kernel. */
    if ((s=get_mem_map(ip->proc_nr, rmp->mp_seg)) != OK)
        panic(__FILE__, "couldn't get process entry", s);
    if (rmp->mp_seg[T].mem_len != 0) rmp->mp_flags |= SEPARATE;
    minix_clicks += rmp->mp_seg[S].mem_phys +
        rmp->mp_seg[S].mem_len - rmp->mp_seg[T].mem_phys;
    patch_mem_chunks(mem_chunks, rmp->mp_seg);

    /* Tell FS about this system process. */
    mess.PR_SLOT = ip->proc_nr;
    mess.PR_PID = rmp->mp_pid;
    mess.PR_ENDPT = rmp->mp_endpoint;
    if (OK != (s=send(FS_PROC_NR, &mess)))
        panic(__FILE__, "can't sync up with FS", s);
    printf("%s", ip->proc_name);    /* display process name */
}
printf("\n");    /* last process done */

/* Override some details. INIT, PM, FS and RS are somewhat special. */
mproc[PM_PROC_NR].mp_pid = PM_PID;    /* PM has magic pid */
mproc[RS_PROC_NR].mp_parent = INIT_PROC_NR;    /* INIT is root */
sigfillset(&mproc[PM_PROC_NR].mp_ignore);    /* guard against signals */

/* Tell FS that no more system processes follow and synchronize. */
mess.PR_ENDPT = NONE;
if (sendrec(FS_PROC_NR, &mess) != OK || mess.m_type != OK)
    panic(__FILE__, "can't sync up with FS", NO_NUM);

#if ENABLE_BOOTDEV
/* Possibly we must correct the memory chunks for the boot device. */
if (kinfo.bootdev_size > 0) {
    mem_map[T].mem_phys = kinfo.bootdev_base >> CLICK_SHIFT;
    mem_map[T].mem_len = 0;
    mem_map[D].mem_len = (kinfo.bootdev_size+CLICK_SIZE-1) >> CLICK_SHIFT;
    patch_mem_chunks(mem_chunks, mem_map);
}
#endif /* ENABLE_BOOTDEV */

/* Withhold some memory from x86 VM */
do_x86_vm(mem_chunks);

/* Initialize tables to all physical memory and print memory information. */
printf("Physical memory:");
mem_init(mem_chunks, &free_clicks);
total_clicks = minix_clicks + free_clicks;
printf(" total %u KB,", click_to_round_k(total_clicks));
printf(" system %u KB,", click_to_round_k(minix_clicks));
printf(" free %u KB.\n", click_to_round_k(free_clicks));
#if (CHIP == INTEL)
    uts_val.machine[0] = 'i';
    strcpy(uts_val.machine + 1, itoa(getprocessor()));
#endif
}

/*=====
*                               get_nice_value                               *
*=====*/
PRIVATE int get_nice_value(queue)
int queue;    /* store mem chunks here */
{
    /* Processes in the boot image have a priority assigned. The PM doesn't know
    * about priorities, but uses 'nice' values instead. The priority is between
    * MIN_USER_Q and MAX_USER_Q. We have to scale between PRIO_MIN and PRIO_MAX.
    */
    int nice_val = (queue - USER_Q) * (PRIO_MAX-PRIO_MIN+1) /
        (MIN_USER_Q-MAX_USER_Q+1);

```



```

    if (nice_val > PRIO_MAX) nice_val = PRIO_MAX; /* shouldn't happen */
    if (nice_val < PRIO_MIN) nice_val = PRIO_MIN; /* shouldn't happen */
    return nice_val;
}

#if _WORD_SIZE == 2
/* In real mode only 1M can be addressed, and in 16-bit protected we can go
 * no further than we can count in clicks. (The 286 is further limited by
 * its 24 bit address bus, but we can assume in that case that no more than
 * 16M memory is reported by the BIOS.)
 */
#define MAX_REAL          0x00100000L
#define MAX_16BIT         (0xFFFF0L << CLICK_SHIFT)
#endif

/*=====
 *                               get_mem_chunks                               *
 *=====*/
PRIVATE void get_mem_chunks(mem_chunks)
struct memory *mem_chunks;                /* store mem chunks here */
{
    /* Initialize the free memory list from the 'memory' boot variable. Translate
     * the byte offsets and sizes in this list to clicks, properly truncated. Also
     * make sure that we don't exceed the maximum address space of the 286 or the
     * 8086, i.e. when running in 16-bit protected mode or real mode.
     */
    long base, size, limit;
    char *s, *end;                        /* use to parse boot variable */
    int i, done = 0;
    struct memory *memp;
#if _WORD_SIZE == 2
    unsigned long max_address;
    struct machine machine;
    if (OK != (i=sys_getmachine(&machine)))
        panic(__FILE__, "sys_getmachine failed", i);
#endif

    /* Initialize everything to zero. */
    for (i = 0; i < NR_MEMS; i++) {
        memp = &mem_chunks[i];           /* next mem chunk is stored here */
        memp->base = memp->size = 0;
    }

    /* The available memory is determined by MINIX' boot loader as a list of
     * (base:size)-pairs in boothead.s. The 'memory' boot variable is set in
     * in boot.s. The format is "b0:s0,b1:s1,b2:s2", where b0:s0 is low mem,
     * b1:s1 is mem between 1M and 16M, b2:s2 is mem above 16M. Pairs b1:s1
     * and b2:s2 are combined if the memory is adjacent.
     */
    s = find_param("memory");              /* get memory boot variable */
    for (i = 0; i < NR_MEMS && !done; i++) {
        memp = &mem_chunks[i];            /* next mem chunk is stored here */
        base = size = 0;                   /* initialize next base:size pair */
        if (*s != 0) {                     /* get fresh data, unless at end */

            /* Read fresh base and expect colon as next char. */
            base = strtoul(s, &end, 0x10); /* get number */
            if (end != s && *end == ':') s = ++end; /* skip ':' */
            else *s=0;                      /* terminate, should not happen */

            /* Read fresh size and expect comma or assume end. */
            size = strtoul(s, &end, 0x10); /* get number */
            if (end != s && *end == ',') s = ++end; /* skip ',' */
            else done = 1;

        }
        limit = base + size;
    }
#if _WORD_SIZE == 2
    max_address = machine.protected ? MAX_16BIT : MAX_REAL;
    if (limit > max_address) limit = max_address;
#endif
    base = (base + CLICK_SIZE-1) & ~(long)(CLICK_SIZE-1);
    limit &= ~(long)(CLICK_SIZE-1);
    if (limit <= base) continue;
    memp->base = base >> CLICK_SHIFT;
}

```

```

        memptr->size = (limit - base) >> CLICK_SHIFT;
    }
}

/*=====
 *
 *          patch_mem_chunks
 *
 *=====*/
PRIVATE void patch_mem_chunks(mem_chunks, map_ptr)
struct memory *mem_chunks;          /* store mem chunks here */
struct mem_map *map_ptr;            /* memory to remove */
{
    /* Remove server memory from the free memory list. The boot monitor
     * promises to put processes at the start of memory chunks. The
     * tasks all use same base address, so only the first task changes
     * the memory lists. The servers and init have their own memory
     * spaces and their memory will be removed from the list.
     */
    struct memory *memp;
    for (memp = mem_chunks; memptr < &mem_chunks[NR_MEMS]; memptr++) {
        if (memp->base == map_ptr[T].mem_phys) {
            memptr->base += map_ptr[T].mem_len + map_ptr[S].mem_vir;
            memptr->size -= map_ptr[T].mem_len + map_ptr[S].mem_vir;
            break;
        }
    }
    if (memp >= &mem_chunks[NR_MEMS])
    {
        panic(__FILE__, "patch_mem_chunks: can't find map in mem_chunks, start",
            map_ptr[T].mem_phys);
    }
}

#define PAGE_SIZE          4096
#define PAGE_TABLE_COVER (1024*PAGE_SIZE)
/*=====
 *
 *          do_x86_vm
 *
 *=====*/
PRIVATE void do_x86_vm(mem_chunks)
struct memory mem_chunks[NR_MEMS];
{
    phys_bytes high, bytes;
    phys_clicks clicks, base_click;
    unsigned pages;
    int i, r;

    /* Compute the highest memory location */
    high= 0;
    for (i= 0; i<NR_MEMS; i++)
    {
        if (mem_chunks[i].size == 0)
            continue;
        if (mem_chunks[i].base + mem_chunks[i].size > high)
            high= mem_chunks[i].base + mem_chunks[i].size;
    }

    high <= CLICK_SHIFT;
#if VERBOSE_VM
    printf("do_x86_vm: found high 0x%x\n", high);
#endif

    /* The number of pages we need is one for the page directory, enough
     * page tables to cover the memory, and one page for alignment.
     */
    pages= 1 + (high + PAGE_TABLE_COVER-1)/PAGE_TABLE_COVER + 1;
    bytes= pages*PAGE_SIZE;
    clicks= (bytes + CLICK_SIZE-1) >> CLICK_SHIFT;

#if VERBOSE_VM
    printf("do_x86_vm: need %d pages\n", pages);
    printf("do_x86_vm: need %d bytes\n", bytes);
    printf("do_x86_vm: need %d clicks\n", clicks);
#endif

    for (i= 0; i<NR_MEMS; i++)

```

```

    {
        if (mem_chunks[i].size <= clicks)
            continue;
        break;
    }
    if (i >= NR_MEMS)
        panic("PM", "not enough memory for VM page tables?", NO_NUM);
    base_click= mem_chunks[i].base;
    mem_chunks[i].base += clicks;
    mem_chunks[i].size -= clicks;

#if VERBOSE_VM
    printf("do_x86_vm: using 0x%x clicks @ 0x%x\n", clicks, base_click);
#endif
    r= sys_vm_setbuf(base_click << CLICK_SHIFT, clicks << CLICK_SHIFT,
        high);
    if (r != 0)
        printf("do_x86_vm: sys_vm_setbuf failed: %d\n", r);
}

/*=====
 *                               send_work                               *
 *=====*/
PRIVATE void send_work()
{
    int r, call;
    struct mproc *rmp;
    message m;

    m.m_type= PM_IDLE;
    for (rmp= mproc; rmp < &mproc[NR_PROCS]; rmp++)
    {
        call= rmp->mp_fs_call;
        if (call == PM_IDLE)
            call= rmp->mp_fs_call2;
        if (call == PM_IDLE)
            continue;
        switch(call)
        {
        case PM_STIME:
            m.m_type= call;
            m.PM_STIME_TIME= boottime;

            /* FS does not reply */
            rmp->mp_fs_call= PM_IDLE;

            /* Wakeup the original caller */
            setreply(rmp-mproc, OK);
            break;

        case PM_SETSID:
            m.m_type= call;
            m.PM_SETSID_PROC= rmp->mp_endpoint;

            /* FS does not reply */
            rmp->mp_fs_call= PM_IDLE;

            /* Wakeup the original caller */
            setreply(rmp-mproc, rmp->mp_procgrp);
            break;

        case PM_SETGID:
            m.m_type= call;
            m.PM_SETGID_PROC= rmp->mp_endpoint;
            m.PM_SETGID_EGID= rmp->mp_effgid;
            m.PM_SETGID_RGID= rmp->mp_realgid;

            /* FS does not reply */
            rmp->mp_fs_call= PM_IDLE;

            /* Wakeup the original caller */
            setreply(rmp-mproc, OK);
            break;
        }
    }
}

```

```
case PM_SETUID:
    m.m_type= call;
    m.PM_SETUID_PROC= rmp->mp_endpoint;
    m.PM_SETUID_EGID= rmp->mp_effuid;
    m.PM_SETUID_RGID= rmp->mp_realuid;

    /* FS does not reply */
    rmp->mp_fs_call= PM_IDLE;

    /* Wakeup the original caller */
    setreply(rmp-mproc, OK);
    break;

case PM_FORK:
{
    int parent_e, parent_p;
    struct mproc *parent_mp;

    parent_p = rmp->mp_parent;
    parent_mp = &mproc[parent_p];

    m.m_type= call;
    m.PM_FORK_PPROC= parent_mp->mp_endpoint;
    m.PM_FORK_CPROC= rmp->mp_endpoint;
    m.PM_FORK_CPID= rmp->mp_pid;

    /* FS does not reply */
    rmp->mp_fs_call= PM_IDLE;

    /* Wakeup the newly created process */
    setreply(rmp-mproc, OK);

    /* Wakeup the parent */
    setreply(parent_mp-mproc, rmp->mp_pid);
    break;
}

case PM_EXIT:
case PM_EXIT_TR:
    m.m_type= call;
    m.PM_EXIT_PROC= rmp->mp_endpoint;

    /* Mark the process as busy */
    rmp->mp_fs_call= PM_BUSY;

    break;

case PM_UNPAUSE:
    m.m_type= call;
    m.PM_UNPAUSE_PROC= rmp->mp_endpoint;

    /* FS does not reply */
    rmp->mp_fs_call2= PM_IDLE;

    /* Ask the kernel to deliver the signal */
    r= sys_sigsend(rmp->mp_endpoint,
        &rmp->mp_sigmsg);
    if (r != OK)
        panic(__FILE__, "sys_sigsend failed", r);

    break;

case PM_UNPAUSE_TR:
    m.m_type= call;
    m.PM_UNPAUSE_PROC= rmp->mp_endpoint;

    /* FS does not reply */
    rmp->mp_fs_call= PM_IDLE;

    break;

case PM_EXEC:
    m.m_type= call;
    m.PM_EXEC_PROC= rmp->mp_endpoint;
```

```

        m.PM_EXEC_PATH= rmp->mp_exec_path;
        m.PM_EXEC_PATH_LEN= rmp->mp_exec_path_len;
        m.PM_EXEC_FRAME= rmp->mp_exec_frame;
        m.PM_EXEC_FRAME_LEN= rmp->mp_exec_frame_len;

        /* Mark the process as busy */
        rmp->mp_fs_call= PM_BUSY;

        break;

case PM_FORK_NB:
{
    int parent_e, parent_p;
    struct mproc *parent_mp;

    parent_p = rmp->mp_parent;
    parent_mp = &mproc[parent_p];

    m.m_type= PM_FORK;
    m.PM_FORK_PPROC= parent_mp->mp_endpoint;
    m.PM_FORK_CPROC= rmp->mp_endpoint;
    m.PM_FORK_CPID= rmp->mp_pid;

    /* FS does not reply */
    rmp->mp_fs_call= PM_IDLE;

    break;
}

case PM_DUMP_CORE:
    m.m_type= call;
    m.PM_CORE_PROC= rmp->mp_endpoint;
    m.PM_CORE_SEGPTR= (char *)rmp->mp_seg;

    /* Mark the process as busy */
    rmp->mp_fs_call= PM_BUSY;

    break;

default:
    printf("send_work: should report call 0x%x to FS\n",
           call);
    break;
}
break;
}
if (m.m_type != PM_IDLE)
{
    if (rmp->mp_fs_call == PM_IDLE &&
        rmp->mp_fs_call2 == PM_IDLE &&
        (rmp->mp_flags & PM_SIG_PENDING))
    {
        rmp->mp_flags &= ~PM_SIG_PENDING;
        check_pending(rmp);
        if (!(rmp->mp_flags & PM_SIG_PENDING))
        {
            /* Allow the process to be scheduled */
            sys_nice(rmp->mp_endpoint, rmp->mp_nice);
        }
    }
}
else if (report_reboot)
{
    m.m_type= PM_REBOOT;
    report_reboot= FALSE;
}
r= send(FS_PROC_NR, &m);
if (r != OK) panic("pm", "send_work: send failed", r);
}

PRIVATE void handle_fs_reply(m_ptr)
message *m_ptr;
{
    int r, proc_e, proc_n;

```

```

struct mproc *rmp;

switch(m_ptr->m_type)
{
case PM_EXIT_REPLY:
case PM_EXIT_REPLY_TR:
    proc_e= m_ptr->PM_EXIT_PROC;
    if (pm_isokendpt(proc_e, &proc_n) != OK)
    {
        panic(__FILE__,
              "PM_EXIT_REPLY: got bad endpoint from FS",
              proc_e);
    }
    rmp= &mproc[proc_n];

    /* Call is finished */
    rmp->mp_fs_call= PM_IDLE;

    if (!(rmp->mp_flags & PRIV_PROC))
    {
        /* destroy the (user) process */
        if((r=sys_exit(proc_e)) != OK)
        {
            panic(__FILE__,
                  "PM_EXIT_REPLY: sys_exit failed", r);
        }
    }

    /* Release the memory occupied by the child. */
    if (find_share(rmp, rmp->mp_ino, rmp->mp_dev,
                  rmp->mp_ctime) == NULL) {
        /* No other process shares the text segment,
         * so free it.
         */
        free_mem(rmp->mp_seg[T].mem_phys,
                 rmp->mp_seg[T].mem_len);
    }
    /* Free the data and stack segments. */
    free_mem(rmp->mp_seg[D].mem_phys, rmp->mp_seg[S].mem_vir +
             rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);

    if (m_ptr->m_type == PM_EXIT_REPLY_TR &&
        rmp->mp_parent != INIT_PROC_NR)
    {
        /* Wake up the parent */
        mproc[rmp->mp_parent].mp_reply.reply_trace = 0;
        setreply(rmp->mp_parent, OK);
    }

    /* Clean up if the parent has collected the exit
     * status
     */
    if (!(rmp->mp_flags & ZOMBIE))
        real_cleanup(rmp);

    break;

case PM_REBOOT_REPLY:
{
    vir_bytes code_addr;
    size_t code_size;

    /* Ask the kernel to abort. All system services, including
     * the PM, will get a HARD_STOP notification. Await the
     * notification in the main loop.
     */
    code_addr = (vir_bytes) monitor_code;
    code_size = strlen(monitor_code) + 1;
    sys_abort(abort_flag, PM_PROC_NR, code_addr, code_size);
    break;
}

case PM_EXEC_REPLY:
    proc_e= m_ptr->PM_EXEC_PROC;

```

```

    if (pm_isokendpt(proc_e, &proc_n) != OK)
    {
        panic(__FILE__,
              "PM_EXIT_REPLY: got bad endpoint from FS",
              proc_e);
    }
    rmp = &mproc[proc_n];

    /* Call is finished */
    rmp->mp_fs_call = PM_IDLE;

    exec_restart(rmp, m_ptr->PM_EXEC_STATUS);

    if (rmp->mp_flags & PM_SIG_PENDING)
    {
        printf("handle_fs_reply: restarting signals\n");
        rmp->mp_flags &= ~PM_SIG_PENDING;
        check_pending(rmp);
        if (!(rmp->mp_flags & PM_SIG_PENDING))
        {
            printf("handle_fs_reply: calling sys_nice\n");
            /* Allow the process to be scheduled */
            sys_nice(rmp->mp_endpoint, rmp->mp_nice);
        }
        else
            printf("handle_fs_reply: more signals\n");
    }
    break;
}

case PM_CORE_REPLY:
{
    int parent_waiting, right_child;
    pid_t pidarg;
    struct mproc *p_mp;

    proc_e = m_ptr->PM_CORE_PROC;
    if (pm_isokendpt(proc_e, &proc_n) != OK)
    {
        panic(__FILE__,
              "PM_EXIT_REPLY: got bad endpoint from FS",
              proc_e);
    }
    rmp = &mproc[proc_n];

    if (m_ptr->PM_CORE_STATUS == OK)
        rmp->mp_sigstatus |= DUMPED;

    /* Call is finished */
    rmp->mp_fs_call = PM_IDLE;

    p_mp = &mproc[rmp->mp_parent]; /* process' parent */
    pidarg = p_mp->mp_wpid; /* who's being waited for? */
    parent_waiting = p_mp->mp_flags & WAITING;
    right_child = /* child meets one of the 3 tests? */
        (pidarg == -1 || pidarg == rmp->mp_pid ||
         -pidarg == rmp->mp_progrp);

    if (parent_waiting && right_child) {
        tell_parent(rmp); /* tell parent */
    } else {
        /* parent not waiting, zombieify child */
        rmp->mp_flags &= (IN_USE|PRIV_PROC);
        rmp->mp_flags |= ZOMBIE;
        /* send parent a "child died" signal */
        sig_proc(p_mp, SIGCHLD);
    }

    if (!(rmp->mp_flags & PRIV_PROC))
    {
        /* destroy the (user) process */
        if((r=sys_exit(proc_e)) != OK)
        {
            panic(__FILE__,
                  "PM_CORE_REPLY: sys_exit failed", r);
        }
    }
}

```

```
        }
    }

    /* Release the memory occupied by the child. */
    if (find_share(rmp, rmp->mp_ino, rmp->mp_dev,
        rmp->mp_ctime) == NULL) {
        /* No other process shares the text segment,
         * so free it.
         */
        free_mem(rmp->mp_seg[T].mem_phys,
            rmp->mp_seg[T].mem_len);
    }
    /* Free the data and stack segments. */
    free_mem(rmp->mp_seg[D].mem_phys, rmp->mp_seg[S].mem_vir +
        rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);

    /* Clean up if the parent has collected the exit
     * status
     */
    if (!(rmp->mp_flags & ZOMBIE))
        real_cleanup(rmp);

    break;
}
default:
    panic(__FILE__, "handle_fs_reply: unknown reply type",
        m_ptr->m_type);
    break;
}
}
```



```

/* Miscellaneous system calls.                                     Author: Kees J. Bot
                                                                31 Mar 2000
*/
The entry points into this file are:
do_reboot: kill all processes, then reboot system
do_procstat: request process status (Jorrit N. Herder)
do_getsysinfo: request copy of PM data structure (Jorrit N. Herder)
do_getprocnr: lookup process slot number (Jorrit N. Herder)
do_allocmem: allocate a chunk of memory (Jorrit N. Herder)
do_freemem: deallocate a chunk of memory (Jorrit N. Herder)
do_getsetpriority: get/set process priority
do_svrctl: process manager control
*/

#include "pm.h"
#include <minix/callnr.h>
#include <signal.h>
#include <sys/svrctl.h>
#include <sys/resource.h>
#include <sys/utsname.h>
#include <minix/com.h>
#include <minix/config.h>
#include <minix/type.h>
#include <string.h>
#include <lib.h>
#include "mproc.h"
#include "param.h"
#include "../kernel/proc.h"

PUBLIC struct utsname uts_val = {
    "Minix",                /* system name */
    "noname",               /* node/network name */
    OS_RELEASE,             /* O.S. release (e.g. 1.5) */
    OS_VERSION,             /* O.S. version (e.g. 10) */
    "xyzy",                 /* machine (cpu) type (filled in later) */
#ifdef i386
    "i386",                 /* architecture */
#else
#error                     /* oops, no 'uname -mk' */
#endif
};

PRIVATE char *uts_tbl[] = {
    uts_val.arch,
    NULL,                   /* No kernel architecture */
    uts_val.machine,
    NULL,                   /* No hostname */
    uts_val.nodename,
    uts_val.release,
    uts_val.version,
    uts_val.sysname,
    NULL,                   /* No bus */
    /* No bus */
};

=====
do_allocmem
=====
PUBLIC int do_allocmem()
{
    vir_clicks mem_clicks;
    phys_clicks mem_base;

    /* This call is dangerous. Memory will be lost if the requesting process
       forgets about it.
    */
    if (mp->mp_effuid != 0)
    {
        printf("PM: unauthorized call of do_allocmem by proc %d\n",
            mp->mp_endpoint);
        return EPERM;
    }

    mem_clicks = (m_in.memsize + CLICK_SIZE - 1) >> CLICK_SHIFT;
    mem_base = alloc_mem(mem_clicks);
    if (mem_base == NO_MEM) return(ENOMEM);
}

```

```

mp->mp_reply.membase = (phys_bytes) (mem_base << CLICK_SHIFT);
return(OK);
}

/*=====
*
* do_freemem
*=====*/
PUBLIC int do_freemem()
{
    vir_clicks mem_clicks;
    phys_clicks mem_base;

    /* This call is dangerous. Even memory belonging to other processes can
     * be freed.
     */
    if (mp->mp_effuid != 0)
    {
        printf("PM: unauthorized call of do_freemem by proc %d\n",
            mp->mp_endpoint);
        return EPERM;
    }

    mem_clicks = (m_in.memsize + CLICK_SIZE - 1) >> CLICK_SHIFT;
    mem_base = (m_in.membase + CLICK_SIZE - 1) >> CLICK_SHIFT;
    free_mem(mem_base, mem_clicks);
    return(OK);
}

/*=====
*
* do_procstat
*=====*/
PUBLIC int do_procstat()
{
    /* For the moment, this is only used to return pending signals to
     * system processes that request the PM for their own status.
     *
     * Future use might include the FS requesting for process status of
     * any user process.
     */

    /* This call should be removed, or made more general. */
    if (mp->mp_effuid != 0)
    {
        printf("PM: unauthorized call of do_procstat by proc %d\n",
            mp->mp_endpoint);
        return EPERM;
    }

    if (m_in.stat_nr == SELF) {
        mp->mp_reply.sig_set = mp->mp_sigpending;
        sigemptyset(&mp->mp_sigpending);
    }
    else {
        return(ENOSYS);
    }
    return(OK);
}

/*=====
*
* do_sysuname
*=====*/
PUBLIC int do_sysuname()
{
    /* Set or get uname strings. */

    int r;
    size_t n, len;
    char *string, *t;
    #if 0 /* for updates */
    char tmp[sizeof(uts_val.nodename)];
    static short sizes[] = {
        0, /* arch, (0 = read-only) */
        0, /* kernel */
        0, /* machine */
    }

```

```

    0,      /* sizeof(uts_val.hostname), */
    sizeof(uts_val.nodename),
    0,      /* release */
    0,      /* version */
    0,      /* sysname */
};
#endif

if ((unsigned) m_in.sysuname_field >= _UTS_MAX) return(EINVAL);

string = uts_tbl[m_in.sysuname_field];
if (string == NULL)
    return EINVAL; /* Unsupported field */

switch (m_in.sysuname_req) {
case _UTS_GET:
    /* Copy an uname string to the user. */
    n = strlen(string) + 1;
    if (n > m_in.sysuname_len) n = m_in.sysuname_len;
    r = sys_vircopy(SELF, D, (phys_bytes) string,
        mp->mp_endpoint, D, (phys_bytes) m_in.sysuname_value,
        (phys_bytes) n);
    if (r < 0) return(r);
    break;
#if 0 /* no updates yet */
case _UTS_SET:
    /* Set an uname string, needs root power. */
    len = sizes[m_in.sysuname_field];
    if (mp->mp_effuid != 0 || len == 0) return(EPERM);
    n = len < m_in.sysuname_len ? len : m_in.sysuname_len;
    if (n <= 0) return(EINVAL);
    r = sys_vircopy(mp->mp_endpoint, D, (phys_bytes) m_in.sysuname_value,
        SELF, D, (phys_bytes) tmp, (phys_bytes) n);
    if (r < 0) return(r);
    tmp[n-1] = 0;
    strcpy(string, tmp);
    break;
#endif

default:
    return(EINVAL);
}
/* Return the number of bytes moved. */
return(n);
}

/*=====
 *                               do_getsysinfo                               *
 *=====*/
PUBLIC int do_getsysinfo()
{
    struct mproc *proc_addr;
    vir_bytes src_addr, dst_addr;
    struct kinfo kinfo;
    struct loadinfo loadinfo;
    static struct proc proctab[NR_PROCS+NR_TASKS];
    size_t len;
    static struct pm_mem_info pmi;
    int s, r;
    size_t holesize;

    /* This call leaks important information (the contents of registers).
     * harmless data (such as the load should get their own calls)
     */
    if (mp->mp_effuid != 0)
    {
        printf("PM: unauthorized call of do_getsysinfo by proc %d '%s'\n",
            mp->mp_endpoint, mp->mp_name);
        sig_proc(mp, SIGEMT);
        return EPERM;
    }
}

```

```

switch(m_in.info_what) {
case SI_KINFO:                                /* kernel info is obtained via PM */
    sys_getkinfo(&kinfo);
    src_addr = (vir_bytes) &kinfo;
    len = sizeof(struct kinfo);
    break;
case SI_PROC_ADDR:                            /* get address of PM process table */
    proc_addr = &mproc[0];
    src_addr = (vir_bytes) &proc_addr;
    len = sizeof(struct mproc *);
    break;
case SI_PROC_TAB:                             /* copy entire process table */
    src_addr = (vir_bytes) mproc;
    len = sizeof(struct mproc) * NR_PROCS;
    break;
case SI_KPROC_TAB:                            /* copy entire process table */
    if((r=sys_getproctab(proctab)) != OK)
        return r;
    src_addr = (vir_bytes) proctab;
    len = sizeof(proctab);
    break;
case SI_MEM_ALLOC:
    holesize = sizeof(pmi.pmi_holes);
    if((r=mem_holes_copy(pmi.pmi_holes, &holesize,
        &pmi.pmi_hi_watermark)) != OK)
        return r;
    src_addr = (vir_bytes) &pmi;
    len = sizeof(pmi);
    break;
case SI_LOADINFO:                             /* loadinfo is obtained via PM */
    sys_getloadinfo(&loadinfo);
    src_addr = (vir_bytes) &loadinfo;
    len = sizeof(struct loadinfo);
    break;
default:
    return(EINVAL);
}

dst_addr = (vir_bytes) m_in.info_where;
if (OK != (s=sys_datacopy(SELF, src_addr, who_e, dst_addr, len)))
    return(s);
return(OK);
}

/*=====
*                                     do_getprocnr                                     *
*=====*/
PUBLIC int do_getprocnr()
{
    register struct mproc *rmp;
    static char search_key[PROC_NAME_LEN+1];
    int key_len;
    int s;

    /* This call should be moved to DS. */
    if (mp->mp_effuid != 0)
    {
        printf("PM: unauthorized call of do_procstat by proc %d\n",
            mp->mp_endpoint);
        return EPERM;
    }

    if (m_in.pid >= 0) {
        /* lookup process by pid */
        for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
            if ((rmp->mp_flags & IN_USE) && (rmp->mp_pid==m_in.pid)) {
                mp->mp_reply.endpt = rmp->mp_endpoint;
                return(OK);
            }
        }
        return(ESRCH);
    } else if (m_in.namelen > 0) {
        /* lookup process by name */
        key_len = MIN(m_in.namelen, PROC_NAME_LEN);
        if (OK != (s=sys_datacopy(who_e, (vir_bytes) m_in.addr,
            SELF, (vir_bytes) search_key, key_len)))

```

```

        return(s);
    search_key[key_len] = '\0';      /* terminate for safety */
    for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
        if (((rmp->mp_flags & (IN_USE | ZOMBIE)) == IN_USE) &&
            strncmp(rmp->mp_name, search_key, key_len)==0) {
            mp->mp_reply.endpt = rmp->mp_endpoint;
            return(OK);
        }
    }
    return(ESRCH);
} else {
    /* return own/parent process number */
    mp->mp_reply.endpt = who_e;
    mp->mp_reply.pendpt = mproc[mp->mp_parent].mp_endpoint;
}

return(OK);
}

/*=====
 *                               do_reboot                               *
 *=====*/
PUBLIC int do_reboot()
{
    int r;

    /* Check permission to abort the system. */
    if (mp->mp_effuid != SUPER_USER) return(EPERM);

    /* See how the system should be aborted. */
    abort_flag = (unsigned) m_in.reboot_flag;
    if (abort_flag >= RBT_INVALID) return(EINVAL);
    if (RBT_MONITOR == abort_flag) {
        int r;
        if (m_in.reboot_strlen >= sizeof(monitor_code))
            return EINVAL;
        if ((r = sys_datacopy(who_e, (vir_bytes) m_in.reboot_code,
            SELF, (vir_bytes) monitor_code, m_in.reboot_strlen)) != OK)
            return r;
        monitor_code[m_in.reboot_strlen] = '\0';
    }
    else
        monitor_code[0] = '\0';

    /* Order matters here. When FS is told to reboot, it exits all its
     * processes, and then would be confused if they're exited again by
     * SIGKILL. So first kill, then reboot.
     */

    check_sig(-1, SIGKILL);          /* kill all users except init */
    sys_nice(INIT_PROC_NR, PRIO_STOP); /* stop init, but keep it around */

    report_reboot= 1;
    r= notify(FS_PROC_NR);
    if (r != OK) panic("pm", "do_reboot: unable to notify FS", r);

    return(SUSPEND);                /* don't reply to caller */
}

/*=====
 *                               do_getsetpriority                       *
 *=====*/
PUBLIC int do_getsetpriority()
{
    int arg_which, arg_who, arg_pri;
    int rmp_nr;
    struct mproc *rmp;

    arg_which = m_in.ml_i1;
    arg_who = m_in.ml_i2;
    arg_pri = m_in.ml_i3;  /* for SETPRIORITY */

    /* Code common to GETPRIORITY and SETPRIORITY. */

    /* Only support PRIO_PROCESS for now. */

```

```

    if (arg_which != PRIO_PROCESS)
        return(EINVAL);

    if (arg_who == 0)
        rmp_nr = who_p;
    else
        if ((rmp_nr = proc_from_pid(arg_who)) < 0)
            return(ESRCH);

    rmp = &mproc[rmp_nr];

    if (mp->mp_effuid != SUPER_USER &&
        mp->mp_effuid != rmp->mp_effuid && mp->mp_effuid != rmp->mp_realuid)
        return EPERM;

    /* If GET, that's it. */
    if (call_nr == GETPRIORITY) {
        return(rmp->mp_nice - PRIO_MIN);
    }

    /* Only root is allowed to reduce the nice level. */
    if (rmp->mp_nice > arg_pri && mp->mp_effuid != SUPER_USER)
        return(EACCES);

    /* We're SET, and it's allowed. Do it and tell kernel. */
    rmp->mp_nice = arg_pri;
    return sys_nice(rmp->mp_endpoint, arg_pri);
}

/*=====
 *                               do_svrctl                               *
 *=====*/
PUBLIC int do_svrctl()
{
    int s, req;
    vir_bytes ptr;
#define MAX_LOCAL_PARAMS 2
    static struct {
        char name[30];
        char value[30];
    } local_param_overrides[MAX_LOCAL_PARAMS];
    static int local_params = 0;

    req = m_in.svrctl_req;
    ptr = (vir_bytes) m_in.svrctl_argp;

    /* Is the request indeed for the MM? */
    if (((req >> 8) & 0xFF) != 'M') return(EINVAL);

    /* Control operations local to the PM. */
    switch(req) {
    case MMSETPARAM:
    case MMGETPARAM: {
        struct sysgetenv sysgetenv;
        char search_key[64];
        char *val_start;
        size_t val_len;
        size_t copy_len;

        /* Copy sysgetenv structure to PM. */
        if (sys_datacopy(who_e, ptr, SELF, (vir_bytes) &sysgetenv,
            sizeof(sysgetenv)) != OK) return(EFAULT);

        /* Set a param override? */
        if (req == MMSETPARAM) {
            if (local_params >= MAX_LOCAL_PARAMS) return ENOSPC;
            if (sysgetenv.keylen <= 0
                || sysgetenv.keylen >=
                    sizeof(local_param_overrides[local_params].name)
                || sysgetenv.vallen <= 0
                || sysgetenv.vallen >=
                    sizeof(local_param_overrides[local_params].value))
                return EINVAL;

```

```

        if ((s = sys_datacopy(who_e, (vir_bytes) sysgetenv.key,
            SELF, (vir_bytes) local_param_overrides[local_params].name,
            sysgetenv.keylen)) != OK)
            return s;
        if ((s = sys_datacopy(who_e, (vir_bytes) sysgetenv.val,
            SELF, (vir_bytes) local_param_overrides[local_params].value,
            sysgetenv.keylen)) != OK)
            return s;
        local_param_overrides[local_params].name[sysgetenv.keylen] = '\0';
        local_param_overrides[local_params].value[sysgetenv.vallen] = '\0';

        local_params++;

    return OK;
}

if (sysgetenv.keylen == 0) {          /* copy all parameters */
    val_start = monitor_params;
    val_len = sizeof(monitor_params);
}
else {                                /* lookup value for key */
    int p;
    /* Try to get a copy of the requested key. */
    if (sysgetenv.keylen > sizeof(search_key)) return(EINVAL);
    if ((s = sys_datacopy(who_e, (vir_bytes) sysgetenv.key,
        SELF, (vir_bytes) search_key, sysgetenv.keylen)) != OK)
        return(s);

    /* Make sure key is null-terminated and lookup value.
     * First check local overrides.
     */
    search_key[sysgetenv.keylen-1] = '\0';
    for(p = 0; p < local_params; p++) {
        if (!strcmp(search_key, local_param_overrides[p].name)) {
            val_start = local_param_overrides[p].value;
            break;
        }
    }
    if (p >= local_params && (val_start = find_param(search_key)) == NULL)
        return(ESRCH);
    val_len = strlen(val_start) + 1;
}

/* See if it fits in the client's buffer. */
if (val_len > sysgetenv.vallen)
    return E2BIG;

/* Value found, make the actual copy (as far as possible). */
copy_len = MIN(val_len, sysgetenv.vallen);
if ((s=sys_datacopy(SELF, (vir_bytes) val_start,
    who_e, (vir_bytes) sysgetenv.val, copy_len)) != OK)
    return(s);

return OK;
}

#ifdef ENABLE_SWAP
case MMSWAPON: {
    struct mmswapon swapon;

    if (mp->mp_effuid != SUPER_USER) return(EPERM);

    if (sys_datacopy(who_e, (phys_bytes) ptr,
        PM_PROC_NR, (phys_bytes) &swapon,
        (phys_bytes) sizeof(swapon)) != OK) return(EFAULT);

    return(swap_on(swapon.file, swapon.offset, swapon.size)); }

case MMSWAPOFF: {
    if (mp->mp_effuid != SUPER_USER) return(EPERM);

    return(swap_off()); }
#endif /* SWAP */

```

```
default:
    return(EINVAL);
}
```



```

/* This table has one slot per process. It contains all the process management
 * information for each process. Among other things, it defines the text, data
 * and stack segments, uids and gids, and various flags. The kernel and file
 * systems have tables that are also indexed by process, with the contents
 * of corresponding slots referring to the same process in all three.
 */
#include <timers.h>

EXTERN struct mproc {
    struct mem_map mp_seg[NR_LOCAL_SEGS]; /* points to text, data, stack */
    char mp_exitstatus; /* storage for status when process exits */
    char mp_sigstatus; /* storage for signal # for killed procs */
    pid_t mp_pid; /* process id */
    int mp_endpoint; /* kernel endpoint id */
    pid_t mp_progrp; /* pid of process group (used for signals) */
    pid_t mp_wpid; /* pid this process is waiting for */
    int mp_parent; /* index of parent process */

    /* Child user and system times. Accounting done on child exit. */
    clock_t mp_child_utime; /* cumulative user time of children */
    clock_t mp_child_stime; /* cumulative sys time of children */

    /* Real and effective uids and gids. */
    uid_t mp_realuid; /* process' real uid */
    uid_t mp_effuid; /* process' effective uid */
    gid_t mp_realgid; /* process' real gid */
    gid_t mp_effgid; /* process' effective gid */

    /* File identification for sharing. */
    ino_t mp_ino; /* inode number of file */
    dev_t mp_dev; /* device number of file system */
    time_t mp_ctime; /* inode changed time */

    /* Signal handling information. */
    sigset_t mp_ignore; /* 1 means ignore the signal, 0 means don't */
    sigset_t mp_catch; /* 1 means catch the signal, 0 means don't */
    sigset_t mp_sig2mess; /* 1 means transform into notify message */
    sigset_t mp_sigmask; /* signals to be blocked */
    sigset_t mp_sigmask2; /* saved copy of mp_sigmask */
    sigset_t mp_sigpending; /* pending signals to be handled */
    struct sigaction mp_sigact[_NSIG + 1]; /* as in sigaction(2) */
    vir_bytes mp_sigreturn; /* address of C library __sigreturn function */
    struct sigmsg mp_sigmsg; /* Save the details of the signal until the
    * PM_UNPAUSE request is delivered.
    */
    struct timer mp_timer; /* watchdog timer for alarm(2) */

    /* Backwards compatibility for signals. */
    sighandler_t mp_func; /* all sigs vectored to a single user fcn */

    unsigned mp_flags; /* flag bits */
    vir_bytes mp_proargs; /* ptr to proc's initial stack arguments */
    struct mproc *mp_swapq; /* queue of procs waiting to be swapped in */
    message mp_reply; /* reply message to be sent to one */

    /* Communication with FS */
    int mp_fs_call; /* Record the call for normal system calls */
    int mp_fs_call2; /* Record the call for signals */
    char *mp_exec_path; /* Path of executable */
    vir_bytes mp_exec_path_len; /* Length of path (including nul) */
    char *mp_exec_frame; /* Arguments */
    vir_bytes mp_exec_frame_len; /* Length of arguments */

    /* Scheduling priority. */
    signed int mp_nice; /* nice is PRIO_MIN..PRIO_MAX, standard 0. */

    char mp_name[PROC_NAME_LEN]; /* process name */
} mproc[NR_PROCS];

/* Flag values */
#define IN_USE 0x001 /* set when 'mproc' slot in use */
#define WAITING 0x002 /* set by WAIT system call */
#define ZOMBIE 0x004 /* set by EXIT, cleared by WAIT */
#define PAUSED 0x008 /* set by PAUSE system call */

```

```
#define ALARM_ON      0x010  /* set when SIGALRM timer started */
#define SEPARATE      0x020  /* set if file is separate I & D space */
#define TRACED        0x040  /* set if process is to be traced */
#define STOPPED       0x080  /* set if process stopped for tracing */
#define SIGSUSPENDED  0x100  /* set by SIGSUSPEND system call */
#define REPLY         0x200  /* set if a reply message is pending */
#define ONSWAP        0x400  /* set if data segment is swapped out */
#define SWAPIN        0x800  /* set if on the "swap this in" queue */
#define DONT_SWAP     0x1000 /* never swap out this process */
#define PRIV_PROC     0x2000 /* system process, special privileges */
#define PM_SIG_PENDING 0x4000 /* process got a signal while waiting for FS */
#define PARTIAL_EXEC  0x8000 /* Process got a new map but no content */

#define NIL_MPROC ((struct mproc *) 0)
```

```
/* The following names are synonyms for the variables in the input message. */
#define addr m1_p1
#define exec_name m1_p1
#define exec_len m1_i1
#define func m6_f1
#define grp_id m1_i1
#define namelen m1_i2
#define pid m1_i1
#define endpt m1_i1
#define pendpt m1_i2
#define seconds m1_i1
#define sig m6_i1
#define stack_bytes m1_i2
#define stack_ptr m1_p2
#define status m1_i1
#define usr_id m1_i1
#define request m2_i2
#define taddr m2_l1
#define data m2_l2
#define sig_nr m1_i2
#define sig_nsa m1_p1
#define sig_osa m1_p2
#define sig_ret m1_p3
#define stat_nr m2_i1
#define sig_set m2_l1
#define sig_how m2_i1
#define sig_flags m2_i2
#define sig_context m2_p1
#ifdef _SIGMESSAGE
#define sig_msg m1_i1
#endif
#define info_what m1_i1
#define info_where m1_p1
#define reboot_flag m1_i1
#define reboot_code m1_p1
#define reboot_strlen m1_i2
#define svrctl_req m2_i1
#define svrctl_argp m2_p1
#define stime m2_l1
#define memsize m4_l1
#define membase m4_l2
#define sysuname_req m1_i1
#define sysuname_field m1_i2
#define sysuname_len m1_i3
#define sysuname_value m1_p1

/* The following names are synonyms for the variables in a reply message. */
#define reply_res m_type
#define reply_res2 m2_i1
#define reply_res3 m2_i2
#define reply_ptr m2_p1
#define reply_mask m2_l1
#define reply_trace m2_l2
#define reply_time m2_l1
#define reply_utime m2_l2
#define reply_t1 m4_l1
#define reply_t2 m4_l2
#define reply_t3 m4_l3
#define reply_t4 m4_l4
#define reply_t5 m4_l5

/* The following names are used to inform the FS about certain events. */
#define tell_fs_arg1 m1_i1
#define tell_fs_arg2 m1_i2
#define tell_fs_arg3 m1_i3
```

```
/* This is the master header for PM. It includes some other files
 * and defines the principal constants.
 */
#define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
#define _MINIX             1    /* tell headers to include MINIX stuff */
#define _SYSTEM           1    /* tell headers that this is the kernel */

/* The following are so basic, all the *.c files get them automatically. */
#include <minix/config.h>      /* MUST be first */
#include <ansi.h>              /* MUST be second */
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>

#include <fcntl.h>
#include <unistd.h>
#include <minix/syslib.h>
#include <minix/sysutil.h>

#include <limits.h>
#include <errno.h>

#include "const.h"
#include "type.h"
#include "proto.h"
#include "glo.h"
```

```

/* Function prototypes. */

struct mproc;
struct stat;
struct mem_map;
struct memory;

#include <timers.h>

/* alloc.c */
_PROTOTYPE( phys_clicks alloc_mem, (phys_clicks clicks) ) ;
_PROTOTYPE( void free_mem, (phys_clicks base, phys_clicks clicks) ) ;
_PROTOTYPE( void mem_init, (struct memory *chunks, phys_clicks *free) ) ;
#if ENABLE_SWAP
_PROTOTYPE( int swap_on, (char *file, u32_t offset, u32_t size) ) ;
_PROTOTYPE( int swap_off, (void) ) ;
_PROTOTYPE( void swap_in, (void) ) ;
_PROTOTYPE( void swap_inqueue, (struct mproc *rmp) ) ;
#else /* !SWAP */
#define swap_in() ((void)0)
#define swap_inqueue(rmp) ((void)0)
#endif /* !SWAP */
_PROTOTYPE( int mem_holes_copy, (struct hole *, size_t *, u32_t *) ) ;

/* break.c */
_PROTOTYPE( int adjust, (struct mproc *rmp,
                        vir_clicks data_clicks, vir_bytes sp) ) ;
_PROTOTYPE( int do_brk, (void) ) ;
_PROTOTYPE( int size_ok, (int file_type, vir_clicks tc, vir_clicks dc,
                        vir_clicks sc, vir_clicks dvir, vir_clicks s_vir) ) ;

/* devio.c */
_PROTOTYPE( int do_dev_io, (void) ) ;
_PROTOTYPE( int do_dev_io, (void) ) ;

/* dmp.c */
_PROTOTYPE( int do_fkey_pressed, (void) ) ;

/* exec.c */
_PROTOTYPE( int do_exec, (void) ) ;
_PROTOTYPE( int exec_newmem, (void) ) ;
_PROTOTYPE( int do_execestart, (void) ) ;
_PROTOTYPE( void exec_restart, (struct mproc *rmp, int result) ) ;
_PROTOTYPE( struct mproc *find_share, (struct mproc *mp_ign, Ino_t ino,
                        Dev_t dev, time_t ctime) ) ;

/* forkexit.c */
_PROTOTYPE( int do_fork, (void) ) ;
_PROTOTYPE( int do_fork_nb, (void) ) ;
_PROTOTYPE( int do_pm_exit, (void) ) ;
_PROTOTYPE( int do_waitpid, (void) ) ;
_PROTOTYPE( void pm_exit, (struct mproc *rmp, int exit_status,
                        int for_trace) ) ;
_PROTOTYPE( void tell_parent, (struct mproc *child) ) ;
_PROTOTYPE( void real_cleanup, (struct mproc *rmp) ) ;

/* getset.c */
_PROTOTYPE( int do_getset, (void) ) ;

/* main.c */
_PROTOTYPE( int main, (void) ) ;

/* misc.c */
_PROTOTYPE( int do_reboot, (void) ) ;
_PROTOTYPE( int do_procstat, (void) ) ;
_PROTOTYPE( int do_sysuname, (void) ) ;
_PROTOTYPE( int do_getsysinfo, (void) ) ;
_PROTOTYPE( int do_getprocnr, (void) ) ;
_PROTOTYPE( int do_svrctl, (void) ) ;
_PROTOTYPE( int do_allocmem, (void) ) ;
_PROTOTYPE( int do_freemem, (void) ) ;
_PROTOTYPE( int do_getsetpriority, (void) ) ;

```

```
#if (MACHINE == MACINTOSH)
_PROTOTYPE( phys_clicks start_click, (void) ) ;
#endif

_PROTOTYPE( void setreply, (int proc_nr, int result) ) ;

/* signal.c */
_PROTOTYPE( int do_alarm, (void) ) ;
_PROTOTYPE( int do_kill, (void) ) ;
_PROTOTYPE( int ksig_pending, (void) ) ;
_PROTOTYPE( int do_pause, (void) ) ;
_PROTOTYPE( int set_alarm, (int proc_nr, int sec) ) ;
_PROTOTYPE( int check_sig, (pid_t proc_id, int signo) ) ;
_PROTOTYPE( void sig_proc, (struct mproc *rmp, int sig_nr) ) ;
_PROTOTYPE( int do_sigaction, (void) ) ;
_PROTOTYPE( int do_sigpending, (void) ) ;
_PROTOTYPE( int do_sigprocmask, (void) ) ;
_PROTOTYPE( int do_sigreturn, (void) ) ;
_PROTOTYPE( int do_sigsuspend, (void) ) ;
_PROTOTYPE( void check_pending, (struct mproc *rmp) ) ;

/* time.c */
_PROTOTYPE( int do_stime, (void) ) ;
_PROTOTYPE( int do_time, (void) ) ;
_PROTOTYPE( int do_times, (void) ) ;
_PROTOTYPE( int do_gettimeofday, (void) ) ;

/* timers.c */
_PROTOTYPE( void pm_set_timer, (timer_t *tp, int delta,
    tmr_func_t watchdog, int arg));
_PROTOTYPE( void pm_expire_timers, (clock_t now));
_PROTOTYPE( void pm_cancel_timer, (timer_t *tp));

/* trace.c */
_PROTOTYPE( int do_trace, (void) ) ;
_PROTOTYPE( void stop_proc, (struct mproc *rmp, int sig_nr) ) ;

/* utility.c */
_PROTOTYPE( pid_t get_free_pid, (void) ) ;
_PROTOTYPE( int no_sys, (void) ) ;
_PROTOTYPE( void panic, (char *who, char *mess, int num) ) ;
_PROTOTYPE( int get_stack_ptr, (int proc_nr, vir_bytes *sp) ) ;
_PROTOTYPE( int get_mem_map, (int proc_nr, struct mem_map *mem_map) ) ;
_PROTOTYPE( char *find_param, (const char *key));
_PROTOTYPE( int proc_from_pid, (pid_t p));
_PROTOTYPE( int pm_isokendpt, (int ep, int *proc));
```

```

/* This file handles signals, which are asynchronous events and are generally
 * a messy and unpleasant business.  Signals can be generated by the KILL
 * system call, or from the keyboard (SIGINT) or from the clock (SIGALRM).
 * In all cases control eventually passes to check_sig() to see which processes
 * can be signaled.  The actual signaling is done by sig_proc().
 *
 * The entry points into this file are:
 * do_sigaction: perform the SIGACTION system call
 * do_sigpending: perform the SIGPENDING system call
 * do_sigprocmask: perform the SIGPROCMASK system call
 * do_sigreturn: perform the SIGRETURN system call
 * do_sigsuspend: perform the SIGSUSPEND system call
 * do_kill: perform the KILL system call
 * do_alarm: perform the ALARM system call by calling set_alarm()
 * set_alarm: tell the clock task to start or stop a timer
 * do_pause: perform the PAUSE system call
 * ksig_pending: the kernel notified about pending signals
 * sig_proc: interrupt or terminate a signaled process
 * check_sig: check which processes to signal with sig_proc()
 * check_pending: check if a pending signal can now be delivered
 */

```

```

#include "pm.h"
#include <sys/stat.h>
#include <sys/ptrace.h>
#include <minix/callnr.h>
#include <minix/endpoint.h>
#include <minix/com.h>
#include <signal.h>
#include <sys/resource.h>
#include <sys/sigcontext.h>
#include <string.h>
#include "mproc.h"
#include "param.h"

```

```

FORWARD _PROTOTYPE( int dump_core, (struct mproc *rmp) ) ;
FORWARD _PROTOTYPE( void unpause, (int pro, int for_trace) ) ;
FORWARD _PROTOTYPE( void handle_ksig, (int proc_nr, sigset_t sig_map) ) ;
FORWARD _PROTOTYPE( void cause_sigalrm, (struct timer *tp) ) ;

```

```

/*=====
 *                               do_sigaction                               *
 *=====*/
PUBLIC int do_sigaction()
{
    int r;
    struct sigaction svec;
    struct sigaction *svp;

    if (m_in.sig_nr == SIGKILL) return(OK);
    if (m_in.sig_nr < 1 || m_in.sig_nr > _NSIG) return (EINVAL);
    svp = &m->mp_sigact[m_in.sig_nr];
    if ((struct sigaction *) m_in.sig_osa != (struct sigaction *) NULL) {
        r = sys_datacopy(PM_PROC_NR, (vir_bytes) svp,
            who_e, (vir_bytes) m_in.sig_osa, (phys_bytes) sizeof(svec));
        if (r != OK) return(r);
    }

    if ((struct sigaction *) m_in.sig_nsa == (struct sigaction *) NULL)
        return(OK);

    /* Read in the sigaction structure. */
    r = sys_datacopy(who_e, (vir_bytes) m_in.sig_nsa,
        PM_PROC_NR, (vir_bytes) &svec, (phys_bytes) sizeof(svec));
    if (r != OK) return(r);

    if (svec.sa_handler == SIG_IGN) {
        sigaddset(&m->mp_ignore, m_in.sig_nr);
        sigdelset(&m->mp_sigpending, m_in.sig_nr);
        sigdelset(&m->mp_catch, m_in.sig_nr);
        sigdelset(&m->mp_sig2mess, m_in.sig_nr);
    } else if (svec.sa_handler == SIG_DFL) {
        sigdelset(&m->mp_ignore, m_in.sig_nr);
        sigdelset(&m->mp_catch, m_in.sig_nr);
    }
}

```

```

        sigdelset(&mp->mp_sig2mess, m_in.sig_nr);
    } else if (svec.sa_handler == SIG_MESS) {
        if (! (mp->mp_flags & PRIV_PROC)) return(EPERM);
        sigdelset(&mp->mp_ignore, m_in.sig_nr);
        sigaddset(&mp->mp_sig2mess, m_in.sig_nr);
        sigdelset(&mp->mp_catch, m_in.sig_nr);
    } else {
        sigdelset(&mp->mp_ignore, m_in.sig_nr);
        sigaddset(&mp->mp_catch, m_in.sig_nr);
        sigdelset(&mp->mp_sig2mess, m_in.sig_nr);
    }
    mp->mp_sigact[m_in.sig_nr].sa_handler = svec.sa_handler;
    sigdelset(&svec.sa_mask, SIGKILL);
    mp->mp_sigact[m_in.sig_nr].sa_mask = svec.sa_mask;
    mp->mp_sigact[m_in.sig_nr].sa_flags = svec.sa_flags;
    mp->mp_sigreturn = (vir_bytes) m_in.sig_ret;
    return(OK);
}

/*=====
 *                               do_sigpending                               *
 *=====*/
PUBLIC int do_sigpending()
{
    mp->mp_reply.reply_mask = (long) mp->mp_sigpending;
    return OK;
}

/*=====
 *                               do_sigprocmask                           *
 *=====*/
PUBLIC int do_sigprocmask()
{
    /* Note that the library interface passes the actual mask in sigmask_set,
     * not a pointer to the mask, in order to save a copy. Similarly,
     * the old mask is placed in the return message which the library
     * interface copies (if requested) to the user specified address.
     *
     * The library interface must set SIG_INQUIRE if the 'act' argument
     * is NULL.
     *
     * KILL and STOP can't be masked.
     */

    int i;

    mp->mp_reply.reply_mask = (long) mp->mp_sigmask;

    switch (m_in.sig_how) {
        case SIG_BLOCK:
            sigdelset((sigset_t *)&m_in.sig_set, SIGKILL);
            sigdelset((sigset_t *)&m_in.sig_set, SIGSTOP);
            for (i = 1; i <= _NSIG; i++) {
                if (sigismember((sigset_t *)&m_in.sig_set, i))
                    sigaddset(&mp->mp_sigmask, i);
            }
            break;

        case SIG_UNBLOCK:
            for (i = 1; i <= _NSIG; i++) {
                if (sigismember((sigset_t *)&m_in.sig_set, i))
                    sigdelset(&mp->mp_sigmask, i);
            }
            check_pending(mp);
            break;

        case SIG_SETMASK:
            sigdelset((sigset_t *) &m_in.sig_set, SIGKILL);
            sigdelset((sigset_t *) &m_in.sig_set, SIGSTOP);
            mp->mp_sigmask = (sigset_t) m_in.sig_set;
            check_pending(mp);
            break;

        case SIG_INQUIRE:

```



```

        break;

    default:
        return(EINVAL);
        break;
}
return OK;
}

/*=====
 *                               do_sigsuspend                               *
 *=====*/
PUBLIC int do_sigsuspend()
{
    mp->mp_sigmask2 = mp->mp_sigmask;    /* save the old mask */
    mp->mp_sigmask = (sigset_t) m_in.sig_set;
    sigdelset(&mp->mp_sigmask, SIGKILL);
    mp->mp_flags |= SIGSUSPENDED;
    check_pending(mp);
    return(SUSPEND);
}

/*=====
 *                               do_sigreturn                               *
 *=====*/
PUBLIC int do_sigreturn()
{
    /* A user signal handler is done.  Restore context and check for
     * pending unblocked signals.
     */

    int r;

    mp->mp_sigmask = (sigset_t) m_in.sig_set;
    sigdelset(&mp->mp_sigmask, SIGKILL);

    r = sys_sigreturn(who_e, (struct sigmsg *) m_in.sig_context);
    check_pending(mp);
    return(r);
}

/*=====
 *                               do_kill                                   *
 *=====*/
PUBLIC int do_kill()
{
    /* Perform the kill(pid, signo) system call. */

    return check_sig(m_in.pid, m_in.sig_nr);
}

/*=====
 *                               ksig_pending                               *
 *=====*/
PUBLIC int ksig_pending()
{
    /* Certain signals, such as segmentation violations originate in the kernel.
     * When the kernel detects such signals, it notifies the PM to take further
     * action. The PM requests the kernel to send messages with the process
     * slot and bit map for all signaled processes. The File System, for example,
     * uses this mechanism to signal writing on broken pipes (SIGPIPE).
     *
     * The kernel has notified the PM about pending signals. Request pending
     * signals until all signals are handled. If there are no more signals,
     * NONE is returned in the process number field.
     */
    int proc_nr_e;
    sigset_t sig_map;

    while (TRUE) {
        int r;
        /* get an arbitrary pending signal */
        if((r=sys_getksig(&proc_nr_e, &sig_map)) != OK)
            panic(__FILE__, "sys_getksig failed", r);
    }
}

```

```

    if (NONE == proc_nr_e) {
        break;
    } else {
        int proc_nr_p;
        if(pm_isokendpt(proc_nr_e, &proc_nr_p) != OK)
            panic(__FILE__, "sys_getksig strange process", proc_nr_e);
        handle_ksig(proc_nr_e, sig_map);
        /* If the process still exists to the kernel after the signal
           * has been handled ...
           */
        if ((mproc[proc_nr_p].mp_flags & (IN_USE | ZOMBIE)) == IN_USE)
        {
            if((r=sys_endksig(proc_nr_e)) != OK) /* ... tell kernel it's done */
                panic(__FILE__, "sys_endksig failed", r);
        }
    }
}
return(SUSPEND);
}

/*=====
*                                     handle_ksig                                     *
*=====*/
PRIVATE void handle_ksig(proc_nr_e, sig_map)
int proc_nr_e;
sigset_t sig_map;
{
    register struct mproc *rmp;
    int i, proc_nr;
    pid_t proc_id, id;

    if(pm_isokendpt(proc_nr_e, &proc_nr) != OK || proc_nr < 0)
        return;
    rmp = &mproc[proc_nr];
    if ((rmp->mp_flags & (IN_USE | ZOMBIE)) != IN_USE)
        return;
    proc_id = rmp->mp_pid;
    mp = &mproc[0];
    mp->mp_procgrp = rmp->mp_procgrp;

    /* Check each bit in turn to see if a signal is to be sent. Unlike
     * kill(), the kernel may collect several unrelated signals for a
     * process and pass them to PM in one blow. Thus loop on the bit
     * map. For SIGINT, SIGWINCH and SIGQUIT, use proc_id 0 to indicate
     * a broadcast to the recipient's process group. For SIGKILL, use
     * proc_id -1 to indicate a systemwide broadcast.
     */
    for (i = 1; i <= _NSIG; i++) {
        if (!sigismember(&sig_map, i)) continue;
        switch (i) {
            case SIGINT:
            case SIGQUIT:
            case SIGWINCH:
                id = 0; break; /* broadcast to process group */
            default:
                id = proc_id;
                break;
        }
        check_sig(id, i);
    }
}

/*=====
*                                     do_alarm                                     *
*=====*/
PUBLIC int do_alarm()
{
    /* Perform the alarm(seconds) system call. */
    return(set_alarm(who_e, m_in.seconds));
}

/*=====
*                                     set_alarm                                     *
*=====*/

```

```

PUBLIC int set_alarm(proc_nr_e, sec)
int proc_nr_e;           /* process that wants the alarm */
int sec;                 /* how many seconds delay before the signal */
{
/* This routine is used by do_alarm() to set the alarm timer. It is also used
 * to turn the timer off when a process exits with the timer still on.
 */
    clock_t ticks;        /* number of ticks for alarm */
    clock_t exptime;      /* needed for remaining time on previous alarm */
    clock_t uptime;       /* current system time */
    int remaining;        /* previous time left in seconds */
    int s;
    int proc_nr_n;

    if(pm_isokendpt(proc_nr_e, &proc_nr_n) != OK)
        return EINVAL;

    /* First determine remaining time of previous alarm, if set. */
    if (mproc[proc_nr_n].mp_flags & ALARM_ON) {
        if (s=getuptime(&uptime)) != OK)
            panic(__FILE__, "set_alarm couldn't get uptime", s);
        exptime = *tmr_exp_time(&mproc[proc_nr_n].mp_timer);
        remaining = (int) ((exptime - uptime + (HZ-1))/HZ);
        if (remaining < 0) remaining = 0;
    } else {
        remaining = 0;
    }

    /* Tell the clock task to provide a signal message when the time comes.
     *
     * Large delays cause a lot of problems. First, the alarm system call
     * takes an unsigned seconds count and the library has cast it to an int.
     * That probably works, but on return the library will convert "negative"
     * unsigneds to errors. Presumably no one checks for these errors, so
     * force this call through. Second, If unsigned and long have the same
     * size, converting from seconds to ticks can easily overflow. Finally,
     * the kernel has similar overflow bugs adding ticks.
     *
     * Fixing this requires a lot of ugly casts to fit the wrong interface
     * types and to avoid overflow traps. ALRM_EXP_TIME has the right type
     * (clock_t) although it is declared as long. How can variables like
     * this be declared properly without combinatorial explosion of message
     * types?
     */
    ticks = (clock_t) (HZ * (unsigned long) (unsigned) sec);
    if ( (unsigned long) ticks / HZ != (unsigned) sec)
        ticks = LONG_MAX; /* eternity (really TMR_NEVER) */

    if (ticks != 0) {
        pm_set_timer(&mproc[proc_nr_n].mp_timer, ticks,
            cause_sigalrm, proc_nr_e);
        mproc[proc_nr_n].mp_flags |= ALARM_ON;
    } else if (mproc[proc_nr_n].mp_flags & ALARM_ON) {
        pm_cancel_timer(&mproc[proc_nr_n].mp_timer);
        mproc[proc_nr_n].mp_flags &= ~ALARM_ON;
    }
    return(remaining);
}

/*=====
 *                               cause_sigalrm                               *
 *=====*/
PRIVATE void cause_sigalrm(tp)
struct timer *tp;
{
    int proc_nr_e, proc_nr_n;
    register struct mproc *rmp;

    /* get process from timer */
    if(pm_isokendpt(tmr_arg(tp)->ta_int, &proc_nr_n) != OK) {
        printf("PM: ignoring timer for invalid endpoint %d\n",
            tmr_arg(tp)->ta_int);
        return;
    }
}

```

```

    rmp = &mproc[proc_nr_n];

    if ((rmp->mp_flags & (IN_USE | ZOMBIE)) != IN_USE) return;
    if ((rmp->mp_flags & ALARM_ON) == 0) return;
    rmp->mp_flags &= ~ALARM_ON;
    check_sig(rmp->mp_pid, SIGALRM);
}

/*=====
 *                               do_pause                               *
 *=====*/
PUBLIC int do_pause()
{
    /* Perform the pause() system call. */

    mp->mp_flags |= PAUSED;
    return(SUSPEND);
}

/*=====
 *                               sig_proc                               *
 *=====*/
PUBLIC void sig_proc(rmp, signo)
register struct mproc *rmp;    /* pointer to the process to be signaled */
int signo;                    /* signal to send to process (1 to _NSIG) */
{
    /* Send a signal to a process. Check to see if the signal is to be caught,
     * ignored, transformed into a message (for system processes) or blocked.
     * - If the signal is to be transformed into a message, request the KERNEL to
     * send the target process a system notification with the pending signal as an
     * argument.
     * - If the signal is to be caught, request the KERNEL to push a sigcontext
     * structure and a sigframe structure onto the catcher's stack. Also, KERNEL
     * will reset the program counter and stack pointer, so that when the process
     * next runs, it will be executing the signal handler. When the signal handler
     * returns, sigreturn(2) will be called. Then KERNEL will restore the signal
     * context from the sigcontext structure.
     * If there is insufficient stack space, kill the process.
     */

    vir_bytes new_sp;
    int s;
    int slot;
    int sigflags;

    slot = (int) (rmp - mproc);
    if ((rmp->mp_flags & (IN_USE | ZOMBIE)) != IN_USE) {
        printf("PM: signal %d sent to %s process %d\n",
            signo, (rmp->mp_flags & ZOMBIE) ? "zombie" : "dead", slot);
        panic(__FILE__, "", NO_NUM);
    }
    if (rmp->mp_fs_call != PM_IDLE || rmp->mp_fs_call2 != PM_IDLE)
    {
        sigaddset(&rmp->mp_sigpending, signo);
        rmp->mp_flags |= PM_SIG_PENDING;
        /* keep the process from running */
        sys_nice(rmp->mp_endpoint, PRIO_STOP);
        return;
    }
    if ((rmp->mp_flags & TRACED) && signo != SIGKILL) {
        /* A traced process has special handling. */
        unpause(slot, TRUE /*for_trace*/);
        stop_proc(rmp, signo); /* a signal causes it to stop */
        return;
    }
    /* Some signals are ignored by default. */
    if (sigismember(&rmp->mp_ignore, signo)) {
        return;
    }
    if (sigismember(&rmp->mp_sigmask, signo)) {
        /* Signal should be blocked. */
        sigaddset(&rmp->mp_sigpending, signo);

```

```

        return;
    }
    #if ENABLE_SWAP
    if (rmp->mp_flags & ONSWAP) {
        /* Process is swapped out, leave signal pending. */
        sigaddset(&rmp->mp_sigpending, signo);
        swap_inqueue(rmp);
        return;
    }
    #endif
    sigflags = rmp->mp_sigact[signo].sa_flags;
    if (sigismember(&rmp->mp_catch, signo)) {
        if (rmp->mp_flags & SIGSUSPENDED)
            rmp->mp_sigmsg.sm_mask = rmp->mp_sigmask2;
        else
            rmp->mp_sigmsg.sm_mask = rmp->mp_sigmask;
        rmp->mp_sigmsg.sm_signo = signo;
        rmp->mp_sigmsg.sm_sighandler =
            (vir_bytes) rmp->mp_sigact[signo].sa_handler;
        rmp->mp_sigmsg.sm_sigreturn = rmp->mp_sigreturn;
        if ((s=get_stack_ptr(rmp->mp_endpoint, &new_sp)) != OK)
            panic(__FILE__, "couldn't get new stack pointer (for sig)", s);
        rmp->mp_sigmsg.sm_stkptr = new_sp;

        /* Make room for the sigcontext and sigframe struct. */
        new_sp -= sizeof(struct sigcontext)
            + 3 * sizeof(char *) + 2 * sizeof(int);

        if (adjust(rmp, rmp->mp_seg[D].mem_len, new_sp) != OK)
            goto doterminate;

        rmp->mp_sigmask |= rmp->mp_sigact[signo].sa_mask;
        if (sigflags & SA_NODEFER)
            sigdelset(&rmp->mp_sigmask, signo);
        else
            sigaddset(&rmp->mp_sigmask, signo);

        if (sigflags & SA_RESETHAND) {
            sigdelset(&rmp->mp_catch, signo);
            rmp->mp_sigact[signo].sa_handler = SIG_DFL;
        }
        sigdelset(&rmp->mp_sigpending, signo);

        /* Check to see if process is hanging on a PAUSE, WAIT or SIGSUSPEND
         * call.
         */
        if (rmp->mp_flags & (PAUSED | WAITING | SIGSUSPENDED)) {
            rmp->mp_flags &= ~(PAUSED | WAITING | SIGSUSPENDED);
            setreply(slot, EINTR);

            /* Ask the kernel to deliver the signal */
            s = sys_sigsend(rmp->mp_endpoint, &rmp->mp_sigmsg);
            if (s != OK)
                panic(__FILE__, "sys_sigsend failed", s);

            /* Done */
            return;
        }

        /* Ask FS to unpause the process. Deliver the signal when FS is
         * ready.
         */
        unpause(slot, FALSE /*!for_trace*/);
        return;
    }
    else if (sigismember(&rmp->mp_sig2mess, signo)) {
        /* Mark event pending in process slot and send notification. */
        sigaddset(&rmp->mp_sigpending, signo);
        notify(rmp->mp_endpoint);
        return;
    }
}

doterminate:

```

```

/* Signal should not or cannot be caught. Take default action. */
if (sigismember(&ign_sset, signo)) return;

rmp->mp_sigstatus = (char) signo;
if (sigismember(&core_sset, signo) && slot != FS_PROC_NR) {
#ifdef ENABLE_SWAP
    if (rmp->mp_flags & ONSWAP) {
        /* Process is swapped out, leave signal pending. */
        sigaddset(&rmp->mp_sigpending, signo);
        swap_inqueue(rmp);
        return;
    }
#endif

    s= dump_core(rmp);
    if (s == SUSPEND)
        return;

    /* Not dumping core, just call exit */
}
pm_exit(rmp, 0, FALSE /*!for_trace*/); /* terminate process */
}

/*=====
*                                     check_sig                                     *
*=====*/
PUBLIC int check_sig(proc_id, signo)
pid_t proc_id; /* pid of proc to sig, or 0 or -1, or -pgrp */
int signo; /* signal to send to process (0 to _NSIG) */
{
/* Check to see if it is possible to send a signal. The signal may have to be
* sent to a group of processes. This routine is invoked by the KILL system
* call, and also when the kernel catches a DEL or other signal.
*/

register struct mproc *rmp;
int count; /* count # of signals sent */
int error_code;

if (signo < 0 || signo > _NSIG) return(EINVAL);

/* Return EINVAL for attempts to send SIGKILL to INIT alone. */
if (proc_id == INIT_PID && signo == SIGKILL) return(EINVAL);

/* Search the proc table for processes to signal.
* (See forkexit.c about pid magic.)
*/
count = 0;
error_code = ESRCH;
for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
    if (!(rmp->mp_flags & IN_USE)) continue;
    if ((rmp->mp_flags & ZOMBIE) && signo != 0) continue;

    /* Check for selection. */
    if (proc_id > 0 && proc_id != rmp->mp_pid) continue;
    if (proc_id == 0 && mp->mp_procgrp != rmp->mp_procgrp) continue;
    if (proc_id == -1 && rmp->mp_pid <= INIT_PID) continue;
    if (proc_id < -1 && rmp->mp_procgrp != -proc_id) continue;

    /* Do not kill servers and drivers when broadcasting SIGKILL. */
    if (proc_id == -1 && signo == SIGKILL &&
        (rmp->mp_flags & PRIV_PROC)) continue;

    /* Check for permission. */
    if (mp->mp_effuid != SUPER_USER
        && mp->mp_realuid != rmp->mp_realuid
        && mp->mp_effuid != rmp->mp_realuid
        && mp->mp_realuid != rmp->mp_effuid
        && mp->mp_effuid != rmp->mp_effuid) {
        error_code = EPERM;
        continue;
    }

    count++;
}

```

```

    if (signo == 0) continue;

    /* 'sig_proc' will handle the disposition of the signal. The
     * signal may be caught, blocked, ignored, or cause process
     * termination, possibly with core dump.
     */
    sig_proc(rmp, signo);

    if (proc_id > 0) break; /* only one process being signaled */
}

/* If the calling process has killed itself, don't reply. */
if ((mp->mp_flags & (IN_USE | ZOMBIE)) != IN_USE) return(SUSPEND);
return(count > 0 ? OK : error_code);
}

/*=====
 *                               check_pending                               *
 *=====*/
PUBLIC void check_pending(rmp)
register struct mproc *rmp;
{
    /* Check to see if any pending signals have been unblocked. The
     * first such signal found is delivered.
     *
     * If multiple pending unmasked signals are found, they will be
     * delivered sequentially.
     *
     * There are several places in this file where the signal mask is
     * changed. At each such place, check_pending() should be called to
     * check for newly unblocked signals.
     */

    int i;

    for (i = 1; i <= _NSIG; i++) {
        if (sigismember(&rmp->mp_sigpending, i) &&
            !sigismember(&rmp->mp_sigmask, i)) {
            sigdelset(&rmp->mp_sigpending, i);
            sig_proc(rmp, i);
            break;
        }
    }
}

/*=====
 *                               unpause                               *
 *=====*/
PRIVATE void unpause(pro, for_trace)
int pro; /* which process number */
int for_trace; /* for tracing */
{
    /* A signal is to be sent to a process. If that process is hanging on a
     * system call, the system call must be terminated with EINTR. Possible
     * calls are PAUSE, WAIT, READ and WRITE, the latter two for pipes and ttys.
     * First check if the process is hanging on an PM call. If not, tell FS,
     * so it can check for READs and WRITES from pipes, ttys and the like.
     */
    register struct mproc *rmp;
    int r;

    rmp = &mproc[pro];

    /* Check to see if process is hanging on a PAUSE, WAIT or SIGSUSPEND call. */
    if (rmp->mp_flags & (PAUSED | WAITING | SIGSUSPENDED)) {
        rmp->mp_flags &= ~(PAUSED | WAITING | SIGSUSPENDED);
        setreply(pro, EINTR);
        return;
    }

    /* Process is not hanging on an PM call. Ask FS to take a look. */
    if (for_trace)
    {
        if (rmp->mp_fs_call != PM_IDLE)

```

```

        panic( __FILE__, "unpause: not idle", rmp->mp_fs_call);
        rmp->mp_fs_call= PM_UNPAUSE_TR;
    }
    else
    {
        if (rmp->mp_fs_call2 != PM_IDLE)
            panic( __FILE__, "unpause: not idle", rmp->mp_fs_call2);
        rmp->mp_fs_call2= PM_UNPAUSE;
    }
    r= notify(FS_PROC_NR);
    if (r != OK) panic("pm", "unpause: unable to notify FS", r);
}

/*=====
*                                     dump_core                                     *
*=====*/
PRIVATE int dump_core(rmp)
register struct mproc *rmp;      /* whose core is to be dumped */
{
    /* Make a core dump on the file "core", if possible. */

    int r, proc_nr, proc_nr_e, parent_waiting;
    pid_t procgrp;
    vir_bytes current_sp;
    struct mproc *p_mp;
    clock_t t[5];

    /* Do not create core files for set uid execution */
    if (rmp->mp_realuid != rmp->mp_effuid) return OK;

    /* Make sure the stack segment is up to date.
     * We don't want adjust() to fail unless current_sp is preposterous,
     * but it might fail due to safety checking. Also, we don't really want
     * the adjust() for sending a signal to fail due to safety checking.
     * Maybe make SAFETY_BYTES a parameter.
     */
    if ((r= get_stack_ptr(rmp->mp_endpoint, &current_sp)) != OK)
        panic(__FILE__, "couldn't get new stack pointer (for core)", r);
    adjust(rmp, rmp->mp_seg[D].mem_len, current_sp);

    /* Tell FS about the exiting process. */
    if (rmp->mp_fs_call != PM_IDLE)
        panic(__FILE__, "dump_core: not idle", rmp->mp_fs_call);
    rmp->mp_fs_call= PM_DUMP_CORE;
    r= notify(FS_PROC_NR);
    if (r != OK) panic(__FILE__, "dump_core: unable to notify FS", r);

    /* Also perform most of the normal exit processing. Informing the parent
     * has to wait until we know whether the coredump was successful or not.
     */

    proc_nr = (int) (rmp - mproc);      /* get process slot number */
    proc_nr_e = rmp->mp_endpoint;

    /* Remember a session leader's process group. */
    procgrp = (rmp->mp_pid == mp->mp_procgrp) ? mp->mp_procgrp : 0;

    /* If the exited process has a timer pending, kill it. */
    if (rmp->mp_flags & ALARM_ON) set_alarm(proc_nr_e, (unsigned) 0);

    /* Do accounting: fetch usage times and accumulate at parent. */
    if ((r=sys_times(proc_nr_e, t)) != OK)
        panic(__FILE__, "pm_exit: sys_times failed", r);

    p_mp = &mproc[rmp->mp_parent];      /* process' parent */
    p_mp->mp_child_utime += t[0] + rmp->mp_child_utime; /* add user time */
    p_mp->mp_child_stime += t[1] + rmp->mp_child_stime; /* add system time */

    /* Tell the kernel the process is no longer runnable to prevent it from
     * being scheduled in between the following steps. Then tell FS that it
     * the process has exited and finally, clean up the process at the kernel.
     * This order is important so that FS can tell drivers to cancel requests
     * such as copying to/ from the exiting process, before it is gone.
     */
}

```



```
sys_nice(proc_nr_e, PRIO_STOP);          /* stop the process */

if(proc_nr_e != FS_PROC_NR)              /* if it is not FS that is exiting.. */
{
    if (rmp->mp_flags & PRIV_PROC)
    {
        /* destroy system processes without waiting for FS */
        if((r= sys_exit(rmp->mp_endpoint)) != OK)
            panic(__FILE__, "pm_exit: sys_exit failed", r);

        /* Just send a SIGCHLD. Dealing with waieldpid is too complicated
         * here.
         */
        p_mp = &mproc[rmp->mp_parent];    /* process' parent */
        sig_proc(p_mp, SIGCHLD);

        /* Zombify to avoid calling sys_endksig */
        rmp->mp_flags |= ZOMBIE;
    }
}
else
{
    printf("PM: FS died\n");
    return;
}

/* Pending reply messages for the dead process cannot be delivered. */
rmp->mp_flags &= ~REPLY;

/* Keep the process around until FS is finished with it. */

/* If the process has children, disinherit them. INIT is the new parent. */
for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
    if (rmp->mp_flags & IN_USE && rmp->mp_parent == proc_nr) {
        /* 'rmp' now points to a child to be disinherited. */
        rmp->mp_parent = INIT_PROC_NR;
        parent_waiting = mproc[INIT_PROC_NR].mp_flags & WAITING;
        if (parent_waiting && (rmp->mp_flags & ZOMBIE))
        {
            tell_parent(rmp);
            real_cleanup(rmp);
        }
    }
}

/* Send a hangup to the process' process group if it was a session leader. */
if (procgrp != 0) check_sig(-procgrp, SIGHUP);

return SUSPEND;
}
```

```

/* This file contains the table used to map system call numbers onto the
 * routines that perform them.
 */

#define _TABLE

#include "pm.h"
#include <minix/callnr.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

/* Miscellaneous */
char core_name[] = "core";          /* file name where core images are produced */

_PROTOTYPE (int (*call_vec[NCALLS]), (void) ) = {
    no_sys,          /* 0 = unused */
    do_pm_exit,      /* 1 = exit */
    do_fork,          /* 2 = fork */
    no_sys,          /* 3 = read */
    no_sys,          /* 4 = write */
    no_sys,          /* 5 = open */
    no_sys,          /* 6 = close */
    do_waitpid,      /* 7 = wait */
    no_sys,          /* 8 = creat */
    no_sys,          /* 9 = link */
    no_sys,          /* 10 = unlink */
    do_waitpid,      /* 11 = waitpid */
    no_sys,          /* 12 = chdir */
    do_time,         /* 13 = time */
    no_sys,          /* 14 = mknod */
    no_sys,          /* 15 = chmod */
    no_sys,          /* 16 = chown */
    do_brk,          /* 17 = break */
    no_sys,          /* 18 = stat */
    no_sys,          /* 19 = lseek */
    do_getset,       /* 20 = getpid */
    no_sys,          /* 21 = mount */
    no_sys,          /* 22 = umount */
    do_getset,       /* 23 = setuid */
    do_getset,       /* 24 = getuid */
    do_stime,        /* 25 = stime */
    do_trace,        /* 26 = ptrace */
    do_alarm,        /* 27 = alarm */
    no_sys,          /* 28 = fstat */
    do_pause,        /* 29 = pause */
    no_sys,          /* 30 = utime */
    no_sys,          /* 31 = (stty) */
    no_sys,          /* 32 = (gtty) */
    no_sys,          /* 33 = access */
    no_sys,          /* 34 = (nice) */
    no_sys,          /* 35 = (ftime) */
    no_sys,          /* 36 = sync */
    do_kill,         /* 37 = kill */
    no_sys,          /* 38 = rename */
    no_sys,          /* 39 = mkdir */
    no_sys,          /* 40 = rmdir */
    no_sys,          /* 41 = dup */
    no_sys,          /* 42 = pipe */
    do_times,        /* 43 = times */
    no_sys,          /* 44 = (prof) */
    no_sys,          /* 45 = unused */
    do_getset,       /* 46 = setgid */
    do_getset,       /* 47 = getgid */
    no_sys,          /* 48 = (signal) */
    no_sys,          /* 49 = unused */
    no_sys,          /* 50 = unused */
    no_sys,          /* 51 = (acct) */
    no_sys,          /* 52 = (phys) */
    no_sys,          /* 53 = (lock) */
    no_sys,          /* 54 = ioctl */
    no_sys,          /* 55 = fcntl */
    no_sys,          /* 56 = (mpx) */
    no_sys,          /* 57 = unused */

```

```
no_sys,      /* 58 = unused */
do_exec,     /* 59 = execve */
no_sys,      /* 60 = umask */
no_sys,      /* 61 = chroot */
do_getset,   /* 62 = setsid */
do_getset,   /* 63 = getpgrp */

no_sys,      /* 64 = unused */
no_sys,      /* 65 = unused */
no_sys,      /* 66 = unused */
no_sys,      /* 67 = unused */
no_sys,      /* 68 = unused */
no_sys,      /* 69 = unused */
no_sys,      /* 70 = unused */
do_sigaction, /* 71 = sigaction */
do_sigsuspend, /* 72 = sigsuspend */
do_sigpending, /* 73 = sigpending */
do_sigprocmask, /* 74 = sigprocmask */
do_sigreturn, /* 75 = sigreturn */
do_reboot,   /* 76 = reboot */
do_svrctl,   /* 77 = svrctl */
do_sysuname, /* 78 = sysuname */
do_getsysinfo, /* 79 = getsysinfo */
no_sys,      /* 80 = unused */
no_sys,      /* 81 = unused */
no_sys,      /* 82 = (fstatfs) */
no_sys,      /* 83 = unused */
no_sys,      /* 84 = unused */
no_sys,      /* 85 = (select) */
no_sys,      /* 86 = (fchdir) */
no_sys,      /* 87 = (fsync) */
do_getsetpriority, /* 88 = getpriority */
do_getsetpriority, /* 89 = setpriority */
do_time,     /* 90 = gettimeofday */
do_getset,   /* 91 = seteuid */
do_getset,   /* 92 = setegid */
no_sys,      /* 93 = (truncate) */
no_sys,      /* 94 = (ftruncate) */
no_sys,      /* 95 = (fchmod) */
no_sys,      /* 96 = (fchown) */
};
/* This should not fail with "array size is negative": */
extern int dummy[sizeof(call_vec) == NCALLS * sizeof(call_vec[0]) ? 1 : -1];
```

```

/* This file takes care of those system calls that deal with time.
 *
 * The entry points into this file are
 *   do_time:      perform the TIME system call
 *   do_stime:     perform the STIME system call
 *   do_times:     perform the TIMES system call
 */

#include "pm.h"
#include <minix/callnr.h>
#include <minix/com.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

/*=====
 *                               do_time                               *
 *=====*/
PUBLIC int do_time()
{
/* Perform the time(tp) system call. This returns the time in seconds since
 * 1.1.1970. MINIX is an astrophysically naive system that assumes the earth
 * rotates at a constant rate and that such things as leap seconds do not
 * exist.
 */
    clock_t uptime;
    int s;

    if ( (s=getuptime(&uptime)) != OK)
        panic(__FILE__, "do_time couldn't get uptime", s);

    mp->mp_reply.reply_time = (time_t) (boottime + (uptime/HZ));
    mp->mp_reply.reply_utime = (uptime%HZ)*1000000/HZ;
    return(OK);
}

/*=====
 *                               do_stime                               *
 *=====*/
PUBLIC int do_stime()
{
/* Perform the stime(tp) system call. Retrieve the system's uptime (ticks
 * since boot) and store the time in seconds at system boot in the global
 * variable 'boottime'.
 */
    clock_t uptime;
    int s;

    if (mp->mp_effuid != SUPER_USER) {
        return(EPERM);
    }
    if ( (s=getuptime(&uptime)) != OK)
        panic(__FILE__, "do_stime couldn't get uptime", s);
    boottime = (long) m_in.stime - (uptime/HZ);

    if (mp->mp_fs_call != PM_IDLE)
        panic("pm", "do_stime: not idle", mp->mp_fs_call);
    mp->mp_fs_call= PM_STIME;
    s= notify(FS_PROC_NR);
    if (s != OK) panic("pm", "do_stime: unable to notify FS", s);

    /* Do not reply until FS is ready to process the stime request */
    return(SUSPEND);
}

/*=====
 *                               do_times                               *
 *=====*/
PUBLIC int do_times()
{
/* Perform the times(buffer) system call. */
    register struct mproc *rmp = mp;
    clock_t t[5];
    int s;

```

```
if (OK != (s=sys_times(who_e, t)))
    panic(__FILE__, "do_times couldn't get times", s);
rmp->mp_reply.reply_t1 = t[0];          /* user time */
rmp->mp_reply.reply_t2 = t[1];          /* system time */
rmp->mp_reply.reply_t3 = rmp->mp_child_otime; /* child user time */
rmp->mp_reply.reply_t4 = rmp->mp_child_stime; /* child system time */
rmp->mp_reply.reply_t5 = t[4];          /* uptime since boot */

return(OK);
}
```

```

/* PM watchdog timer management. These functions in this file provide
 * a convenient interface to the timers library that manages a list of
 * watchdog timers. All details of scheduling an alarm at the CLOCK task
 * are hidden behind this interface.
 * Only system processes are allowed to set an alarm timer at the kernel.
 * Therefore, the PM maintains a local list of timers for user processes
 * that requested an alarm signal.
 *
 * The entry points into this file are:
 *   pm_set_timer:      reset and existing or set a new watchdog timer
 *   pm_expire_timers: check for expired timers and run watchdog functions
 *   pm_cancel_timer:  remove a time from the list of timers
 */

#include "pm.h"

#include <timers.h>
#include <minix/syslib.h>
#include <minix/com.h>

PRIVATE timer_t *pm_timers = NULL;

/*=====
 *                               pm_set_timer                               *
 *=====*/
PUBLIC void pm_set_timer(timer_t *tp, int ticks, tmr_func_t watchdog, int arg)
{
    int r;
    clock_t now, prev_time = 0, next_time;

    if ((r = getuptime(&now)) != OK)
        panic(__FILE__, "PM couldn't get uptime", NO_NUM);

    /* Set timer argument and add timer to the list. */
    tmr_arg(tp)->ta_int = arg;
    prev_time = tmrs_settimer(&pm_timers, tp, now+ticks, watchdog, &next_time);

    /* Reschedule our synchronous alarm if necessary. */
    if (!prev_time || prev_time > next_time) {
        if (sys_setalarm(next_time, 1) != OK)
            panic(__FILE__, "PM set timer couldn't set alarm.", NO_NUM);
    }

    return;
}

/*=====
 *                               pm_expire_timers                           *
 *=====*/
PUBLIC void pm_expire_timers(clock_t now)
{
    clock_t next_time;

    /* Check for expired timers and possibly reschedule an alarm. */
    tmrs_exptimers(&pm_timers, now, &next_time);
    if (next_time > 0) {
        if (sys_setalarm(next_time, 1) != OK)
            panic(__FILE__, "PM expire timer couldn't set alarm.", NO_NUM);
    }
}

/*=====
 *                               pm_cancel_timer                           *
 *=====*/
PUBLIC void pm_cancel_timer(timer_t *tp)
{
    clock_t next_time, prev_time;
    prev_time = tmrs_clrtimer(&pm_timers, tp, &next_time);

    /* If the earliest timer has been removed, we have to set the alarm to
     * the next timer, or cancel the alarm altogether if the last timer has
     * been cancelled (next_time will be 0 then).
     */
}

```

```
    if (prev_time < next_time || ! next_time) {  
        if (sys_setalarm(next_time, 1) != OK)  
            panic(__FILE__, "PM expire timer couldn't set alarm.", NO_NUM);  
    }  
}
```

```

/* This file handles the process manager's part of debugging, using the
 * ptrace system call. Most of the commands are passed on to the system
 * task for completion.
 *
 * The debugging commands available are:
 * T_STOP      stop the process
 * T_OK        enable tracing by parent for this process
 * T_GETINS    return value from instruction space
 * T_GETDATA   return value from data space
 * T_GETUSER   return value from user process table
 * T_SETINS    set value in instruction space
 * T_SETDATA   set value in data space
 * T_SETUSER   set value in user process table
 * T_RESUME    resume execution
 * T_EXIT      exit
 * T_STEP      set trace bit
 *
 * The T_OK and T_EXIT commands are handled here, and the T_RESUME and
 * T_STEP commands are partially handled here and completed by the system
 * task. The rest are handled entirely by the system task.
 */

#include "pm.h"
#include <minix/com.h>
#include <sys/ptrace.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

#define NIL_MPROC      ((struct mproc *) 0)

FORWARD _PROTOTYPE( struct mproc *find_proc, (pid_t lpid) );

/*=====
 *                      do_trace
 *=====*/
PUBLIC int do_trace()
{
    register struct mproc *child;
    int r;

    /* the T_OK call is made by the child fork of the debugger before it execs
     * the process to be traced
     */
    if (m_in.request == T_OK) { /* enable tracing by parent for this proc */
        mp->mp_flags |= TRACED;
        mp->mp_reply.reply_trace = 0;
        return(OK);
    }
    if ((child=find_proc(m_in.pid))==NIL_MPROC || !(child->mp_flags & STOPPED)) {
        return(ESRCH);
    }
    /* all the other calls are made by the parent fork of the debugger to
     * control execution of the child
     */
    switch (m_in.request) {
    case T_EXIT: /* exit */
        pm_exit(child, (int) m_in.data, TRUE /*for_trace*/);
        /* Do not reply to the caller until FS has processed the exit
         * request.
         */
        return SUSPEND;
    case T_RESUME:
    case T_STEP: /* resume execution */
        if (m_in.data < 0 || m_in.data > _NSIG) return(EIO);
        if (m_in.data > 0) { /* issue signal */
            child->mp_flags &= ~TRACED; /* so signal is not diverted */
            sig_proc(child, (int) m_in.data);
            child->mp_flags |= TRACED;
        }
        child->mp_flags &= ~STOPPED;
        break;
    }
    r = sys_trace(m_in.request, child->mp_endpoint, m_in.taddr, &m_in.data);
}

```



```

    if (r != OK) return(r);

    mp->mp_reply.reply_trace = m_in.data;
    return(OK);
}

/*=====
 *                               find_proc                               *
 *=====*/
PRIVATE struct mproc *find_proc(lpid)
pid_t lpid;
{
    register struct mproc *rmp;

    for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++)
        if (rmp->mp_flags & IN_USE && rmp->mp_pid == lpid) return(rmp);
    return(NIL_MPROC);
}

/*=====
 *                               stop_proc                               *
 *=====*/
PUBLIC void stop_proc(rmp, signo)
register struct mproc *rmp;
int signo;
{
    /* A traced process got a signal so stop it. */

    register struct mproc *rpmp = mproc + rmp->mp_parent;
    int r;

    r = sys_trace(-1, rmp->mp_endpoint, 0L, (long *) 0);
    if (r != OK) panic("pm", "sys_trace failed", r);

    rmp->mp_flags |= STOPPED;
    if (rpmp->mp_flags & WAITING) {
        rpmp->mp_flags &= ~WAITING;      /* parent is no longer waiting */
        rpmp->mp_reply.reply_res2 = 0177 | (signo << 8);
        setreply(rmp->mp_parent, rmp->mp_pid);
    } else {
        rmp->mp_sigstatus = signo;
    }
    return;
}

```

```
/* If there were any type definitions local to the Process Manager, they would  
* be here. This file is included only for symmetry with the kernel and File  
* System, which do have some local type definitions.  
*/
```

```

/* This file contains some utility routines for PM.
 *
 * The entry points are:
 *   find_param:      look up a boot monitor parameter
 *   get_free_pid:    get a free process or group id
 *   allowed:         see if an access is permitted
 *   no_sys:          called for invalid system call numbers
 *   panic:           PM has run aground of a fatal error
 *   get_mem_map:     get memory map of given process
 *   get_stack_ptr:   get stack pointer of given process
 *   proc_from_pid:   return process pointer from pid number
 */

#include "pm.h"
#include <sys/stat.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <minix/endpoint.h>
#include <fcntl.h>
#include <signal.h>          /* needed only because mproc.h needs it */
#include "mproc.h"
#include "param.h"

#include <minix/config.h>
#include <timers.h>
#include <string.h>
#include "../kernel/const.h"
#include "../kernel/config.h"
#include "../kernel/type.h"
#include "../kernel/proc.h"

/*=====
 *                               get_free_pid                               *
 *=====*/
PUBLIC pid_t get_free_pid()
{
    static pid_t next_pid = INIT_PID + 1;      /* next pid to be assigned */
    register struct mproc *rmp;               /* check process table */
    int t;                                     /* zero if pid still free */

    /* Find a free pid for the child and put it in the table. */
    do {
        t = 0;
        next_pid = (next_pid < NR_PIDS ? next_pid + 1 : INIT_PID + 1);
        for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++)
            if (rmp->mp_pid == next_pid || rmp->mp_procgrp == next_pid) {
                t = 1;
                break;
            }
    } while (t);                               /* 't' = 0 means pid free */
    return(next_pid);
}

/*=====
 *                               no_sys                               *
 *=====*/
PUBLIC int no_sys()
{
    /* A system call number not implemented by PM has been requested. */
    printf("PM: in no_sys, call nr %d from %d\n", call_nr, who_e);
    return(ENOSYS);
}

/*=====
 *                               panic                               *
 *=====*/
PUBLIC void panic(who, mess, num)
char *who;          /* who caused the panic */
char *mess;         /* panic message string */
int num;            /* number to go with it */
{
    /* An unrecoverable error has occurred. Panics are caused when an internal
     * inconsistency is detected, e.g., a programming error or illegal value of a

```

```

* defined constant. The process manager decides to exit.
*/
message m;
int s;

/* Switch to primary console and print panic message. */
printf("PM panic(%s): %s", who, mess);
if (num != NO_NUM) printf(":%d", num);
printf("\n");

/* Exit PM. */
sys_exit(SELF);
}

/*=====
*                               find_param                               *
*=====*/
PUBLIC char *find_param(name)
const char *name;
{
    register const char *namep;
    register char *envp;

    for (envp = (char *) monitor_params; *envp != 0;) {
        for (namep = name; *namep != 0 && *namep == *envp; namep++, envp++)
            ;
        if (*namep == '\0' && *envp == '=')
            return(envp + 1);
        while (*envp++ != 0)
            ;
    }
    return(NULL);
}

/*=====
*                               get_mem_map                             *
*=====*/
PUBLIC int get_mem_map(proc_nr, mem_map)
int proc_nr;
struct mem_map *mem_map;
/* process to get map of */
/* put memory map here */
{
    struct proc p;
    int s;

    if ((s=sys_getproc(&p, proc_nr)) != OK)
        return(s);
    memcpy(mem_map, p.p_memmap, sizeof(p.p_memmap));
    return(OK);
}

/*=====
*                               get_stack_ptr                             *
*=====*/
PUBLIC int get_stack_ptr(proc_nr_e, sp)
int proc_nr_e;
vir_bytes *sp;
/* process to get sp of */
/* put stack pointer here */
{
    struct proc p;
    int s;

    if ((s=sys_getproc(&p, proc_nr_e)) != OK)
        return(s);
    *sp = p.p_reg.sp;
    return(OK);
}

/*=====
*                               proc_from_pid                             *
*=====*/
PUBLIC int proc_from_pid(mp_pid)
pid_t mp_pid;
{
    int rmp;

```

```
    for (rmp = 0; rmp < NR_PROCS; rmp++)
        if (mproc[rmp].mp_pid == mp_pid)
            return rmp;

    return -1;
}

/*=====
 *                               pm_isokendpt                               *
 *=====*/
PUBLIC int pm_isokendpt(int endpoint, int *proc)
{
    *proc = _ENDPOINT_P(endpoint);
    if(*proc < -NR_TASKS || *proc >= NR_PROCS)
        return EINVAL;
    if(*proc >= 0 && endpoint != mproc[*proc].mp_endpoint)
        return EDEADSRCDST;
    if(*proc >= 0 && !(mproc[*proc].mp_flags & IN_USE))
        return EDEADSRCDST;
    return OK;
}
```

```
# Makefile for Reincarnation Server (RS)
SERVER = rs
UTIL = service

# directories
u = /usr
i = $u/include
s = $i/sys
m = $i/minix
b = $i/ibm

# programs, flags, etc.
CC =      exec cc
CFLAGS = -I$i
LDFLAGS = -i
UTIL_LIBS = -lsys
LIBS = -lsys -lsysutil

UTIL_OBJ = service.o
OBJ = exec.o main.o manager.o

# build local binary
all build:      $(SERVER) $(UTIL)
$(UTIL):      $(UTIL_OBJ)
              $(CC) -o $@ $(LDFLAGS) $(UTIL_OBJ) $(UTIL_LIBS)
$(SERVER):    $(OBJ)
              $(CC) -o $@ $(LDFLAGS) $(OBJ) $(LIBS)
              install -S 48k $@

# install with other servers
install:      /bin/$(UTIL) /usr/sbin/$(SERVER)
/bin/$(UTIL):   $(UTIL)
              install -c $? $@
/usr/sbin/$(SERVER): $(SERVER)
              install -o root -c $? $@

# clean up local files
clean:
              rm -f $(UTIL) $(SERVER) *.o *.bak

depend:
              /usr/bin/mkdep "$(CC) -E $(CPPFLAGS)" *.c > .depend

# Include generated dependencies.
include .depend
```

```
/* Header file for the system service manager server.
 *
 * Created:
 *   Jul 22, 2005           by Jorrit N. Herder
 */

#define _SYSTEM            1    /* get OK and negative error codes */
#define _MINIX             1    /* tell headers to include MINIX stuff */

#define VERBOSE            0    /* display diagnostics */

#include <ansi.h>
#include <sys/types.h>
#include <limits.h>
#include <errno.h>
#include <signal.h>
#include <unistd.h>

#include <minix/callnr.h>
#include <minix/config.h>
#include <minix/type.h>
#include <minix/const.h>
#include <minix/com.h>
#include <minix/syslib.h>
#include <minix/sysutil.h>
#include <minix/keymap.h>
#include <minix/bitmap.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>

#include "proto.h"
#include "manager.h"
```

```

/* Reincarnation Server. This servers starts new system services and detects
 * they are exiting. In case of errors, system services can be restarted.
 * The RS server periodically checks the status of all registered services
 * services to see whether they are still alive. The system services are
 * expected to periodically send a heartbeat message.
 *
 * Created:
 *   Jul 22, 2005          by Jorrit N. Herder
 */
#include "inc.h"
#include <minix/dmap.h>
#include <minix/endpoint.h>
#include "../kernel/const.h"
#include "../kernel/type.h"

/* Declare some local functions. */
FORWARD _PROTOTYPE(void init_server, (void)                );
FORWARD _PROTOTYPE(void sig_handler, (void)                );
FORWARD _PROTOTYPE(void get_work, (message *m)             );
FORWARD _PROTOTYPE(void reply, (int whom, int result)      );

/* Data buffers to retrieve info during initialization. */
PRIVATE struct boot_image image[NR_BOOT_PROCS];
PUBLIC struct dmap dmap[NR_DEVICES];

/*=====
 *                               main                               *
 *=====*/
PUBLIC int main(void)
{
/* This is the main routine of this service. The main loop consists of
 * three major activities: getting new work, processing the work, and
 * sending the reply. The loop never terminates, unless a panic occurs.
 */
    message m;                /* request message */
    int call_nr, who_e, who_p; /* call number and caller */
    int result;               /* result to return */
    sigset_t sigset;          /* system signal set */
    int s;

    /* Initialize the server, then go to work. */
    init_server();

    /* Main loop - get work and do it, forever. */
    while (TRUE) {

        /* Wait for request message. */
        get_work(&m);
        who_e = m.m_source;
        who_p = _ENDPOINT_P(who_e);
        if(who_p < -NR_TASKS || who_p >= NR_PROCS)
            panic("RS", "message from bogus source", who_e);

        call_nr = m.m_type;

        /* Now determine what to do. Three types of requests are expected:
         * - Heartbeat messages (notifications from registered system services)
         * - System notifications (POSIX signals or synchronous alarm)
         * - User requests (control messages to manage system services)
         */

        /* Notification messages are control messages and do not need a reply.
         * These include heartbeat messages and system notifications.
         */
        if (m.m_type & NOTIFY_MESSAGE) {
            switch (call_nr) {
                case SYN_ALARM:
                    do_period(&m);                /* check drivers status */
                    continue;
                case PROC_EVENT:
                    sig_handler();
                    continue;
                default:
                    /* heartbeat notification */
                    if (rproc_ptr[who_p] != NULL) /* mark heartbeat time */

```



```

        rproc_ptr[who_p]->r_alive_tm = m.NOTIFY_TIMESTAMP;
    }
}

/* If this is not a notification message, it is a normal request.
 * Handle the request and send a reply to the caller.
 */
else {
    switch(call_nr) {
        case RS_UP:                result = do_up(&m, FALSE /*!do_copy*/); break;
        case RS_UP_COPY:           result = do_up(&m, TRUE /*do_copy*/);  break;
        case RS_DOWN:              result = do_down(&m);                  break;
        case RS_REFRESH:           result = do_refresh(&m);               break;
        case RS_RESCUE:            result = do_rescue(&m);                break;
        case RS_SHUTDOWN:          result = do_shutdown(&m);              break;
        case GETSYSINFO:           result = do_getsysinfo(&m);            break;
        default:
            printf("Warning, RS got unexpected request %d from %d\n",
                m.m_type, m.m_source);
            result = EINVAL;
    }

    /* Finally send reply message, unless disabled. */
    if (result != EDONTREPLY) {
        reply(who_e, result);
    }
}
}
}

/*=====
 *                               init_server                               *
 *=====*/
PRIVATE void init_server(void)
{
    /* Initialize the reincarnation server. */
    struct sigaction sa;
    struct boot_image *ip;
    int s,t;

    /* Install signal handlers. Ask PM to transform signal into message. */
    sa.sa_handler = SIG_MESS;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGCHLD,&sa,NULL)<0) panic("RS","sigaction failed", errno);
    if (sigaction(SIGTERM,&sa,NULL)<0) panic("RS","sigaction failed", errno);

    /* Initialize the system process table. Use the boot image from the kernel
     * and the device map from the FS to gather all needed information.
     */
    if ((s = sys_getimage(image)) != OK)
        panic("RS","warning: couldn't get copy of image table", s);
    if ((s = getsysinfo(FS_PROC_NR, SI_DMAP_TAB, dmap)) < 0)
        panic("RS","warning: couldn't get copy of dmap table", errno);

    #if 0
    /* Now initialize the table with the processes in the system image.
     * Prepend /sbin/ to the binaries so that we can actually find them.
     */
    for (s=0; s< NR_BOOT_PROCS; s++) {
        ip = &image[s];
        if (ip->proc_nr >= 0) {
            nr_in_use ++;
            rproc[s].r_flags = RS_IN_USE;
            rproc[s].r_proc_nr_e = ip->endpoint;
            rproc[s].r_pid = getnpid(ip->proc_nr);
            for(t=0; t< NR_DEVICES; t++)
                if (dmap[t].dmap_driver == ip->proc_nr)
                    rproc[s].r_dev_nr = t;
            strcpy(rproc[s].r_cmd, "/sbin/");
            strcpy(rproc[s].r_cmd+6, ip->proc_name);
            rproc[s].r_argc = 1;
            rproc[s].r_argv[0] = rproc[s].r_cmd;
        }
    }
}

```

```

        rproc[s].r_argv[1] = NULL;
    }
}
#endif

/* Set alarm to periodically check driver status. */
if (OK != (s=sys_setalarm(RS_DELTA_T, 0)))
    panic("RS", "couldn't set alarm", s);
}

/*=====
 *                               sig_handler                               *
 *=====*/
PRIVATE void sig_handler()
{
    sigset_t sigset;
    int sig;

    /* Try to obtain signal set from PM. */
    if (getsigset(&sigset) != 0) return;

    /* Check for known signals. */
    if (sigismember(&sigset, SIGCHLD)) do_exit(NULL);
    if (sigismember(&sigset, SIGTERM)) do_shutdown(NULL);
}

/*=====
 *                               get_work                               *
 *=====*/
PRIVATE void get_work(m_in)
message *m_in;                                /* pointer to message */
{
    int s;                                    /* receive status */
    if (OK != (s=receive(ANY, m_in)))        /* wait for message */
        panic("RS", "receive failed", s);
}

/*=====
 *                               reply                               *
 *=====*/
PRIVATE void reply(who, result)
int who;                                     /* replyee */
int result;                                /* report result */
{
    message m_out;                          /* reply message */
    int s;                                  /* send status */

    m_out.m_type = result;                  /* build reply message */
    if (OK != (s=send(who, &m_out)))        /* send the message */
        panic("RS", "unable to send reply", s);
}

```

```

/*
 * Changes:
 *   Jul 22, 2005:      Created (Jorrit N. Herder)
 */

#include "inc.h"
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <minix/dmap.h>
#include <minix/endpoint.h>
#include <lib.h>

/* Allocate variables. */
struct rproc rproc[NR_SYS_PROCS];          /* system process table */
struct rproc *rproc_ptr[NR_PROCS];        /* mapping for fast access */
int nr_in_use;                             /* number of services */
extern int errno;                          /* error status */

/* Prototypes for internal functions that do the hard work. */
FORWARD _PROTOTYPE( int start_service, (struct rproc *rp) );
FORWARD _PROTOTYPE( int stop_service, (struct rproc *rp,int how) );
FORWARD _PROTOTYPE( int fork_nb, (void) );
FORWARD _PROTOTYPE( int read_exec, (struct rproc *rp) );

PRIVATE int shutting_down = FALSE;

#define EXEC_FAILED      49                /* recognizable status */

/*=====
 *                               do_up
 *=====*/
PUBLIC int do_up(m_ptr, do_copy)
message *m_ptr;                          /* request message pointer */
int do_copy;                             /* keep copy in memory */
{
/* A request was made to start a new system service. Dismember the request
 * message and gather all information needed to start the service. Starting
 * is done by a helper routine.
 */
    register struct rproc *rp;           /* system process table */
    int slot_nr;                         /* local table entry */
    int arg_count;                       /* number of arguments */
    char *cmd_ptr;                       /* parse command string */
    enum dev_style dev_style;            /* device style */
    int s;                               /* status variable */

    /* See if there is a free entry in the table with system processes. */
    if (nr_in_use >= NR_SYS_PROCS) return(EAGAIN);
    for (slot_nr = 0; slot_nr < NR_SYS_PROCS; slot_nr++) {
        rp = &rproc[slot_nr];           /* get pointer to slot */
        if (! rp->r_flags & RS_IN_USE)    /* check if available */
            break;
    }
    nr_in_use++;                          /* update administration */

    /* Obtain command name and parameters. This is a space-separated string
     * that looks like "/sbin/service arg1 arg2 ...". Arguments are optional.
     */
    if (m_ptr->RS_CMD_LEN > MAX_COMMAND_LEN) return(E2BIG);
    if (OK!=(s=sys_datacopy(m_ptr->m_source, (vir_bytes) m_ptr->RS_CMD_ADDR,
        SELF, (vir_bytes) rp->r_cmd, m_ptr->RS_CMD_LEN))) return(s);
    rp->r_cmd[m_ptr->RS_CMD_LEN] = '\0';    /* ensure it is terminated */
    if (rp->r_cmd[0] != '/') return(EINVAL); /* insist on absolute path */

    /* Build argument vector to be passed to execute call. The format of the
     * arguments vector is: path, arguments, NULL.
     */
    arg_count = 0;                        /* initialize arg count */
    rp->r_argv[arg_count++] = rp->r_cmd;    /* start with path */
    cmd_ptr = rp->r_cmd;                   /* do some parsing */
    while(*cmd_ptr != '\0') {              /* stop at end of string */

```

```

    if (*cmd_ptr == ' ') {                /* next argument */
        *cmd_ptr = '\0';                  /* terminate previous */
        while (++cmd_ptr == ' ');          /* skip spaces */
        if (*cmd_ptr == '\0') break;       /* no arg following */
        if (arg_count > MAX_NR_ARGS+1) break; /* arg vector full */
        rp->r_argv[arg_count++] = cmd_ptr; /* add to arg vector */
    }
    cmd_ptr++;                             /* continue parsing */
}
rp->r_argv[arg_count] = NULL;              /* end with NULL pointer */
rp->r_argc = arg_count;

rp->r_exec = NULL;
if (do_copy)
{
    s = read_exec(rp);
    if (s != OK)
        return s;
}

/* Initialize some fields. */
rp->r_period = m_ptr->RS_PERIOD;
rp->r_dev_nr = m_ptr->RS_DEV_MAJOR;
rp->r_dev_style = STYLE_DEV;
rp->r_restarts = -1;                      /* will be incremented */

/* All information was gathered. Now try to start the system service. */
return(start_service(rp));
}

/*=====
 *                               do_down                               *
 *=====*/
PUBLIC int do_down(message *m_ptr)
{
    register struct rproc *rp;
    pid_t pid = (pid_t) m_ptr->RS_PID;

    for (rp=BEG_RPROC_ADDR; rp<END_RPROC_ADDR; rp++) {
        if (rp->r_flags & RS_IN_USE && rp->r_pid == pid) {
#if VERBOSE
            printf("stopping %d(%d)\n", pid, m_ptr->RS_PID);
#endif
            stop_service(rp, RS_EXITING);
            return(OK);
        }
    }
#if VERBOSE
    printf("not found %d(%d)\n", pid, m_ptr->RS_PID);
#endif
    return(ESRCH);
}

/*=====
 *                               do_refresh                             *
 *=====*/
PUBLIC int do_refresh(message *m_ptr)
{
    register struct rproc *rp;
    pid_t pid = (pid_t) m_ptr->RS_PID;

    for (rp=BEG_RPROC_ADDR; rp<END_RPROC_ADDR; rp++) {
        if (rp->r_flags & RS_IN_USE && rp->r_pid == pid) {
#if VERBOSE
            printf("refreshing %d(%d)\n", pid, m_ptr->RS_PID);
#endif
            stop_service(rp, RS_REFRESHING);
            return(OK);
        }
    }
#if VERBOSE
    printf("not found %d(%d)\n", pid, m_ptr->RS_PID);

```

```

#endif
    return(ESRCH);
}

/*=====
 *                               do_rescue                               *
 *=====*/
PUBLIC int do_rescue(message *m_ptr)
{
    char rescue_dir[MAX_RESCUE_DIR_LEN];
    int s;

    /* Copy rescue directory from user. */
    if (m_ptr->RS_CMD_LEN > MAX_RESCUE_DIR_LEN) return(E2BIG);
    if (OK!=(s=sys_datacopy(m_ptr->m_source, (vir_bytes) m_ptr->RS_CMD_ADDR,
        SELF, (vir_bytes) rescue_dir, m_ptr->RS_CMD_LEN))) return(s);
    rescue_dir[m_ptr->RS_CMD_LEN] = '\0'; /* ensure it is terminated */
    if (rescue_dir[0] != '/') return(EINVAL); /* insist on absolute path */

    /* Change RS' directory to the rescue directory. Provided that the needed
     * binaries are in the rescue dir, this makes recovery possible even if the
     * (root) file system is no longer available, because no directory lookups
     * are required. Thus if an absolute path fails, we can try to strip the
     * path and see if the command is in the rescue dir.
     */
    if (chdir(rescue_dir) != 0) return(errno);
    return(OK);
}

/*=====
 *                               do_shutdown                               *
 *=====*/
PUBLIC int do_shutdown(message *m_ptr)
{
    /* Set flag so that RS server knows services shouldn't be restarted. */
    shutting_down = TRUE;
    return(OK);
}

/*=====
 *                               do_exit                               *
 *=====*/
PUBLIC void do_exit(message *m_ptr)
{
    register struct rproc *rp;
    pid_t exit_pid;
    int exit_status;

    #if VERBOSE
        printf("RS: got SIGCHLD signal, doing wait to get exited child.\n");
    #endif

    /* See which child exited and what the exit status is. This is done in a
     * loop because multiple childs may have exited, all reported by one
     * SIGCHLD signal. The WNOHANG options is used to prevent blocking if,
     * somehow, no exited child can be found.
     */
    while ( (exit_pid = waitpid(-1, &exit_status, WNOHANG)) != 0 ) {

    #if VERBOSE
        printf("RS: pid %d, ", exit_pid);
        if (WIFSIGNALED(exit_status)) {
            printf("killed, signal number %d\n", WTERMSIG(exit_status));
        }
        else if (WIFEXITED(exit_status)) {
            printf("normal exit, status %d\n", WEXITSTATUS(exit_status));
        }
    #endif
    }

    /* Search the system process table to see who exited.
     * This should always succeed.
     */
    for (rp=BEG_RPROC_ADDR; rp<END_RPROC_ADDR; rp++) {
        if ((rp->r_flags & RS_IN_USE) && rp->r_pid == exit_pid) {

```

```

    int proc;
    proc = _ENDPOINT_P(rp->r_proc_nr_e);

    rproc_ptr[proc] = NULL;          /* invalidate */

    if ((rp->r_flags & RS_EXITING) || shutting_down) {
        rp->r_flags = 0;              /* release slot */
        if (rp->r_exec)
        {
            free(rp->r_exec);
            rp->r_exec = NULL;
        }
        rproc_ptr[proc] = NULL;
    }
    else if (rp->r_flags & RS_REFRESHING) {
        rp->r_restarts = -1;          /* reset counter */
        start_service(rp);           /* direct restart */
    }
    else if (WIFEXITED(exit_status) &&
              WEXITSTATUS(exit_status) == EXEC_FAILED) {
        rp->r_flags = 0;              /* release slot */
    }
    else {
#if VERBOSE
        printf("Unexpected exit. Restarting %s\n", rp->r_cmd);
#endif

        /* Determine what to do. If this is the first unexpected
         * exit, immediately restart this service. Otherwise use
         * a binary exponential backoff.
         */

#if 0
        rp->r_restarts = 0;
#endif
        if (rp->r_restarts > 0) {
            rp->r_backoff = 1 << MIN(rp->r_restarts, (BACKOFF_BITS-2));
            rp->r_backoff = MIN(rp->r_backoff, MAX_BACKOFF);
            if (rp->r_exec != NULL && rp->r_backoff > 1)
                rp->r_backoff = 1;
        }
        else {
            start_service(rp);        /* direct restart */
        }
    }
    break;
}
}
}

/*=====
 *                               do_period                               *
 *=====*/
PUBLIC void do_period(m_ptr)
message *m_ptr;
{
    register struct rproc *rp;
    clock_t now = m_ptr->NOTIFY_TIMESTAMP;
    int s;

    /* Search system services table. Only check slots that are in use. */
    for (rp=BEG_RPROC_ADDR; rp<END_RPROC_ADDR; rp++) {
        if (rp->r_flags & RS_IN_USE) {

            /* If the service is to be revived (because it repeatedly exited,
             * and was not directly restarted), the binary backoff field is
             * greater than zero.
             */
            if (rp->r_backoff > 0) {
                rp->r_backoff -= 1;
                if (rp->r_backoff == 0) {
                    start_service(rp);
                }
            }
        }
    }
}

```

```

/* If the service was signaled with a SIGTERM and fails to respond,
 * kill the system service with a SIGKILL signal.
 */
else if (rp->r_stop_tm > 0 && now - rp->r_stop_tm > 2*RS_DELTA_T
&& rp->r_pid > 0) {
    kill(rp->r_pid, SIGKILL);          /* terminate */
}

/* There seems to be no special conditions. If the service has a
 * period assigned check its status.
 */
else if (rp->r_period > 0) {

    /* Check if an answer to a status request is still pending. If
     * the driver didn't respond within time, kill it to simulate
     * a crash. The failure will be detected and the service will
     * be restarted automatically.
     */
    if (rp->r_alive_tm < rp->r_check_tm) {
        if (now - rp->r_alive_tm > 2*rp->r_period &&
            rp->r_pid > 0) {
#if VERBOSE
            printf("RS: service %d reported late\n", rp->r_proc_nr_e);
#endif
            kill(rp->r_pid, SIGKILL);          /* simulate crash */
        }
    }

    /* No answer pending. Check if a period expired since the last
     * check and, if so request the system service's status.
     */
    else if (now - rp->r_check_tm > rp->r_period) {
#if VERBOSE
        printf("RS: status request sent to %d\n", rp->r_proc_nr_e);
#endif
        notify(rp->r_proc_nr_e);              /* request status */
        rp->r_check_tm = now;                  /* mark time */
    }
}

/* Reschedule a synchronous alarm for the next period. */
if (OK != (s=sys_setalarm(RS_DELTA_T, 0)))
    panic("RS", "couldn't set alarm", s);
}

/*=====
 *                               start_service                               *
 *=====*/
PRIVATE int start_service(rp)
struct rproc *rp;
{
/* Try to execute the given system service. Fork a new process. The child
 * process will be inhibited from running by the NO_PRIV flag. Only let the
 * child run once its privileges have been set by the parent.
 */
    int child_proc_nr_e, child_proc_nr_n;          /* child process slot */
    pid_t child_pid;                                /* child's process id */
    char *file_only;
    int s, use_copy;
    message m;

    use_copy= (rp->r_exec != NULL);

    /* Now fork and branch for parent and child process (and check for error). */
    if (use_copy)
        child_pid= fork_nb();
    else
        child_pid = fork();

    switch(child_pid) {
        /* see fork(2) */
        case -1:
            /* fork failed */

```

```

report("RS", "warning, fork() failed", errno);          /* shouldn't happen */
return(errno);                                          /* return error */

case 0:                                                  /* child process */
/* Try to execute the binary that has an absolute path. If this fails,
 * e.g., because the root file system cannot be read, try to strip of
 * the path, and see if the command is in RS' current working dir.
 */
if (!use_copy)
{
    execve(rp->r_argv[0], rp->r_argv, NULL);              /* POSIX execute */
    file_only = strchr(rp->r_argv[0], '/') + 1;
    execve(file_only, rp->r_argv, NULL);                 /* POSIX execute */
}
printf("RS: exec failed for %s: %d\n", rp->r_argv[0], errno);
exit(EXEC_FAILED);                                     /* terminate child */

default:                                                /* parent process */
    child_proc_nr_e = getnprocnr(child_pid);            /* get child slot */
    break;                                              /* continue below */
}

if (use_copy)
{
    extern char **environ;
    dev_execve(child_proc_nr_e, rp->r_exec, rp->r_exec_len, rp->r_argv,
                environ);
}

/* Set the privilege structure for the child process to let it run.
 * This should succeed: we tested number in use above.
 */
if ((s = sys_privctl(child_proc_nr_e, SYS_PRIV_INIT, 0, NULL)) < 0) {
    report("RS", "sys_privctl call failed", s);          /* to let child run */
    rp->r_flags |= RS_EXITING;                            /* expect exit */
    if (child_pid > 0) kill(child_pid, SIGKILL);          /* kill driver */
    else report("RS", "didn't kill pid", child_pid);
    return(s);                                           /* return error */
}

if (rp->r_dev_nr > 0) {                                  /* set driver map */
    if ((s = mapdriver(child_proc_nr_e, rp->r_dev_nr, rp->r_dev_style,
        !!use_copy /* force */)) < 0) {
        report("RS", "couldn't map driver", errno);
        rp->r_flags |= RS_EXITING;                        /* expect exit */
        if (child_pid > 0) kill(child_pid, SIGKILL);      /* kill driver */
        else report("RS", "didn't kill pid", child_pid);
        return(s);                                       /* return error */
    }
}

#ifdef VERBOSE
    printf("RS: started '%s', major %d, pid %d, endpoint %d, proc %d\n",
        rp->r_cmd, rp->r_dev_nr, child_pid,
        child_proc_nr_e, child_proc_nr_n);
#endif

/* The system service now has been successfully started. Update the rest
 * of the system process table that is maintain by the RS server. The only
 * thing that can go wrong now, is that execution fails at the child. If
 * that's the case, the child will exit.
 */
child_proc_nr_n = _ENDPOINT_P(child_proc_nr_e);
rp->r_flags = RS_IN_USE;                                /* mark slot in use */
rp->r_restarts += 1;                                    /* raise nr of restarts */
rp->r_proc_nr_e = child_proc_nr_e;                     /* set child details */
rp->r_pid = child_pid;
rp->r_check_tm = 0;                                     /* not check yet */
getuptime(&rp->r_alive_tm);                             /* currently alive */
rp->r_stop_tm = 0;                                       /* not exiting yet */
rproc_ptr[child_proc_nr_n] = rp;                      /* mapping for fast access */
return(OK);
}

```



```

/*=====
*
*                               stop_service
*=====*/
PRIVATE int stop_service(rp,how)
struct rproc *rp;
int how;
{
    /* Try to stop the system service. First send a SIGTERM signal to ask the
     * system service to terminate. If the service didn't install a signal
     * handler, it will be killed. If it did and ignores the signal, we'll
     * find out because we record the time here and send a SIGKILL.
     */
    #if VERBOSE
        printf("RS tries to stop %s (pid %d)\n", rp->r_cmd, rp->r_pid);
    #endif

    rp->r_flags |= how;                               /* what to on exit? */
    if(rp->r_pid > 0) kill(rp->r_pid, SIGTERM);        /* first try friendly */
    else report("RS", "didn't kill pid", rp->r_pid);
    getuptime(&rp->r_stop_tm);                        /* record current time */
}

/*=====
*
*                               do_getsysinfo
*=====*/
PUBLIC int do_getsysinfo(m_ptr)
message *m_ptr;
{
    vir_bytes src_addr, dst_addr;
    int dst_proc;
    size_t len;
    int s;

    switch(m_ptr->m1_i1) {
    case SI_PROC_TAB:
        src_addr = (vir_bytes) rproc;
        len = sizeof(struct rproc) * NR_SYS_PROCS;
        break;
    default:
        return(EINVAL);
    }

    dst_proc = m_ptr->m_source;
    dst_addr = (vir_bytes) m_ptr->m1_p1;
    if (OK != (s=sys_datacopy(SELF, src_addr, dst_proc, dst_addr, len)))
        return(s);
    return(OK);
}

PRIVATE pid_t fork_nb()
{
    message m;

    return(_syscall(PM_PROC_NR, FORK_NB, &m));
}

PRIVATE int read_exec(rp)
struct rproc *rp;
{
    int e, r, fd;
    char *e_name;
    struct stat sb;

    e_name= rp->r_argv[0];
    r= stat(e_name, &sb);
    if (r != 0)
        return -errno;

    fd= open(e_name, O_RDONLY);
    if (fd == -1)
        return -errno;

    rp->r_exec_len= sb.st_size;
}

```

```
rp->r_exec= malloc(rp->r_exec_len);
if (rp->r_exec == NULL)
{
    printf("read_exec: unable to allocate %d bytes\n",
           rp->r_exec_len);
    close(fd);
    return ENOMEM;
}

r= read(fd, rp->r_exec, rp->r_exec_len);
e= errno;
close(fd);
if (r == rp->r_exec_len)
    return OK;

printf("read_exec: read failed %d, errno %d\n", r, e);

free(rp->r_exec);
rp->r_exec= NULL;

if (r >= 0)
    return EIO;
else
    return -e;
}
```

```

/* This table has one slot per system process. It contains information for
 * servers and driver needed by the reincarnation server to keep track of
 * each process' status.
 */

/* Space reserved for program and arguments. */
#define MAX_COMMAND_LEN      512      /* maximum argument string length */
#define MAX_NR_ARGS          4        /* maximum number of arguments */
#define MAX_RESCUE_DIR_LEN   64       /* maximum rescue dir length */

/* Definition of the system process table. This table only has entries for
 * the servers and drivers, and thus is not directly indexed by slot number.
 */
extern struct rproc {
    int r_proc_nr_e;          /* process endpoint number */
    pid_t r_pid;              /* process id */
    dev_t r_dev_nr;          /* major device number */
    int r_dev_style;          /* device style */

    int r_restarts;           /* number of restarts (initially zero) */
    long r_backoff;           /* number of periods to wait before revive */
    unsigned r_flags;         /* status and policy flags */

    long r_period;            /* heartbeat period (or zero) */
    clock_t r_check_tm;       /* timestamp of last check */
    clock_t r_alive_tm;       /* timestamp of last heartbeat */
    clock_t r_stop_tm;        /* timestamp of SIGTERM signal */

    char *r_exec;             /* Executable image */
    size_t r_exec_len;        /* Length of image */

    char r_cmd[MAX_COMMAND_LEN]; /* raw command plus arguments */
    char *r_argv[MAX_NR_ARGS+2]; /* parsed arguments vector */
    int r_argc;               /* number of arguments */
} rproc[NR_SYS_PROCS];

/* Mapping for fast access to the system process table. */
extern struct rproc *rproc_ptr[NR_PROCS];
extern int nr_in_use;

/* Flag values. */
#define RS_IN_USE      0x001 /* set when process slot is in use */
#define RS_EXITING     0x002 /* set when exit is expected */
#define RS_REFRESHING  0x004 /* set when refresh must be done */

/* Constants determining RS period and binary exponential backoff. */
#define RS_DELTA_T      60 /* check every T ticks */
#define BACKOFF_BITS    (sizeof(long)*8) /* bits in backoff field */
#define MAX_BACKOFF     30 /* max backoff in RS_DELTA_T */

/* Magic process table addresses. */
#define BEG_RPROC_ADDR (&rproc[0])
#define END_RPROC_ADDR (&rproc[NR_SYS_PROCS])
#define NIL_RPROC ((struct mproc *) 0)

```

```
/* Function prototypes. */

/* exec.c */
_PROTOTYPE( int dev_execve, (int proc_e,
                             char *exec, size_t exec_len, char *argv[], char **env));

/* main.c */
_PROTOTYPE( int main, (void));

/* manager.c */
_PROTOTYPE( int do_up, (message *m, int do_copy));
_PROTOTYPE( int do_down, (message *m));
_PROTOTYPE( int do_refresh, (message *m));
_PROTOTYPE( int do_rescue, (message *m));
_PROTOTYPE( int do_shutdown, (message *m));
_PROTOTYPE( void do_period, (message *m));
_PROTOTYPE( void do_exit, (message *m));
_PROTOTYPE( int do_getsysinfo, (message *m));
```

```

/* Utility to start or stop system services. Requests are sent to the
 * reincarnation server that does the actual work.
 *
 * Changes:
 *   Jul 22, 2005:      Created (Jorrit N. Herder)
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <minix/config.h>
#include <minix/com.h>
#include <minix/const.h>
#include <minix/type.h>
#include <minix/ipc.h>
#include <minix/syslib.h>
#include <sys/types.h>
#include <sys/stat.h>

/* This array defines all known requests. */
PRIVATE char *known_requests[] = {
    "up",
    "down",
    "refresh",
    "rescue",
    "shutdown",
    "catch for illegal requests"
};
#define ILLEGAL_REQUEST    sizeof(known_requests)/sizeof(char *)

/* Global error number set for failed system calls. */
#define OK 0
extern int errno;

/* Define names for arguments provided to this utility. The first few
 * arguments are required and have a known index. Thereafter, some optional
 * argument pairs like "-args arglist" follow.
 */
#define ARG_NAME            0          /* own application name */

/* The following are relative to optind */
#define ARG_REQUEST         0          /* request to perform */
#define ARG_PATH            1          /* rescue dir or system service */
#define ARG_PID             1          /* pid of system service */

#define MIN_ARG_COUNT       1          /* require an action */

#define ARG_ARGS            "-args"    /* list of arguments to be passed */
#define ARG_DEV             "-dev"     /* major device number for drivers */
#define ARG_PRIV            "-priv"    /* required privileges */
#define ARG_PERIOD          "-period"  /* heartbeat period in ticks */

/* The function parse_arguments() verifies and parses the command line
 * parameters passed to this utility. Request parameters that are needed
 * are stored globally in the following variables:
 */
PRIVATE int req_type;
PRIVATE int req_pid;
PRIVATE char *req_path;
PRIVATE char *req_args;
PRIVATE int req_major;
PRIVATE long req_period;
PRIVATE char *req_priv;

/* Buffer to build "/command arg1 arg2 ..." string to pass to RS server. */
PRIVATE char command[4096];

/* An error occurred. Report the problem, print the usage, and exit.
 */
PRIVATE void print_usage(char *app_name, char *problem)
{

```

```

printf("Warning,%s\n", problem);
printf("Usage:\n");
printf("  %s [-c] up <binary> [%s <args>] [%s <special>] [%s <ticks>]\n",
    app_name, ARG_ARGS, ARG_DEV, ARG_PERIOD);
printf("  %s down <pid>\n", app_name);
printf("  %s refresh <pid>\n", app_name);
printf("  %s rescue <dir>\n", app_name);
printf("  %s shutdown\n", app_name);
printf("\n");
}

/* A request to the RS server failed. Report and exit.
 */
PRIVATE void failure(int num)
{
    printf("Request to RS failed: %s (error %d)\n", strerror(num), num);
    exit(num);
}

/* Parse and verify correctness of arguments. Report problem and exit if an
 * error is found. Store needed parameters in global variables.
 */
PRIVATE int parse_arguments(int argc, char **argv)
{
    struct stat stat_buf;
    char *hz;
    int req_nr;
    int c, i;
    int c_flag;

    c_flag= 0;
    while (c= getopt(argc, argv, "c?"), c != -1)
    {
        switch(c)
        {
            case '?':
                print_usage(argv[ARG_NAME], "wrong number of arguments");
                exit(EINVAL);
            case 'c':
                c_flag= 1;
                break;
            default:
                fprintf(stderr, "%s: getopt failed: %c\n",
                    argv[ARG_NAME], c);
                exit(1);
        }
    }

    /* Verify argument count. */
    if (argc < optind+MIN_ARG_COUNT) {
        print_usage(argv[ARG_NAME], "wrong number of arguments");
        exit(EINVAL);
    }

    /* Verify request type. */
    for (req_type=0; req_type< ILLEGAL_REQUEST; req_type++) {
        if (strcmp(known_requests[req_type],argv[optind+ARG_REQUEST])==0) break;
    }
    if (req_type == ILLEGAL_REQUEST) {
        print_usage(argv[ARG_NAME], "illegal request type");
        exit(ENOSYS);
    }
    req_nr = RS_RQ_BASE + req_type;

    if (req_nr == RS_UP) {

        if (c_flag)
            req_nr= RS_UP_COPY;

        /* Verify argument count. */
        if (argc - 1 < optind+ARG_PATH) {
            print_usage(argv[ARG_NAME], "action requires a binary to start");
            exit(EINVAL);
        }
    }
}

```

```

}

/* Verify the name of the binary of the system service. */
req_path = argv[optind+ARG_PATH];
if (req_path[0] != '/') {
    print_usage(argv[ARG_NAME], "binary should be absolute path");
    exit(EINVAL);
}
if (stat(req_path, &stat_buf) == -1) {
    perror(req_path);
    fprintf(stderr, "couldn't get stat binary\n");
    exit(errno);
}
if (!(stat_buf.st_mode & S_IFREG)) {
    print_usage(argv[ARG_NAME], "binary is not a regular file");
    exit(EINVAL);
}

/* Check optional arguments that come in pairs like "-args arglist". */
for (i=optind+MIN_ARG_COUNT+1; i<argc; i=i+2) {
    if (!(i+1 < argc)) {
        print_usage(argv[ARG_NAME], "optional argument not complete");
        exit(EINVAL);
    }
    if (strcmp(argv[i], ARG_ARGS)==0) {
        req_args = argv[i+1];
    }
    else if (strcmp(argv[i], ARG_PERIOD)==0) {
        req_period = strtol(argv[i+1], &hz, 10);
        if (strcmp(hz, "HZ")==0) req_period *= HZ;
        if (req_period < 1) {
            print_usage(argv[ARG_NAME],
                "period is at least be one tick");
            exit(EINVAL);
        }
    }
    else if (strcmp(argv[i], ARG_DEV)==0) {
        if (stat(argv[i+1], &stat_buf) == -1) {
            print_usage(argv[ARG_NAME], "couldn't get status of device");
            exit(errno);
        }
        if (!(stat_buf.st_mode & (S_IFBLK | S_IFCHR))) {
            print_usage(argv[ARG_NAME], "special file is not a device");
            exit(EINVAL);
        }
        req_major = (stat_buf.st_rdev >> MAJOR) & BYTE;
    }
    else if (strcmp(argv[i], ARG_ARGS)==0) {
        req_priv = argv[i+1];
    }
    else {
        print_usage(argv[ARG_NAME], "unknown optional argument given");
        exit(EINVAL);
    }
}
}
else if (req_nr == RS_DOWN || req_nr == RS_REFRESH) {
    /* Verify argument count. */
    if (argc - 1 < optind+ARG_PID) {
        print_usage(argv[ARG_NAME], "action requires a pid to stop");
        exit(EINVAL);
    }
    if (!(req_pid = atoi(argv[optind+ARG_PID])) > 0) {
        print_usage(argv[ARG_NAME], "pid must be greater than zero");
        exit(EINVAL);
    }
}
else if (req_nr == RS_RESCUE) {
    /* Verify argument count. */
    if (argc - 1 < optind+ARG_PATH) {
        print_usage(argv[ARG_NAME], "action requires rescue directory");
        exit(EINVAL);
    }
}

```

```

    }
    req_path = argv[optind+ARG_PATH];
    if (req_path[0] != '/') {
        print_usage(argv[ARG_NAME], "rescue dir should be absolute path");
        exit(EINVAL);
    }
    if (stat(argv[optind+ARG_PATH], &stat_buf) == -1) {
        print_usage(argv[ARG_NAME], "couldn't get status of directory");
        exit(errno);
    }
    if ( ! (stat_buf.st_mode & S_IFDIR)) {
        print_usage(argv[ARG_NAME], "file is not a directory");
        exit(EINVAL);
    }
}
else if (req_nr == RS_SHUTDOWN) {
    /* no extra arguments required */
}

/* Return the request number if no error were found. */
return(req_nr);
}

/* Main program.
 */
PUBLIC int main(int argc, char **argv)
{
    message m;
    int result;
    int request;
    int s;

    /* Verify and parse the command line arguments. All arguments are checked
     * here. If an error occurs, the problem is reported and exit(2) is called.
     * all needed parameters to perform the request are extracted and stored
     * global variables.
     */
    request = parse_arguments(argc, argv);

    /* Arguments seem fine. Try to perform the request. Only valid requests
     * should end up here. The default is used for not yet supported requests.
     */
    switch(request) {
    case RS_UP:
    case RS_UP_COPY:
        /* Build space-separated command string to be passed to RS server. */
        strcpy(command, req_path);
        command[strlen(req_path)] = ' ';
        strcpy(command+strlen(req_path)+1, req_args);

        /* Build request message and send the request. */
        m.RS_CMD_ADDR = command;
        m.RS_CMD_LEN = strlen(command);
        m.RS_DEV_MAJOR = req_major;
        m.RS_PERIOD = req_period;
        if (OK != (s=_taskcall(RS_PROC_NR, request, &m)))
            failure(-s);
        result = m.m_type;
        break;
    case RS_DOWN:
    case RS_REFRESH:
        m.RS_PID = req_pid;
        if (OK != (s=_taskcall(RS_PROC_NR, request, &m)))
            failure(-s);
        break;
    case RS_RESCUE:
        m.RS_CMD_ADDR = req_path;
        m.RS_CMD_LEN = strlen(req_path);
        if (OK != (s=_taskcall(RS_PROC_NR, request, &m)))
            failure(-s);
        break;
    case RS_SHUTDOWN:
        if (OK != (s=_taskcall(RS_PROC_NR, request, &m)))

```



```
        failure(-s);
    break;
default:
    print_usage(argv[ARG_NAME], "request is not yet supported" );
    result = EGENERIC;
}
return(result);
}
```

```

#include "inc.h"
#include <a.out.h>

#define BLOCK_SIZE      1024

static void do_exec(int proc_e, char *exec, size_t exec_len, char *progname,
    char *frame, int frame_len);
FORWARD _PROTOTYPE( int read_header, (char *exec, size_t exec_len, int *sep_id,
    vir_bytes *text_bytes, vir_bytes *data_bytes,
    vir_bytes *bss_bytes, phys_bytes *tot_bytes, vir_bytes *pc,
    int *hdrlenp) );
FORWARD _PROTOTYPE( int exec_newmem, (int proc_e, vir_bytes text_bytes,
    vir_bytes data_bytes, vir_bytes bss_bytes, vir_bytes tot_bytes,
    vir_bytes frame_len, int sep_id,
    Dev_t st_dev, ino_t st_ino, time_t st_ctime, char *progname,
    int new_uid, int new_gid,
    vir_bytes *stack_top, int *load_textp, int *allow_setuidp) );
FORWARD _PROTOTYPE( int exec_restart, (int proc_e, int result) );
FORWARD _PROTOTYPE( void patch_ptr, (char stack[ARG_MAX],
    vir_bytes base) );
FORWARD _PROTOTYPE( int read_seg, (char *exec, size_t exec_len, off_t off,
    int proc_e, int seg, phys_bytes seg_bytes) );

static int self_e= NONE;

int dev_execve(int proc_e, char *exec, size_t exec_len, char **argv,
    char **Xenvp)
{
    char * const *ap;
    char * const *ep;
    char *frame;
    char **vp;
    char *sp, *progname;
    size_t argc;
    size_t frame_size;
    size_t string_off;
    size_t n;
    int ov;
    message m;

    /* Assumptions: size_t and char *, it's all the same thing. */

    /* Create a stack image that only needs to be patched up slightly
     * by the kernel to be used for the process to be executed.
     */

    ov= 0;
    frame_size= 0;
    string_off= 0;
    argc= 0;

    /* No overflow yet. */
    /* Size of the new initial stack. */
    /* Offset to start of the strings. */
    /* Argument count. */

    for (ap= argv; *ap != NULL; ap++) {
        n = sizeof(*ap) + strlen(*ap) + 1;
        frame_size+= n;
        if (frame_size < n) ov= 1;
        string_off+= sizeof(*ap);
        argc++;
    }

    if 0
    printf("here: %s, %d\n", __FILE__, __LINE__);
    for (ep= envp; *ep != NULL; ep++) {
        n = sizeof(*ep) + strlen(*ep) + 1;
        frame_size+= n;
        if (frame_size < n) ov= 1;
        string_off+= sizeof(*ap);
    }

    /* Add an argument count and two terminating nulls. */
    frame_size+= sizeof(argc) + sizeof(*ap) + sizeof(*ep);
    string_off+= sizeof(argc) + sizeof(*ap) + sizeof(*ep);

    /* Align. */

```

```

    frame_size= (frame_size + sizeof(char *) - 1) & ~(sizeof(char *) - 1);

    /* The party is off if there is an overflow. */
    if ((ov || frame_size < 3 * sizeof(char *)) {
        errno= E2BIG;
        return -1;
    }

    /* Allocate space for the stack frame. */
    if ((frame = (char *) sbrk(frame_size)) == (char *) -1) {
        errno = E2BIG;
        return -1;
    }

    /* Set arg count, init pointers to vector and string tables. */
    * (size_t *) frame = argc;
    vp = (char **) (frame + sizeof(argc));
    sp = frame + string_off;

    /* Load the argument vector and strings. */
    for (ap= argv; *ap != NULL; ap++) {
        *vp++= (char *) (sp - frame);
        n= strlen(*ap) + 1;
        memcpy(sp, *ap, n);
        sp+= n;
    }
    *vp++= NULL;

#if 0
    /* Load the environment vector and strings. */
    for (ep= envp; *ep != NULL; ep++) {
        *vp++= (char *) (sp - frame);
        n= strlen(*ep) + 1;
        memcpy(sp, *ep, n);
        sp+= n;
    }
#endif
    *vp++= NULL;

    /* Padding. */
    while (sp < frame + frame_size) *sp++= 0;

    (progrname=strrchr(argv[0], '/')) ? progrname++ : (progrname=argv[0]);
    do_exec(proc_e, exec, exec_len, progrname, frame, frame_size);

    /* Failure, return the memory used for the frame and exit. */
    (void) sbrk(-frame_size);
    return -1;
}

static void do_exec(int proc_e, char *exec, size_t exec_len, char *progrname,
    char *frame, int frame_len)
{
    int r;
    int hdrlen, sep_id, load_text, allow_setuid;
    int need_restart, error;
    vir_bytes stack_top, vsp;
    vir_bytes text_bytes, data_bytes, bss_bytes, pc;
    phys_bytes tot_bytes;
    off_t off;
    uid_t new_uid;
    gid_t new_gid;

    need_restart= 0;
    error= 0;

    self_e = getnprocnr(getpid());

    /* Read the file header and extract the segment sizes. */
    r = read_header(exec, exec_len, &sep_id,
        &text_bytes, &data_bytes, &bss_bytes,
        &tot_bytes, &pc, &hdrlen);
    if (r != OK)
    {

```

```

        printf("do_exec: read_header failed\n");
        goto fail;
    }
    need_restart= 1;

    new_uid= getuid();
    new_gid= getgid();
    /* XXX what should we use to identify the executable? */
    r= exec_newmem(proc_e, text_bytes, data_bytes, bss_bytes, tot_bytes,
        frame_len, sep_id, 0 /*dev*/, proc_e /*inum*/, 0 /*ctime*/,
        progname, new_uid, new_gid, &stack_top, &load_text,
        &allow_setuid);
    if (r != OK)
    {
        printf("do_exec: exec_newmap failed: %d\n", r);
        error= r;
        goto fail;
    }

    /* Patch up stack and copy it from FS to new core image. */
    vsp = stack_top;
    vsp -= frame_len;
    patch_ptr(frame, vsp);
    r = sys_datacopy(SELFS, (vir_bytes) frame,
        proc_e, (vir_bytes) vsp, (phys_bytes) frame_len);
    if (r != OK) panic(__FILE__, "pm_exec stack copy err on", proc_e);

    off = hdrlen;

    /* Read in text and data segments. */
    if (load_text) {
        r= read_seg(exec, exec_len, off, proc_e, T, text_bytes);
        if (r != OK)
        {
            printf("do_exec: read_seg failed: %d\n", r);
            error= r;
            goto fail;
        }
    }
    else
        printf("do_exec: not loading text segment\n");

    off += text_bytes;
    r= read_seg(exec, exec_len, off, proc_e, D, data_bytes);
    if (r != OK)
    {
        printf("do_exec: read_seg failed: %d\n", r);
        error= r;
        goto fail;
    }

    exec_restart(proc_e, OK);

    return;

fail:
    printf("do_exec(fail): error = %d\n", error);
    if (need_restart)
        exec_restart(proc_e, error);
}

/*=====
 *
 *                               exec_newmem
 *=====*/
PRIVATE int exec_newmem(proc_e, text_bytes, data_bytes, bss_bytes, tot_bytes,
    frame_len, sep_id, st_dev, st_ino, st_ctime, progname,
    new_uid, new_gid, stack_top, load_text, allow_setuid)
int proc_e;
vir_bytes text_bytes;
vir_bytes data_bytes;
vir_bytes bss_bytes;
vir_bytes tot_bytes;
vir_bytes frame_len;
int sep_id;

```

```

dev_t st_dev;
ino_t st_ino;
time_t st_ctime;
int new_uid;
int new_gid;
char *progrname;
vir_bytes *stack_top;
int *load_textp;
int *allow_setuidp;
{
    int r;
    struct exec_newmem e;
    message m;

    e.text_bytes= text_bytes;
    e.data_bytes= data_bytes;
    e.bss_bytes= bss_bytes;
    e.tot_bytes= tot_bytes;
    e.args_bytes= frame_len;
    e.sep_id= sep_id;
    e.st_dev= st_dev;
    e.st_ino= st_ino;
    e.st_ctime= st_ctime;
    e.new_uid= new_uid;
    e.new_gid= new_gid;
    strncpy(e.progrname, progrname, sizeof(e.progrname)-1);
    e.progrname[sizeof(e.progrname)-1]= '\0';

    m.m_type= EXEC_NEWMEM;
    m.EXC_NM_PROC= proc_e;
    m.EXC_NM_PTR= (char *)&e;
    r= sendrec(PM_PROC_NR, &m);
    if (r != OK)
        return r;
}

#if 0
printf("exec_newmem: r= %d, m_type= %d\n", r, m.m_type);
#endif
*stack_top= m.ml_i1;
*load_textp= !(m.ml_i2 & EXC_NM_RF_LOAD_TEXT);
*allow_setuidp= !(m.ml_i2 & EXC_NM_RF_ALLOW_SETUID);

#if 0
printf("exec_newmem: stack_top= 0x%x\n", *stack_top);
printf("exec_newmem: load_text= %d\n", *load_textp);
#endif
return m.m_type;
}

/*=====
*                               exec_restart                               *
*=====*/
PRIVATE int exec_restart(proc_e, result)
int proc_e;
int result;
{
    int r;
    message m;

    m.m_type= EXEC_RESTART;
    m.EXC_RS_PROC= proc_e;
    m.EXC_RS_RESULT= result;
    r= sendrec(PM_PROC_NR, &m);
    if (r != OK)
        return r;
    return m.m_type;
}

/*=====
*                               read_header                               *
*=====*/
PRIVATE int read_header(exec, exec_len, sep_id, text_bytes, data_bytes,
    bss_bytes, tot_bytes, pc, hdrlenp)
char *exec;
/* executable image */

```

```

size_t exec_len;          /* size of the image */
int *sep_id;              /* true iff sep I&D */
vir_bytes *text_bytes;    /* place to return text size */
vir_bytes *data_bytes;    /* place to return initialized data size */
vir_bytes *bss_bytes;     /* place to return bss size */
phys_bytes *tot_bytes;    /* place to return total size */
vir_bytes *pc;            /* program entry point (initial PC) */
int *hdrlenp;
{
/* Read the header and extract the text, data, bss and total sizes from it. */
off_t pos;
block_t b;
struct exec hdr;          /* a.out header is read in here */

/* Read the header and check the magic number. The standard MINIX header
 * is defined in <a.out.h>. It consists of 8 chars followed by 6 longs.
 * Then come 4 more longs that are not used here.
 *   Byte 0: magic number 0x01
 *   Byte 1: magic number 0x03
 *   Byte 2: normal = 0x10 (not checked, 0 is OK), separate I/D = 0x20
 *   Byte 3: CPU type, Intel 16 bit = 0x04, Intel 32 bit = 0x10,
 *           Motorola = 0x0B, Sun SPARC = 0x17
 *   Byte 4: Header length = 0x20
 *   Bytes 5-7 are not used.
 *
 *   Now come the 6 longs
 *   Bytes 8-11: size of text segments in bytes
 *   Bytes 12-15: size of initialized data segment in bytes
 *   Bytes 16-19: size of bss in bytes
 *   Bytes 20-23: program entry point
 *   Bytes 24-27: total memory allocated to program (text, data + stack)
 *   Bytes 28-31: size of symbol table in bytes
 * The longs are represented in a machine dependent order,
 * little-endian on the 8088, big-endian on the 68000.
 * The header is followed directly by the text and data segments, and the
 * symbol table (if any). The sizes are given in the header. Only the
 * text and data segments are copied into memory by exec. The header is
 * used here only. The symbol table is for the benefit of a debugger and
 * is ignored here.
 */
int r;

pos = 0;          /* Read from the start of the file */

if (exec_len < sizeof(hdr)) return(ENOEXEC);

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

/* Check magic number, cpu type, and flags. */
if (BADMAG(hdr)) return(ENOEXEC);
#if (CHIP == INTEL && _WORD_SIZE == 2)
if (hdr.a_cpu != A_I8086) return(ENOEXEC);
#endif
#if (CHIP == INTEL && _WORD_SIZE == 4)
if (hdr.a_cpu != A_I80386) return(ENOEXEC);
#endif
if ((hdr.a_flags & ~(A_NSYM | A_EXEC | A_SEP)) != 0) return(ENOEXEC);

*sep_id = !(hdr.a_flags & A_SEP);          /* separate I & D or not */

/* Get text and data sizes. */
*text_bytes = (vir_bytes) hdr.a_text; /* text size in bytes */
*data_bytes = (vir_bytes) hdr.a_data; /* data size in bytes */
*bss_bytes = (vir_bytes) hdr.a_bss; /* bss size in bytes */
*tot_bytes = hdr.a_total; /* total bytes to allocate for prog */
if (*tot_bytes == 0) return(ENOEXEC);

if (!*sep_id) {
/* If I & D space is not separated, it is all considered data. Text=0*/
*data_bytes += *text_bytes;
*text_bytes = 0;
}

*pc = hdr.a_entry; /* initial address to start execution */
*hdrlenp = hdr.a_hdrlen & BYTE; /* header length */

```

```

    return(OK);
}

/*=====
 *                               patch_ptr                               *
 *=====*/
PRIVATE void patch_ptr(stack, base)
char stack[ARG_MAX];          /* pointer to stack image within PM */
vir_bytes base;               /* virtual address of stack base inside user */
{
    /* When doing an exec(name, argv, envp) call, the user builds up a stack
     * image with arg and env pointers relative to the start of the stack. Now
     * these pointers must be relocated, since the stack is not positioned at
     * address 0 in the user's address space.
     */

    char **ap, flag;
    vir_bytes v;

    flag = 0;
    ap = (char **) stack;      /* counts number of 0-pointers seen */
    ap++;                      /* points initially to 'nargs' */
    /* now points to argv[0] */
    while (flag < 2) {
        if (ap >= (char **) &stack[ARG_MAX]) return; /* too bad */
        if (*ap != NULL) {
            v = (vir_bytes) *ap; /* v is relative pointer */
            v += base;           /* relocate it */
            *ap = (char *) v;    /* put it back */
        } else {
            flag++;
        }
        ap++;
    }
}

/*=====
 *                               read_seg                               *
 *=====*/
PRIVATE int read_seg(exec, exec_len, off, proc_e, seg, seg_bytes)
char *exec;                   /* executable image */
size_t exec_len;              /* size of the image */
off_t off;                    /* offset in file */
int proc_e;                   /* process number (endpoint) */
int seg;                      /* T, D, or S */
phys_bytes seg_bytes;         /* how much is to be transferred? */
{
    /*
     * The byte count on read is usually smaller than the segment count, because
     * a segment is padded out to a click multiple, and the data segment is only
     * partially initialized.
     */

    int r;
    off_t n, o, b_off, seg_off;

    if (off+seg_bytes > exec_len) return ENOEXEC;
    r = sys_vircopy(SELF, D, (vir_bytes)exec+off, proc_e, seg, 0, seg_bytes);
    return r;
}

```

```
# Makefile for System Process Manager (SM)
SERVER = sm
UTIL = service

# directories
u = /usr
i = $u/include
s = $i/sys
m = $i/minix
b = $i/ibm

# programs, flags, etc.
CC =      exec cc
CFLAGS = -I$i
LDFLAGS = -i
UTIL_LIBS = -lsys
LIBS = -lsys -lsysutil

UTIL_OBJ = service.o
OBJ = sm.o manager.o

# build local binary
all build:      $(SERVER) $(UTIL)
$(UTIL):      $(UTIL_OBJ)
              $(CC) -o $@ $(LDFLAGS) $(UTIL_OBJ) $(UTIL_LIBS)
$(SERVER):    $(OBJ)
              $(CC) -o $@ $(LDFLAGS) $(OBJ) $(LIBS)

# install with other servers
install:      /bin/$(UTIL) /usr/sbin/$(SERVER)
/bin/$(UTIL):    $(UTIL)
                install -c $? $@
/usr/sbin/$(SERVER): $(SERVER)
                install -o root -c $? $@

# clean up local files
clean:
                rm -f $(UTIL) $(SERVER) *.o *.bak

depend:
                /usr/bin/mkdep "$(CC) -E $(CPPFLAGS)" *.c > .depend

# Include generated dependencies.
include .depend
```


Table of Contents

1	Makefile.....	sheets	1 to	1 (1)	pages	1-	1	34	lines
2	Makefile.....	sheets	2 to	2 (1)	pages	2-	2	43	lines
3	glo.h.....	sheets	3 to	3 (1)	pages	3-	3	8	lines
4	inc.h.....	sheets	4 to	4 (1)	pages	4-	4	31	lines
5	main.c.....	sheets	5 to	7 (3)	pages	5-	7	152	lines
6	proto.h.....	sheets	8 to	8 (1)	pages	8-	8	11	lines
7	store.c.....	sheets	9 to	11 (3)	pages	9-	11	164	lines
8	store.h.....	sheets	12 to	12 (1)	pages	12-	12	20	lines
9	Makefile.....	sheets	13 to	13 (1)	pages	13-	13	36	lines
10	buf.h.....	sheets	14 to	15 (2)	pages	14-	15	76	lines
11	cache.c.....	sheets	16 to	23 (8)	pages	16-	23	559	lines
12	cache2.c.....	sheets	24 to	25 (2)	pages	24-	25	127	lines
13	const.h.....	sheets	26 to	27 (2)	pages	26-	27	106	lines
14	device.c.....	sheets	28 to	37 (10)	pages	28-	37	708	lines
15	dmap.c.....	sheets	38 to	41 (4)	pages	38-	41	279	lines
16	file.h.....	sheets	42 to	42 (1)	pages	42-	42	25	lines
17	filedes.c.....	sheets	43 to	44 (2)	pages	43-	44	130	lines
18	fproc.h.....	sheets	45 to	45 (1)	pages	45-	45	42	lines
19	fs.h.....	sheets	46 to	46 (1)	pages	46-	46	28	lines
20	glo.h.....	sheets	47 to	47 (1)	pages	47-	47	33	lines
21	inode.c.....	sheets	48 to	52 (5)	pages	48-	52	367	lines
22	inode.h.....	sheets	53 to	53 (1)	pages	53-	53	45	lines
23	link.c.....	sheets	54 to	62 (9)	pages	54-	62	617	lines
24	lock.c.....	sheets	63 to	65 (3)	pages	63-	65	187	lines
25	lock.h.....	sheets	66 to	66 (1)	pages	66-	66	11	lines
26	main.c.....	sheets	67 to	72 (6)	pages	67-	72	439	lines
27	misc.c.....	sheets	73 to	84 (12)	pages	73-	84	820	lines
28	mount.c.....	sheets	85 to	89 (5)	pages	85-	89	354	lines
29	open.c.....	sheets	90 to	97 (8)	pages	90-	97	574	lines
30	param.h.....	sheets	98 to	98 (1)	pages	98-	98	66	lines
31	path.c.....	sheets	99 to	105 (7)	pages	99-	105	487	lines
32	pipe.c.....	sheets	106 to	111 (6)	pages	106-	111	444	lines
33	protect.c.....	sheets	112 to	115 (4)	pages	112-	115	224	lines
34	proto.h.....	sheets	116 to	119 (4)	pages	116-	119	229	lines
35	read.c.....	sheets	120 to	127 (8)	pages	120-	127	561	lines
36	select.c.....	sheets	128 to	137 (10)	pages	128-	137	703	lines
37	select.h.....	sheets	138 to	138 (1)	pages	138-	138	10	lines
38	stadir.c.....	sheets	139 to	143 (5)	pages	139-	143	304	lines
39	super.c.....	sheets	144 to	148 (5)	pages	144-	148	320	lines
40	super.h.....	sheets	149 to	149 (1)	pages	149-	149	60	lines
41	table.c.....	sheets	150 to	151 (2)	pages	150-	151	120	lines
42	time.c.....	sheets	152 to	152 (1)	pages	152-	152	67	lines
43	timers.c.....	sheets	153 to	153 (1)	pages	153-	153	67	lines
44	type.h.....	sheets	154 to	154 (1)	pages	154-	154	24	lines
45	utility.c.....	sheets	155 to	157 (3)	pages	155-	157	167	lines
46	write.c.....	sheets	158 to	162 (5)	pages	158-	162	339	lines
47	exec.c.....	sheets	163 to	171 (9)	pages	163-	171	623	lines
48	Makefile.....	sheets	172 to	172 (1)	pages	172-	172	46	lines
49	buf.c.....	sheets	173 to	189 (17)	pages	173-	189	1257	lines
50	clock.c.....	sheets	190 to	193 (4)	pages	190-	193	232	lines
51	const.h.....	sheets	194 to	194 (1)	pages	194-	194	35	lines
52	inet.c.....	sheets	195 to	199 (5)	pages	195-	199	344	lines
53	inet.h.....	sheets	200 to	201 (2)	pages	200-	201	119	lines
54	inet_config.c.....	sheets	202 to	206 (5)	pages	202-	206	368	lines
55	inet_config.h.....	sheets	207 to	208 (2)	pages	207-	208	89	lines
56	mnx_eth.c.....	sheets	209 to	220 (12)	pages	209-	220	873	lines
57	mq.c.....	sheets	221 to	221 (1)	pages	221-	221	59	lines
58	mq.h.....	sheets	222 to	222 (1)	pages	222-	222	28	lines
59	osdep_eth.h.....	sheets	223 to	223 (1)	pages	223-	223	34	lines
60	proto.h.....	sheets	224 to	224 (1)	pages	224-	224	29	lines
61	qp.c.....	sheets	225 to	227 (3)	pages	225-	227	175	lines
62	qp.h.....	sheets	228 to	228 (1)	pages	228-	228	22	lines
63	sha2.c.....	sheets	229 to	243 (15)	pages	229-	243	1094	lines
64	sha2.h.....	sheets	244 to	246 (3)	pages	244-	246	169	lines
65	sr.c.....	sheets	247 to	262 (16)	pages	247-	262	1146	lines
66	sr_int.h.....	sheets	263 to	263 (1)	pages	263-	263	55	lines
67	stacktrace.c.....	sheets	264 to	264 (1)	pages	264-	264	36	lines
68	version.c.....	sheets	265 to	265 (1)	pages	265-	265	12	lines
69	arp.c.....	sheets	266 to	284 (19)	pages	266-	284	1360	lines
70	arp.h.....	sheets	285 to	285 (1)	pages	285-	285	33	lines
71	assert.h.....	sheets	286 to	286 (1)	pages	286-	286	32	lines
72	buf.h.....	sheets	287 to	290 (4)	pages	287-	290	250	lines
73	clock.h.....	sheets	291 to	291 (1)	pages	291-	291	40	lines

74	<i>eth.c</i>	sheets	292 to 308 (17)	pages	292-308	1227 lines
75	<i>eth.h</i>	sheets	309 to 309 (1)	pages	309-309	40 lines
76	<i>eth_int.h</i>	sheets	310 to 310 (1)	pages	310-310	64 lines
77	<i>event.c</i>	sheets	311 to 311 (1)	pages	311-311	70 lines
78	<i>event.h</i>	sheets	312 to 312 (1)	pages	312-312	43 lines
79	<i>icmp.c</i>	sheets	313 to 329 (17)	pages	313-329	1230 lines
80	<i>icmp.h</i>	sheets	330 to 330 (1)	pages	330-330	31 lines
81	<i>icmp_lib.h</i>	sheets	331 to 331 (1)	pages	331-331	26 lines
82	<i>io.c</i>	sheets	332 to 332 (1)	pages	332-332	35 lines
83	<i>io.h</i>	sheets	333 to 333 (1)	pages	333-333	22 lines
84	<i>ip.c</i>	sheets	334 to 340 (7)	pages	334-340	471 lines
85	<i>ip.h</i>	sheets	341 to 341 (1)	pages	341-341	29 lines
86	<i>ip_eth.c</i>	sheets	342 to 351 (10)	pages	342-351	723 lines
87	<i>ip_int.h</i>	sheets	352 to 354 (3)	pages	352-354	190 lines
88	<i>ip_ioctl.c</i>	sheets	355 to 364 (10)	pages	355-364	672 lines
89	<i>ip_lib.c</i>	sheets	365 to 368 (4)	pages	365-368	238 lines
90	<i>ip_ps.c</i>	sheets	369 to 372 (4)	pages	369-372	276 lines
91	<i>ip_read.c</i>	sheets	373 to 387 (15)	pages	373-387	1050 lines
92	<i>ip_write.c</i>	sheets	388 to 394 (7)	pages	388-394	516 lines
93	<i>ipr.c</i>	sheets	395 to 411 (17)	pages	395-411	1247 lines
94	<i>ipr.h</i>	sheets	412 to 413 (2)	pages	412-413	91 lines
95	<i>psip.c</i>	sheets	414 to 425 (12)	pages	414-425	819 lines
96	<i>psip.h</i>	sheets	426 to 426 (1)	pages	426-426	24 lines
97	<i>rand256.c</i>	sheets	427 to 427 (1)	pages	427-427	38 lines
98	<i>rand256.h</i>	sheets	428 to 428 (1)	pages	428-428	17 lines
99	<i>sr.h</i>	sheets	429 to 429 (1)	pages	429-429	50 lines
100	<i>tcp.c</i>	sheets	430 to 466 (37)	pages	430-466	2724 lines
101	<i>tcp.h</i>	sheets	467 to 468 (2)	pages	467-468	96 lines
102	<i>tcp_int.h</i>	sheets	469 to 472 (4)	pages	469-472	267 lines
103	<i>tcp_lib.c</i>	sheets	473 to 480 (8)	pages	473-480	593 lines
104	<i>tcp_recv.c</i>	sheets	481 to 501 (21)	pages	481-501	1486 lines
105	<i>tcp_send.c</i>	sheets	502 to 521 (20)	pages	502-521	1444 lines
106	<i>type.h</i>	sheets	522 to 522 (1)	pages	522-522	22 lines
107	<i>udp.c</i>	sheets	523 to 545 (23)	pages	523-545	1675 lines
108	<i>udp.h</i>	sheets	546 to 546 (1)	pages	546-546	37 lines
109	<i>udp_int.h</i>	sheets	547 to 547 (1)	pages	547-547	74 lines
110	<i>queryparam.c</i>	sheets	548 to 550 (3)	pages	548-550	152 lines
111	<i>queryparam.h</i>	sheets	551 to 551 (1)	pages	551-551	46 lines
112	<i>Makefile</i>	sheets	552 to 552 (1)	pages	552-552	39 lines
113	<i>init.c</i>	sheets	553 to 559 (7)	pages	553-559	461 lines
114	<i>Makefile</i>	sheets	560 to 560 (1)	pages	560-560	42 lines
115	<i>dmp.c</i>	sheets	561 to 562 (2)	pages	561-562	103 lines
116	<i>dmp_ds.c</i>	sheets	563 to 563 (1)	pages	563-563	61 lines
117	<i>dmp_fs.c</i>	sheets	564 to 565 (2)	pages	564-565	83 lines
118	<i>dmp_kernel.c</i>	sheets	566 to 573 (8)	pages	566-573	561 lines
119	<i>dmp_pm.c</i>	sheets	574 to 575 (2)	pages	574-575	141 lines
120	<i>dmp_rs.c</i>	sheets	576 to 576 (1)	pages	576-576	63 lines
121	<i>glo.h</i>	sheets	577 to 577 (1)	pages	577-577	19 lines
122	<i>inc.h</i>	sheets	578 to 578 (1)	pages	578-578	34 lines
123	<i>is.h</i>	sheets	579 to 579 (1)	pages	579-579	34 lines
124	<i>main.c</i>	sheets	580 to 582 (3)	pages	580-582	174 lines
125	<i>proto.h</i>	sheets	583 to 583 (1)	pages	583-583	37 lines
126	<i>Makefile</i>	sheets	584 to 584 (1)	pages	584-584	40 lines
127	<i>alloc.c</i>	sheets	585 to 591 (7)	pages	585-591	446 lines
128	<i>break.c</i>	sheets	592 to 594 (3)	pages	592-594	182 lines
129	<i>const.h</i>	sheets	595 to 595 (1)	pages	595-595	24 lines
130	<i>exec.c</i>	sheets	596 to 601 (6)	pages	596-601	379 lines
131	<i>forkexit.c</i>	sheets	602 to 608 (7)	pages	602-608	454 lines
132	<i>getset.c</i>	sheets	609 to 610 (2)	pages	609-610	124 lines
133	<i>glo.h</i>	sheets	611 to 611 (1)	pages	611-611	34 lines
134	<i>main.c</i>	sheets	612 to 624 (13)	pages	612-624	921 lines
135	<i>misc.c</i>	sheets	625 to 632 (8)	pages	625-632	523 lines
136	<i>mproc.h</i>	sheets	633 to 634 (2)	pages	633-634	90 lines
137	<i>param.h</i>	sheets	635 to 635 (1)	pages	635-635	67 lines
138	<i>pm.h</i>	sheets	636 to 636 (1)	pages	636-636	27 lines
139	<i>proto.h</i>	sheets	637 to 638 (2)	pages	637-638	122 lines
140	<i>signal.c</i>	sheets	639 to 649 (11)	pages	639-649	791 lines
141	<i>table.c</i>	sheets	650 to 651 (2)	pages	650-651	118 lines
142	<i>time.c</i>	sheets	652 to 653 (2)	pages	652-653	87 lines
143	<i>timers.c</i>	sheets	654 to 655 (2)	pages	654-655	80 lines
144	<i>trace.c</i>	sheets	656 to 657 (2)	pages	656-657	119 lines
145	<i>type.h</i>	sheets	658 to 658 (1)	pages	658-658	6 lines
146	<i>utility.c</i>	sheets	659 to 661 (3)	pages	659-661	172 lines
147	<i>Makefile</i>	sheets	662 to 662 (1)	pages	662-662	47 lines

148	<i>inc.h</i>	sheets	663	to	663	(1)	pages	663-663	38	lines
149	<i>main.c</i>	sheets	664	to	666	(3)	pages	664-666	205	lines
150	<i>manager.c</i>	sheets	667	to	674	(8)	pages	667-674	544	lines
151	<i>manager.h</i>	sheets	675	to	675	(1)	pages	675-675	56	lines
152	<i>proto.h</i>	sheets	676	to	676	(1)	pages	676-676	21	lines
153	<i>service.c</i>	sheets	677	to	681	(5)	pages	677-681	306	lines
154	<i>exec.c</i>	sheets	682	to	687	(6)	pages	682-687	432	lines
155	<i>Makefile</i>	sheets	688	to	688	(1)	pages	688-688	46	lines