```
/* Mode: C;
 * ifenslave.c: Configure network interfaces for parallel routing.
 *
 *  This program controls the Linux implementation of running multiple
 *  network interfaces in parallel.
 *
 * Author:  Donald Becker <becker@cesdis.gsfc.nasa.gov>
 *      Copyright 1994-1996 Donald Becker
 *
 *      This program is free software; you can redistribute it
 *      and/or modify it under the terms of the GNU General Public
 *      License as published by the Free Software Foundation.
 *
 *  The author may be reached as becker@CESDIS.gsfc.nasa.gov, or C/O
 *  Center of Excellence in Space Data and Information Sciences
 *     Code 930.5, Goddard Space Flight Center, Greenbelt MD 20771
 *
 *  Changes :
 *    - 2000/10/02 Willy Tarreau <willy at meta-x.org> :
 *       - few fixes. Master's MAC address is now correctly taken from
 *         the first device when not previously set ;
 *       - detach support : call BOND_RELEASE to detach an enslaved interface.
 *       - give a mini-howto from command-line help : # ifenslave -h
 *
 *    - 2001/02/16 Chad N. Tindel <ctindel at ieee dot org> :
 *       - Master is now brought down before setting the MAC address.  In
 *         the 2.4 kernel you can't change the MAC address while the device is
 *         up because you get EBUSY.
 *
 *    - 2001/09/13 Takao Indoh <indou dot takao at jp dot fujitsu dot com>
 *       - Added the ability to change the active interface on a mode 1 bond
 *         at runtime.
 *
 *    - 2001/10/23 Chad N. Tindel <ctindel at ieee dot org> :
 *       - No longer set the MAC address of the master.  The bond device will
 *         take care of this itself
 *       - Try the SIOC*** versions of the bonding ioctls before using the
 *         old versions
 *    - 2002/02/18 Erik Habbinga <erik_habbinga @ hp dot com> :
 *       - ifr2.ifr_flags was not initialized in the hwaddr_notset case,
 *         SIOCGIFFLAGS now called before hwaddr_notset test
 *
 *    - 2002/10/31 Tony Cureington <tony.cureington * hp_com> :
 *       - If the master does not have a hardware address when the first slave
 *         is enslaved, the master is assigned the hardware address of that
 *         slave - there is a comment in bonding.c stating "ifenslave takes
 *         care of this now." This corrects the problem of slaves having
 *         different hardware addresses in active-backup mode when
 *         multiple interfaces are specified on a single ifenslave command
 *         (ifenslave bond0 eth0 eth1).
 *
 *    - 2003/03/18 - Tsippy Mendelson <tsippy.mendelson at intel dot com> and
 *                   Shmulik Hen <shmulik.hen at intel dot com>
 *       - Moved setting the slave's mac address and openning it, from
 *         the application to the driver. This enables support of modes
 *         that need to use the unique mac address of each slave.
 *         The driver also takes care of closing the slave and restoring its
 *         original mac address upon release.
 *         In addition, block possibility of enslaving before the master is up.
 *         This prevents putting the system in an undefined state.
 *
 *    - 2003/05/01 - Amir Noam <amir.noam at intel dot com>
 *       - Added ABI version control to restore compatibility between
 *         new/old ifenslave and new/old bonding.
 *       - Prevent adding an adapter that is already a slave.
 *         Fixes the problem of stalling the transmission and leaving
```

```
 *         the slave in a down state.
 *
 *    - 2003/05/01 - Shmulik Hen <shmulik.hen at intel dot com>
 *       - Prevent enslaving if the bond device is down.
 *         Fixes the problem of leaving the system in unstable state and
 *         halting when trying to remove the module.
 *       - Close socket on all abnormal exists.
 *       - Add versioning scheme that follows that of the bonding driver.
 *         current version is 1.0.0 as a base line.
 *
 *    - 2003/05/22 - Jay Vosburgh <fubar at us dot ibm dot com>
 *   - ifenslave -c was broken; it's now fixed
 *   - Fixed problem with routes vanishing from master during enslave
 *     processing.
 *
 *    - 2003/05/27 - Amir Noam <amir.noam at intel dot com>
 *   - Fix backward compatibility issues:
 *     For drivers not using ABI versions, slave was set down while
 *     it should be left up before enslaving.
 *     Also, master was not set down and the default set_mac_address()
 *     would fail and generate an error message in the system log.
 *   - For opt_c: slave should not be set to the master's setting
 *     while it is running. It was already set during enslave. To
 *     simplify things, it is now handled separately.
 *
 *    - 2003/12/01 - Shmulik Hen <shmulik.hen at intel dot com>
 *   - Code cleanup and style changes
 *     set version to 1.1.0
 */

#define APP_VERSION "1.1.0"
#define APP_RELDATE "December 1, 2003"
#define APP_NAME    "ifenslave"

static char *version =
APP_NAME ".c:v" APP_VERSION " (" APP_RELDATE ")\n"
"o Donald Becker (becker@cesdis.gsfc.nasa.gov).\n"
"o Detach support added on 2000/10/02 by Willy Tarreau (willy at meta-x.org).\n"
"o 2.4 kernel support added on 2001/02/16 by Chad N. Tindel\n"
"  (ctindel at ieee dot org).\n";

static const char *usage_msg =
"Usage: ifenslave [-f] <master-if> <slave-if> [<slave-if>...]\n"
"       ifenslave -d   <master-if> <slave-if> [<slave-if>...]\n"
"       ifenslave -c   <master-if> <slave-if>\n"
"       ifenslave --help\n";

static const char *help_msg =
"\n"
"       To create a bond device, simply follow these three steps :\n"
"       - ensure that the required drivers are properly loaded :\n"
"         # modprobe bonding ; modprobe <3c59x|eepro100|pcnet32|tulip|...>\n"
"       - assign an IP address to the bond device :\n"
"         # ifconfig bond0 <addr> netmask <mask> broadcast <bcast>\n"
"       - attach all the interfaces you need to the bond device :\n"
"         # ifenslave [{-f|--force}] bond0 eth0 [eth1 [eth2]...]\n"
"         If bond0 didn't have a MAC address, it will take eth0's. Then, all\n"
"         interfaces attached AFTER this assignment will get the same MAC addr.\n"
"         (except for ALB/TLB modes)\n"
"\n"
"       To set the bond device down and automatically release all the slaves :\n"
"         # ifconfig bond0 down\n"
"\n"
"       To detach a dead interface without setting the bond device down :\n"
"         # ifenslave {-d|--detach} bond0 eth0 [eth1 [eth2]...]\n"
"\n"
```

```
"       To change active slave :\n"
"         # ifenslave {-c|--change-active} bond0 eth0\n"
"\n"
"       To show master interface info\n"
"         # ifenslave bond0\n"
"\n"
"       To show all interfaces info\n"
"       # ifenslave {-a|--all-interfaces}\n"
"\n"
"       To be more verbose\n"
"       # ifenslave {-v|--verbose} ...\n"
"\n"
"       # ifenslave {-u|--usage}   Show usage\n"
"       # ifenslave {-V|--version} Show version\n"
"       # ifenslave {-h|--help}    This message\n"
"\n";

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <getopt.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <linux/if.h>
#include <net/if_arp.h>
#include <linux/if_ether.h>
#include <linux/if_bonding.h>
#include <linux/sockios.h>

typedef unsigned long long u64; /* hack, so we may include kernel's ethtool.h */
typedef __uint32_t u32;     /* ditto */
typedef __uint16_t u16;     /* ditto */
typedef __uint8_t u8;       /* ditto */
#include <linux/ethtool.h>

struct option longopts[] = {
    /* { name  has_arg  *flag  val } */
    {"all-interfaces",  0, 0, 'a'}, /* Show all interfaces. */
    {"change-active",   0, 0, 'c'}, /* Change the active slave.  */
    {"detach",      0, 0, 'd'}, /* Detach a slave interface. */
    {"force",       0, 0, 'f'}, /* Force the operation. */
    {"help",        0, 0, 'h'}, /* Give help */
    {"usage",       0, 0, 'u'}, /* Give usage */
    {"verbose",     0, 0, 'v'}, /* Report each action taken. */
    {"version",     0, 0, 'V'}, /* Emit version information. */
    { 0, 0, 0, 0}
};

/* Command-line flags. */
unsigned int
opt_a = 0,  /* Show-all-interfaces flag. */
opt_c = 0,  /* Change-active-slave flag. */
opt_d = 0,  /* Detach a slave interface. */
opt_f = 0,  /* Force the operation. */
opt_h = 0,  /* Help */
opt_u = 0,  /* Usage */
opt_v = 0,  /* Verbose flag. */
opt_V = 0;  /* Version */

int skfd = -1;      /* AF_INET socket for ioctl() calls.*/
int abi_ver = 0;    /* userland - kernel ABI version */
```

```
int hwaddr_set = 0; /* Master's hwaddr is set */
int saved_errno;

struct ifreq master_mtu, master_flags, master_hwaddr;
struct ifreq slave_mtu, slave_flags, slave_hwaddr;

struct dev_ifr {
    struct ifreq *req_ifr;
    char *req_name;
    int req_type;
};

struct dev_ifr master_ifra[] = {
    {&master_mtu,     "SIOCGIFMTU",     SIOCGIFMTU},
    {&master_flags,   "SIOCGIFFLAGS",   SIOCGIFFLAGS},
    {&master_hwaddr,  "SIOCGIFHWADDR",  SIOCGIFHWADDR},
    {NULL, "", 0}
};

struct dev_ifr slave_ifra[] = {
    {&slave_mtu,     "SIOCGIFMTU",     SIOCGIFMTU},
    {&slave_flags,   "SIOCGIFFLAGS",   SIOCGIFFLAGS},
    {&slave_hwaddr,  "SIOCGIFHWADDR",  SIOCGIFHWADDR},
    {NULL, "", 0}
};

static void if_print(char *ifname);
static int get_drv_info(char *master_ifname);
static int get_if_settings(char *ifname, struct dev_ifr ifra[]);
static int get_slave_flags(char *slave_ifname);
static int set_master_hwaddr(char *master_ifname, struct sockaddr *hwaddr);
static int set_slave_hwaddr(char *slave_ifname, struct sockaddr *hwaddr);
static int set_slave_mtu(char *slave_ifname, int mtu);
static int set_if_flags(char *ifname, short flags);
static int set_if_up(char *ifname, short flags);
static int set_if_down(char *ifname, short flags);
static int clear_if_addr(char *ifname);
static int set_if_addr(char *master_ifname, char *slave_ifname);
static int change_active(char *master_ifname, char *slave_ifname);
static int enslave(char *master_ifname, char *slave_ifname);
static int release(char *master_ifname, char *slave_ifname);
#define v_print(fmt, args...)   \
    if (opt_v)      \
        fprintf(stderr, fmt, ## args )

int main(int argc, char *argv[])
{
    char **spp, *master_ifname, *slave_ifname;
    int c, i, rv;
    int res = 0;
    int exclusive = 0;

    while ((c = getopt_long(argc, argv, "acdfhuvV", longopts, 0)) != EOF) {
        switch (c) {
        case 'a': opt_a++; exclusive++; break;
        case 'c': opt_c++; exclusive++; break;
        case 'd': opt_d++; exclusive++; break;
        case 'f': opt_f++; exclusive++; break;
        case 'h': opt_h++; exclusive++; break;
        case 'u': opt_u++; exclusive++; break;
        case 'v': opt_v++; break;
        case 'V': opt_V++; exclusive++; break;

        case '?':
            fprintf(stderr, usage_msg);
            res = 2;
```

```
        goto out;
    }
}

/* options check */
if (exclusive > 1) {
    fprintf(stderr, usage_msg);
    res = 2;
    goto out;
}

if (opt_v || opt_V) {
    printf(version);
    if (opt_V) {
        res = 0;
        goto out;
    }
}

if (opt_u) {
    printf(usage_msg);
    res = 0;
    goto out;
}

if (opt_h) {
    printf(usage_msg);
    printf(help_msg);
    res = 0;
    goto out;
}

/* Open a basic socket */
if ((skfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    perror("socket");
    res = 1;
    goto out;
}

if (opt_a) {
    if (optind == argc) {
        /* No remaining args */
        /* show all interfaces */
        if_print((char *)NULL);
        goto out;
    } else {
        /* Just show usage */
        fprintf(stderr, usage_msg);
        res = 2;
        goto out;
    }
}

/* Copy the interface name */
spp = argv + optind;
master_ifname = *spp++;

if (master_ifname == NULL) {
    fprintf(stderr, usage_msg);
    res = 2;
    goto out;
}

/* exchange abi version with bonding module */
res = get_drv_info(master_ifname);
if (res) {
```

```
    fprintf(stderr,
        "Master '%s': Error: handshake with driver failed. "
        "Aborting\n",
        master_ifname);
    goto out;
}

slave_ifname = *spp++;

if (slave_ifname == NULL) {
    if (opt_d || opt_c) {
        fprintf(stderr, usage_msg);
        res = 2;
        goto out;
    }

    /* A single arg means show the
     * configuration for this interface
     */
    if_print(master_ifname);
    goto out;
}

res = get_if_settings(master_ifname, master_ifra);
if (res) {
    /* Probably a good reason not to go on */
    fprintf(stderr,
        "Master '%s': Error: get settings failed: %s. "
        "Aborting\n",
        master_ifname, strerror(res));
    goto out;
}

/* check if master is indeed a master;
 * if not then fail any operation
 */
if (!(master_flags.ifr_flags & IFF_MASTER)) {
    fprintf(stderr,
        "Illegal operation; the specified interface '%s' "
        "is not a master. Aborting\n",
        master_ifname);
    res = 1;
    goto out;
}

/* check if master is up; if not then fail any operation */
if (!(master_flags.ifr_flags & IFF_UP)) {
    fprintf(stderr,
        "Illegal operation; the specified master interface "
        "'%s' is not up.\n",
        master_ifname);
    res = 1;
    goto out;
}

/* Only for enslaving */
if (!opt_c && !opt_d) {
    sa_family_t master_family = master_hwaddr.ifr_hwaddr.sa_family;
    unsigned char *hwaddr =
        (unsigned char *)master_hwaddr.ifr_hwaddr.sa_data;

    /* The family '1' is ARPHRD_ETHER for ethernet. */
    if (master_family != 1 && !opt_f) {
        fprintf(stderr,
            "Illegal operation: The specified master "
            "interface '%s' is not ethernet-like.\n "
```

```
            "This program is designed to work with "
            "ethernet-like network interfaces.\n "
            "Use the '-f' option to force the "
            "operation.\n",
            master_ifname);
        res = 1;
        goto out;
    }

    /* Check master's hw addr */
    for (i = 0; i < 6; i++) {
        if (hwaddr[i] != 0) {
            hwaddr_set = 1;
            break;
        }
    }

    if (hwaddr_set) {
        v_print("current hardware address of master '%s' "
            "is %2.2x:%2.2x:%2.2x:%2.2x:%2.2x:%2.2x, "
            "type %d\n",
            master_ifname,
            hwaddr[0], hwaddr[1],
            hwaddr[2], hwaddr[3],
            hwaddr[4], hwaddr[5],
            master_family);
    }
}

/* Accepts only one slave */
if (opt_c) {
    /* change active slave */
    res = get_slave_flags(slave_ifname);
    if (res) {
        fprintf(stderr,
            "Slave '%s': Error: get flags failed. "
            "Aborting\n",
            slave_ifname);
        goto out;
    }
    res = change_active(master_ifname, slave_ifname);
    if (res) {
        fprintf(stderr,
            "Master '%s', Slave '%s': Error: "
            "Change active failed\n",
            master_ifname, slave_ifname);
    }
} else {
    /* Accept multiple slaves */
    do {
        if (opt_d) {
            /* detach a slave interface from the master */
            rv = get_slave_flags(slave_ifname);
            if (rv) {
                /* Can't work with this slave. */
                /* remember the error and skip it*/
                fprintf(stderr,
                    "Slave '%s': Error: get flags "
                    "failed. Skipping\n",
                    slave_ifname);
                res = rv;
                continue;
            }
            rv = release(master_ifname, slave_ifname);
            if (rv) {
                fprintf(stderr,
```

```
                        "Master '%s', Slave '%s': Error: "
                        "Release failed\n",
                        master_ifname, slave_ifname);
                    res = rv;
                }
            } else {
                /* attach a slave interface to the master */
                rv = get_if_settings(slave_ifname, slave_ifra);
                if (rv) {
                    /* Can't work with this slave. */
                    /* remember the error and skip it*/
                    fprintf(stderr,
                        "Slave '%s': Error: get "
                        "settings failed: %s. "
                        "Skipping\n",
                        slave_ifname, strerror(rv));
                    res = rv;
                    continue;
                }
                rv = enslave(master_ifname, slave_ifname);
                if (rv) {
                    fprintf(stderr,
                        "Master '%s', Slave '%s': Error: "
                        "Enslave failed\n",
                        master_ifname, slave_ifname);
                    res = rv;
                }
            }
        } while ((slave_ifname = *spp++) != NULL);
    }

out:
    if (skfd >= 0) {
        close(skfd);
    }

    return res;
}

static short mif_flags;

/* Get the inteface configuration from the kernel. */
static int if_getconfig(char *ifname)
{
    struct ifreq ifr;
    int metric, mtu;    /* Parameters of the master interface. */
    struct sockaddr dstaddr, broadaddr, netmask;
    unsigned char *hwaddr;

    strcpy(ifr.ifr_name, ifname);
    if (ioctl(skfd, SIOCGIFFLAGS, &ifr) < 0)
        return -1;
    mif_flags = ifr.ifr_flags;
    printf("The result of SIOCGIFFLAGS on %s is %x.\n",
           ifname, ifr.ifr_flags);

    strcpy(ifr.ifr_name, ifname);
    if (ioctl(skfd, SIOCGIFADDR, &ifr) < 0)
        return -1;
    printf("The result of SIOCGIFADDR is %2.2x.%2.2x.%2.2x.%2.2x.\n",
           ifr.ifr_addr.sa_data[0], ifr.ifr_addr.sa_data[1],
           ifr.ifr_addr.sa_data[2], ifr.ifr_addr.sa_data[3]);

    strcpy(ifr.ifr_name, ifname);
    if (ioctl(skfd, SIOCGIFHWADDR, &ifr) < 0)
        return -1;
```

```
    /* Gotta convert from 'char' to unsigned for printf(). */
    hwaddr = (unsigned char *)ifr.ifr_hwaddr.sa_data;
    printf("The result of SIOCGIFHWADDR is type %d  "
           "%2.2x:%2.2x:%2.2x:%2.2x:%2.2x:%2.2x.\n",
           ifr.ifr_hwaddr.sa_family, hwaddr[0], hwaddr[1],
           hwaddr[2], hwaddr[3], hwaddr[4], hwaddr[5]);

    strcpy(ifr.ifr_name, ifname);
    if (ioctl(skfd, SIOCGIFMETRIC, &ifr) < 0) {
        metric = 0;
    } else
        metric = ifr.ifr_metric;

    strcpy(ifr.ifr_name, ifname);
    if (ioctl(skfd, SIOCGIFMTU, &ifr) < 0)
        mtu = 0;
    else
        mtu = ifr.ifr_mtu;

    strcpy(ifr.ifr_name, ifname);
    if (ioctl(skfd, SIOCGIFDSTADDR, &ifr) < 0) {
        memset(&dstaddr, 0, sizeof(struct sockaddr));
    } else
        dstaddr = ifr.ifr_dstaddr;

    strcpy(ifr.ifr_name, ifname);
    if (ioctl(skfd, SIOCGIFBRDADDR, &ifr) < 0) {
        memset(&broadaddr, 0, sizeof(struct sockaddr));
    } else
        broadaddr = ifr.ifr_broadaddr;

    strcpy(ifr.ifr_name, ifname);
    if (ioctl(skfd, SIOCGIFNETMASK, &ifr) < 0) {
        memset(&netmask, 0, sizeof(struct sockaddr));
    } else
        netmask = ifr.ifr_netmask;

    return 0;
}

static void if_print(char *ifname)
{
    char buff[1024];
    struct ifconf ifc;
    struct ifreq *ifr;
    int i;

    if (ifname == (char *)NULL) {
        ifc.ifc_len = sizeof(buff);
        ifc.ifc_buf = buff;
        if (ioctl(skfd, SIOCGIFCONF, &ifc) < 0) {
            perror("SIOCGIFCONF failed");
            return;
        }

        ifr = ifc.ifc_req;
        for (i = ifc.ifc_len / sizeof(struct ifreq); --i >= 0; ifr++) {
            if (if_getconfig(ifr->ifr_name) < 0) {
                fprintf(stderr,
                    "%s: unknown interface.\n",
                    ifr->ifr_name);
                continue;
            }

            if (((mif_flags & IFF_UP) == 0) && !opt_a) continue;
```

```
            /*ife_print(&ife);*/
        }
    } else {
        if (if_getconfig(ifname) < 0) {
            fprintf(stderr,
                "%s: unknown interface.\n", ifname);
        }
    }
}

static int get_drv_info(char *master_ifname)
{
    struct ifreq ifr;
    struct ethtool_drvinfo info;
    char *endptr;

    memset(&ifr, 0, sizeof(ifr));
    strncpy(ifr.ifr_name, master_ifname, IFNAMSIZ);
    ifr.ifr_data = (caddr_t)&info;

    info.cmd = ETHTOOL_GDRVINFO;
    strncpy(info.driver, "ifenslave", 32);
    snprintf(info.fw_version, 32, "%d", BOND_ABI_VERSION);

    if (ioctl(skfd, SIOCETHTOOL, &ifr) < 0) {
        if (errno == EOPNOTSUPP) {
            goto out;
        }

        saved_errno = errno;
        v_print("Master '%s': Error: get bonding info failed %s\n",
            master_ifname, strerror(saved_errno));
        return 1;
    }

    abi_ver = strtoul(info.fw_version, &endptr, 0);
    if (*endptr) {
                v_print("Master '%s': Error: got invalid string as an ABI "
            "version from the bonding module\n",
            master_ifname);
        return 1;
    }

out:
    v_print("ABI ver is %d\n", abi_ver);

    return 0;
}

static int change_active(char *master_ifname, char *slave_ifname)
{
    struct ifreq ifr;
    int res = 0;

    if (!(slave_flags.ifr_flags & IFF_SLAVE)) {
        fprintf(stderr,
            "Illegal operation: The specified slave interface "
            "'%s' is not a slave\n",
            slave_ifname);
        return 1;
    }

    strncpy(ifr.ifr_name, master_ifname, IFNAMSIZ);
    strncpy(ifr.ifr_slave, slave_ifname, IFNAMSIZ);
    if ((ioctl(skfd, SIOCBONDCHANGEACTIVE, &ifr) < 0) &&
        (ioctl(skfd, BOND_CHANGE_ACTIVE_OLD, &ifr) < 0)) {
```

```
        saved_errno = errno;
        v_print("Master '%s': Error: SIOCBONDCHANGEACTIVE failed: "
            "%s\n",
            master_ifname, strerror(saved_errno));
        res = 1;
    }

    return res;
}

static int enslave(char *master_ifname, char *slave_ifname)
{
    struct ifreq ifr;
    int res = 0;

    if (slave_flags.ifr_flags & IFF_SLAVE) {
        fprintf(stderr,
            "Illegal operation: The specified slave interface "
            "'%s' is already a slave\n",
            slave_ifname);
        return 1;
    }

    res = set_if_down(slave_ifname, slave_flags.ifr_flags);
    if (res) {
        fprintf(stderr,
            "Slave '%s': Error: bring interface down failed\n",
            slave_ifname);
        return res;
    }

    if (abi_ver < 2) {
        /* Older bonding versions would panic if the slave has no IP
         * address, so get the IP setting from the master.
         */
        set_if_addr(master_ifname, slave_ifname);
    } else {
        res = clear_if_addr(slave_ifname);
        if (res) {
            fprintf(stderr,
                "Slave '%s': Error: clear address failed\n",
                slave_ifname);
            return res;
        }
    }

    if (master_mtu.ifr_mtu != slave_mtu.ifr_mtu) {
        res = set_slave_mtu(slave_ifname, master_mtu.ifr_mtu);
        if (res) {
            fprintf(stderr,
                "Slave '%s': Error: set MTU failed\n",
                slave_ifname);
            return res;
        }
    }

    if (hwaddr_set) {
        /* Master already has an hwaddr
         * so set it's hwaddr to the slave
         */
        if (abi_ver < 1) {
            /* The driver is using an old ABI, so
             * the application sets the slave's
             * hwaddr
             */
            res = set_slave_hwaddr(slave_ifname,
```

```
                        &(master_hwaddr.ifr_hwaddr));
            if (res) {
                fprintf(stderr,
                    "Slave '%s': Error: set hw address "
                    "failed\n",
                    slave_ifname);
                goto undo_mtu;
            }

            /* For old ABI the application needs to bring the
             * slave back up
             */
            res = set_if_up(slave_ifname, slave_flags.ifr_flags);
            if (res) {
                fprintf(stderr,
                    "Slave '%s': Error: bring interface "
                    "down failed\n",
                    slave_ifname);
                goto undo_slave_mac;
            }
        }
        /* The driver is using a new ABI,
         * so the driver takes care of setting
         * the slave's hwaddr and bringing
         * it up again
         */
    } else {
        /* No hwaddr for master yet, so
         * set the slave's hwaddr to it
         */
        if (abi_ver < 1) {
            /* For old ABI, the master needs to be
             * down before setting its hwaddr
             */
            res = set_if_down(master_ifname, master_flags.ifr_flags);
            if (res) {
                fprintf(stderr,
                    "Master '%s': Error: bring interface "
                    "down failed\n",
                    master_ifname);
                goto undo_mtu;
            }
        }

        res = set_master_hwaddr(master_ifname,
                    &(slave_hwaddr.ifr_hwaddr));
        if (res) {
            fprintf(stderr,
                "Master '%s': Error: set hw address "
                "failed\n",
                master_ifname);
            goto undo_mtu;
        }

        if (abi_ver < 1) {
            /* For old ABI, bring the master
             * back up
             */
            res = set_if_up(master_ifname, master_flags.ifr_flags);
            if (res) {
                fprintf(stderr,
                    "Master '%s': Error: bring interface "
                    "up failed\n",
                    master_ifname);
                goto undo_master_mac;
            }
```

```
        }

        hwaddr_set = 1;
    }

    /* Do the real thing */
    strncpy(ifr.ifr_name, master_ifname, IFNAMSIZ);
    strncpy(ifr.ifr_slave, slave_ifname, IFNAMSIZ);
    if ((ioctl(skfd, SIOCBONDENSLAVE, &ifr) < 0) &&
        (ioctl(skfd, BOND_ENSLAVE_OLD, &ifr) < 0)) {
        saved_errno = errno;
        v_print("Master '%s': Error: SIOCBONDENSLAVE failed: %s\n",
            master_ifname, strerror(saved_errno));
        res = 1;
    }

    if (res) {
        goto undo_master_mac;
    }

    return 0;

/* rollback (best effort) */
undo_master_mac:
    set_master_hwaddr(master_ifname, &(master_hwaddr.ifr_hwaddr));
    hwaddr_set = 0;
    goto undo_mtu;
undo_slave_mac:
    set_slave_hwaddr(slave_ifname, &(slave_hwaddr.ifr_hwaddr));
undo_mtu:
    set_slave_mtu(slave_ifname, slave_mtu.ifr_mtu);
    return res;
}

static int release(char *master_ifname, char *slave_ifname)
{
    struct ifreq ifr;
    int res = 0;

    if (!(slave_flags.ifr_flags & IFF_SLAVE)) {
        fprintf(stderr,
            "Illegal operation: The specified slave interface "
            "'%s' is not a slave\n",
            slave_ifname);
        return 1;
    }

    strncpy(ifr.ifr_name, master_ifname, IFNAMSIZ);
    strncpy(ifr.ifr_slave, slave_ifname, IFNAMSIZ);
    if ((ioctl(skfd, SIOCBONDRELEASE, &ifr) < 0) &&
        (ioctl(skfd, BOND_RELEASE_OLD, &ifr) < 0)) {
        saved_errno = errno;
        v_print("Master '%s': Error: SIOCBONDRELEASE failed: %s\n",
            master_ifname, strerror(saved_errno));
        return 1;
    } else if (abi_ver < 1) {
        /* The driver is using an old ABI, so we'll set the interface
         * down to avoid any conflicts due to same MAC/IP
         */
        res = set_if_down(slave_ifname, slave_flags.ifr_flags);
        if (res) {
            fprintf(stderr,
                "Slave '%s': Error: bring interface "
                "down failed\n",
                slave_ifname);
        }
```

```
    }

    /* set to default mtu */
    set_slave_mtu(slave_ifname, 1500);

    return res;
}

static int get_if_settings(char *ifname, struct dev_ifr ifra[])
{
    int i;
    int res = 0;

    for (i = 0; ifra[i].req_ifr; i++) {
        strncpy(ifra[i].req_ifr->ifr_name, ifname, IFNAMSIZ);
        res = ioctl(skfd, ifra[i].req_type, ifra[i].req_ifr);
        if (res < 0) {
            saved_errno = errno;
            v_print("Interface '%s': Error: %s failed: %s\n",
                ifname, ifra[i].req_name,
                strerror(saved_errno));

            return saved_errno;
        }
    }

    return 0;
}

static int get_slave_flags(char *slave_ifname)
{
    int res = 0;

    strncpy(slave_flags.ifr_name, slave_ifname, IFNAMSIZ);
    res = ioctl(skfd, SIOCGIFFLAGS, &slave_flags);
    if (res < 0) {
        saved_errno = errno;
        v_print("Slave '%s': Error: SIOCGIFFLAGS failed: %s\n",
            slave_ifname, strerror(saved_errno));
    } else {
        v_print("Slave %s: flags %04X.\n",
            slave_ifname, slave_flags.ifr_flags);
    }

    return res;
}

static int set_master_hwaddr(char *master_ifname, struct sockaddr *hwaddr)
{
    unsigned char *addr = (unsigned char *)hwaddr->sa_data;
    struct ifreq ifr;
    int res = 0;

    strncpy(ifr.ifr_name, master_ifname, IFNAMSIZ);
    memcpy(&(ifr.ifr_hwaddr), hwaddr, sizeof(struct sockaddr));
    res = ioctl(skfd, SIOCSIFHWADDR, &ifr);
    if (res < 0) {
        saved_errno = errno;
        v_print("Master '%s': Error: SIOCSIFHWADDR failed: %s\n",
            master_ifname, strerror(saved_errno));
        return res;
    } else {
        v_print("Master '%s': hardware address set to "
            "%2.2x:%2.2x:%2.2x:%2.2x:%2.2x:%2.2x.\n",
            master_ifname, addr[0], addr[1], addr[2],
            addr[3], addr[4], addr[5]);
```

```
    }

    return res;
}

static int set_slave_hwaddr(char *slave_ifname, struct sockaddr *hwaddr)
{
    unsigned char *addr = (unsigned char *)hwaddr->sa_data;
    struct ifreq ifr;
    int res = 0;

    strncpy(ifr.ifr_name, slave_ifname, IFNAMSIZ);
    memcpy(&(ifr.ifr_hwaddr), hwaddr, sizeof(struct sockaddr));
    res = ioctl(skfd, SIOCSIFHWADDR, &ifr);
    if (res < 0) {
        saved_errno = errno;

        v_print("Slave '%s': Error: SIOCSIFHWADDR failed: %s\n",
            slave_ifname, strerror(saved_errno));

        if (saved_errno == EBUSY) {
            v_print("  The device is busy: it must be idle "
                "before running this command.\n");
        } else if (saved_errno == EOPNOTSUPP) {
            v_print("  The device does not support setting "
                "the MAC address.\n"
                "  Your kernel likely does not support slave "
                "devices.\n");
        } else if (saved_errno == EINVAL) {
            v_print("  The device's address type does not match "
                "the master's address type.\n");
        }
        return res;
    } else {
        v_print("Slave '%s': hardware address set to "
            "%2.2x:%2.2x:%2.2x:%2.2x:%2.2x:%2.2x.\n",
            slave_ifname, addr[0], addr[1], addr[2],
            addr[3], addr[4], addr[5]);
    }

    return res;
}

static int set_slave_mtu(char *slave_ifname, int mtu)
{
    struct ifreq ifr;
    int res = 0;

    ifr.ifr_mtu = mtu;
    strncpy(ifr.ifr_name, slave_ifname, IFNAMSIZ);

    res = ioctl(skfd, SIOCSIFMTU, &ifr);
    if (res < 0) {
        saved_errno = errno;
        v_print("Slave '%s': Error: SIOCSIFMTU failed: %s\n",
            slave_ifname, strerror(saved_errno));
    } else {
        v_print("Slave '%s': MTU set to %d.\n", slave_ifname, mtu);
    }

    return res;
}

static int set_if_flags(char *ifname, short flags)
{
    struct ifreq ifr;
```

```
    int res = 0;

    ifr.ifr_flags = flags;
    strncpy(ifr.ifr_name, ifname, IFNAMSIZ);

    res = ioctl(skfd, SIOCSIFFLAGS, &ifr);
    if (res < 0) {
        saved_errno = errno;
        v_print("Interface '%s': Error: SIOCSIFFLAGS failed: %s\n",
            ifname, strerror(saved_errno));
    } else {
        v_print("Interface '%s': flags set to %04X.\n", ifname, flags);
    }

    return res;
}

static int set_if_up(char *ifname, short flags)
{
    return set_if_flags(ifname, flags | IFF_UP);
}

static int set_if_down(char *ifname, short flags)
{
    return set_if_flags(ifname, flags & ~IFF_UP);
}

static int clear_if_addr(char *ifname)
{
    struct ifreq ifr;
    int res = 0;

    strncpy(ifr.ifr_name, ifname, IFNAMSIZ);
    ifr.ifr_addr.sa_family = AF_INET;
    memset(ifr.ifr_addr.sa_data, 0, sizeof(ifr.ifr_addr.sa_data));

    res = ioctl(skfd, SIOCSIFADDR, &ifr);
    if (res < 0) {
        saved_errno = errno;
        v_print("Interface '%s': Error: SIOCSIFADDR failed: %s\n",
            ifname, strerror(saved_errno));
    } else {
        v_print("Interface '%s': address cleared\n", ifname);
    }

    return res;
}

static int set_if_addr(char *master_ifname, char *slave_ifname)
{
    struct ifreq ifr;
    int res;
    unsigned char *ipaddr;
    int i;
    struct {
        char *req_name;
        char *desc;
        int g_ioctl;
        int s_ioctl;
    } ifra[] = {
        {"IFADDR", "addr", SIOCGIFADDR, SIOCSIFADDR},
        {"DSTADDR", "destination addr", SIOCGIFDSTADDR, SIOCSIFDSTADDR},
        {"BRDADDR", "broadcast addr", SIOCGIFBRDADDR, SIOCSIFBRDADDR},
        {"NETMASK", "netmask", SIOCGIFNETMASK, SIOCSIFNETMASK},
        {NULL, NULL, 0, 0},
    };
```

```
    for (i = 0; ifra[i].req_name; i++) {
        strncpy(ifr.ifr_name, master_ifname, IFNAMSIZ);
        res = ioctl(skfd, ifra[i].g_ioctl, &ifr);
        if (res < 0) {
            int saved_errno = errno;

            v_print("Interface '%s': Error: SIOCG%s failed: %s\n",
                master_ifname, ifra[i].req_name,
                strerror(saved_errno));

            ifr.ifr_addr.sa_family = AF_INET;
            memset(ifr.ifr_addr.sa_data, 0,
                   sizeof(ifr.ifr_addr.sa_data));
        }

        strncpy(ifr.ifr_name, slave_ifname, IFNAMSIZ);
        res = ioctl(skfd, ifra[i].s_ioctl, &ifr);
        if (res < 0) {
            int saved_errno = errno;

            v_print("Interface '%s': Error: SIOCS%s failed: %s\n",
                slave_ifname, ifra[i].req_name,
                strerror(saved_errno));

        }

        ipaddr = (unsigned char *)ifr.ifr_addr.sa_data;
        v_print("Interface '%s': set IP %s to %d.%d.%d.%d\n",
            slave_ifname, ifra[i].desc,
            ipaddr[0], ipaddr[1], ipaddr[2], ipaddr[3]);
    }

    return 0;
}

/*
 * Local variables:
 *  version-control: t
 *  kept-new-versions: 5
 *  c-indent-level: 4
 *  c-basic-offset: 4
 *  tab-width: 4
 *  compile-command: "gcc -Wall -Wstrict-prototypes -O -I/usr/src/linux/include ifenslave.c -o
ifenslave"
 * End:
 */
```