```
**********************************************************
   47494 Thu Feb 25 15:39:32 2016
new/usr/src/cmd/fm/fmadm/common/faulty.c
2976 remove useless offsetof() macros
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2004, 2010, Oracle and/or its affiliates. All rights reserved.
 */

#include <sys/types.h>
#include <fmadm.h>
#include <errno.h>
#include <limits.h>
#include <strings.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <fm/fmd_log.h>
#include <sys/fm/protocol.h>
#include <fm/libtopo.h>
#include <fm/fmd_adm.h>
#include <fm/fmd_msg.h>
#include <dlfcn.h>
#include <sys/systeminfo.h>
#include <sys/utsname.h>
#include <libintl.h>
#include <locale.h>
#include <sys/smbios.h>
#include <libdevinfo.h>
#include <stdlib.h>
#include <stddef.h>

#define offsetof(s, m)  ((size_t)(&(((s*)0)->m)))

/*
 * Fault records are added to catalog by calling add_fault_record_to_catalog()
 * records are stored in order of importance to the system.
 * If -g flag is set or not_suppressed is not set and the class fru, fault,
 * type are the same then details are merged into an existing record, with uuid
 * records are stored in time order.
 * For each record information is extracted from nvlist and merged into linked
 * list each is checked for identical records for which percentage certainty are
 * added together.
 * print_catalog() is called to print out catalog and release external resources
```

```
 *
 *                         /---------------\
 *      status_rec_list -> |               | -|
 *                         \---------------/
 *                                \/
 *                         /---------------\    /-------\    /-------\
 *      status_fru_list    | status_record | -> | uurec | -> | uurec | -|
 *            \/           |               | |- |       | <- |       |
 *      /-------------\    |               |    \-------/    \-------/
 *      |             | -> |               |       \/           \/
 *      \-------------/    |               |    /-------\    /-------\
 *            \/           |               | -> | asru  | -> | asru  |
 *            ---          |               |    |       | <- |       |
 *                         |               |    \-------/    \-------/
 *      status_asru_list   |  class        |
 *            \/           |  resource     |    /-------\    /-------\
 *      /-------------\    |  fru          | -> | list  | -> | list  |
 *      |             | -> |  serial       |    |       | <- |       |
 *      \-------------/    |               |    \-------/    \-------/
 *            \/           \---------------/
 *            ---               \/    /\
 *                         /---------------\
 *                         | status_record |
 *                         \---------------/
 *
 * Fmadm faulty takes a number of options which affect the format of the
 * output displayed. By default, the display reports the FRU and ASRU along
 * with other information on per-case basis as in the example below.
 *
 * --------------- ------------------------------------  -------------- -------
 * TIME            EVENT-ID                              MSG-ID         SEVERITY
 * --------------- ------------------------------------  -------------- -------
 * Sep 21 10:01:36 d482f935-5c8f-e9ab-9f25-d0aaafec1e6c  AMD-8000-2F    Major
 *
 * Fault class  : fault.memory.dimm_sb
 * Affects      : mem:///motherboard=0/chip=0/memory-controller=0/dimm=0/rank=0
 *                  faulted but still in service
 * FRU          : "CPU 0 DIMM 0" (hc://.../memory-controller=0/dimm=0)
 *                  faulty
 *
 * Description  : The number of errors associated with this memory module has
 *             exceeded acceptable levels.  Refer to
 *             http://illumos.org/msg/AMD-8000-2F for more information.
 *
 * Response     : Pages of memory associated with this memory module are being
 *             removed from service as errors are reported.
 *
 * Impact       : Total system memory capacity will be reduced as pages are
 *             retired.
 *
 * Action       : Schedule a repair procedure to replace the affected memory
 *             module.  Use fmdump -v -u <EVENT_ID> to identify the module.
 *
 * The -v flag is similar, but adds some additonal information such as the
 * resource. The -s flag is also similar but just gives the top line summary.
 * All these options (ie without the -f or -r flags) use the print_catalog()
 * function to do the display.
 *
 * The -f flag changes the output so that it appears sorted on a per-fru basis.
 * The output is somewhat cut down compared to the default output. If -f is
 * used, then print_fru() is used to print the output.
 *
 * -----------------------------------------------------------------------------
 * "SLOT 2" (hc://.../hostbridge=3/pciexrc=3/pciexbus=4/pciexdev=0) faulty
 * 5ca4aeb3-36...f6be-c2e8166dc484 2 suspects in this FRU total certainty 100%
 *
 * Description  : A problem was detected for a PCI device.
 *              Refer to http://illumos.org/msg/PCI-8000-7J
 *              for more information.
 *
 * Response     : One or more device instances may be disabled
 *
 * Impact       : Possible loss of services provided by the device instances
 *              associated with this fault
 *
 * Action       : Schedule a repair procedure to replace the affected device.
 *              Use fmdump -v -u <EVENT_ID> to identify the device or contact
 *              Sun for support.
 *
 * The -r flag changes the output so that it appears sorted on a per-asru basis.
 * The output is very much cut down compared to the default output, just giving
 * the asru fmri and state. Here print_asru() is used to print the output.
 *
 * mem:///motherboard=0/chip=0/memory-controller=0/dimm=0/rank=0        degraded
 *
 * For all fmadm faulty options, the sequence of events is
 *
 * 1) Walk through all the cases in the system using fmd_adm_case_iter() and
 * for each case call dfault_rec(). This will call add_fault_record_to_catalog()
 * This will extract the data from the nvlist and call catalog_new_record() to
 * save the data away in various linked lists in the catalogue.
 *
 * 2) Once this is done, the data can be supplemented by using
 * fmd_adm_rsrc_iter(). However this is now only necessary for the -i option.
 *
 * 3) Finally print_catalog(), print_fru() or print_asru() are called as
 * appropriate to display the information from the catalogue sorted in the
 * requested way.
 *
 */

typedef struct name_list {
        struct name_list *next;
        struct name_list *prev;
        char *name;
        uint8_t pct;
        uint8_t max_pct;
        ushort_t count;
        int status;
        char *label;
} name_list_t;
```
______***unchanged_portion_omitted_***

```
**********************************************************
   12270 Thu Feb 25 15:39:33 2016
new/usr/src/cmd/mdb/common/modules/ii/ii.c
2976 remove useless offsetof() macros
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <stddef.h>

#endif /* ! codereview */
#include <sys/types.h>
#include <sys/mdb_modapi.h>

#include <sys/nsctl/nsctl.h>
#include <sys/unistat/spcs_s.h>
#include <sys/unistat/spcs_s_k.h>


#include <sys/nsctl/dsw.h>
#include <sys/nsctl/dsw_dev.h>

#include <sys/nsctl/nsvers.h>

#define offsetof(s, m)  ((size_t)(&((s *)0)->m))


const mdb_bitmask_t bi_flags_bits[] = {
        { "DSW_GOLDEN", DSW_GOLDEN, DSW_GOLDEN },
        { "DSW_COPYINGP", DSW_COPYINGP, DSW_COPYINGP },
        { "DSW_COPYINGM", DSW_COPYINGM, DSW_COPYINGM },
        { "DSW_COPYINGS", DSW_COPYINGS, DSW_COPYINGS },
        { "DSW_COPYINGX", DSW_COPYINGX, DSW_COPYINGX },
        { "DSW_BMPOFFLINE", DSW_BMPOFFLINE, DSW_BMPOFFLINE },
        { "DSW_SHDOFFLINE", DSW_SHDOFFLINE, DSW_SHDOFFLINE },
        { "DSW_MSTOFFLINE", DSW_MSTOFFLINE, DSW_MSTOFFLINE },
        { "DSW_OVROFFLINE", DSW_OVROFFLINE, DSW_OVROFFLINE },
        { "DSW_TREEMAP", DSW_TREEMAP, DSW_TREEMAP },
        { "DSW_OVERFLOW", DSW_OVERFLOW, DSW_OVERFLOW },
        { "DSW_SHDEXPORT", DSW_SHDEXPORT, DSW_SHDEXPORT },
        { "DSW_SHDIMPORT", DSW_SHDIMPORT, DSW_SHDIMPORT },
        { "DSW_VOVERFLOW", DSW_VOVERFLOW, DSW_VOVERFLOW },
        { "DSW_HANGING", DSW_HANGING, DSW_HANGING },
        { "DSW_CFGOFFLINE", DSW_CFGOFFLINE, DSW_CFGOFFLINE },
```

```
        { "DSW_OVRHDRDRTY", DSW_OVRHDRDRTY, DSW_OVRHDRDRTY },
        { "DSW_RESIZED", DSW_RESIZED, DSW_RESIZED },
        { "DSW_FRECLAIM", DSW_FRECLAIM, DSW_FRECLAIM },
        { NULL, 0, 0 }
};
______unchanged_portion_omitted_
```

```
**********************************************************
    1669 Thu Feb 25 15:39:33 2016
new/usr/src/cmd/mdb/common/modules/libumem/misc.h
2976 remove useless offsetof() macros
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _MDBMOD_MISC_H
#define _MDBMOD_MISC_H

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <mdb/mdb_modapi.h>
#include <stddef.h>
#endif /* ! codereview */

#ifdef __cplusplus
extern "C" {
#endif

#define offsetof(s, m)  ((size_t)(&(((s *)0)->m)))

extern int umem_debug(uintptr_t, uint_t, int, const mdb_arg_t *);

extern int umem_set_standalone(void);
extern ssize_t umem_lookup_by_name(const char *, GElf_Sym *);
extern ssize_t umem_readvar(void *, const char *);

/*
 * Returns non-zero if sym matches libumem*‘prefix*
 */
int is_umem_sym(const char *, const char *);

#define dprintf(x) if (umem_debug_level) { \
        mdb_printf("umem debug: ");  \
        /*CSTYLED*/\
        mdb_printf x ;\
}
______unchanged_portion_omitted_
```

```
**********************************************************
   27115 Thu Feb 25 15:39:34 2016
new/usr/src/cmd/pools/poolstat/poolstat.c
2976 remove useless offsetof() macros
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * poolstat - report active pool statistics
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <unistd.h>
#include <locale.h>
#include <string.h>
#include <ctype.h>
#include <limits.h>
#include <errno.h>
#include <stddef.h>
#endif /* ! codereview */

#include <pool.h>
#include "utils.h"
#include "poolstat.h"
#include "poolstat_utils.h"
#include "statcommon.h"

#ifndef TEXT_DOMAIN
#define TEXT_DOMAIN     "SYS_TEST"
#endif

/* calculate offset of a particular element in a structure      */
#define offsetof(s, m)  ((size_t)(&(((s *)0)->m)))
#define addrof(s)  ((char **)&(s))

/* verify if a field is printable in respect of the current option flags */
#define PRINTABLE(i)    ((lf->plf_ffs[(i)].pff_prt & D_FIELD) || \
        (lf->plf_ffs[(i)].pff_prt & X_FIELD))

typedef int (* formatter) (char *, int, int, poolstat_field_format_t *, char *);

static uint_t timestamp_fmt = NODATE;
```

```
/* available field formatters   */
static int default_f(char *, int, int, poolstat_field_format_t *, char *);
static int bigno_f(char *, int, int, poolstat_field_format_t *, char *);
static int used_stat_f(char *, int, int, poolstat_field_format_t *, char *);
static int header_f(char *, int, int, poolstat_field_format_t *, char *);

/* statistics bags used to collect data from various provider   */
static statistic_bag_t  pool_sbag_s;
static statistic_bag_t  pset_sbag_s;
static statistic_bag_t  *pool_sbag = &pool_sbag_s;
static statistic_bag_t  *pset_sbag = &pset_sbag_s;

/* formatter objects for pset, defined in a default printing sequence   */
static poolstat_field_format_t pset_ffs[] = {
        /* prt flags,name,header,type,width,minwidth,offset,formatter   */
        { DX_FIELD, "id", "id", LL, 3, 1, addrof(pool_sbag),
                offsetof(statistic_bag_t, sb_sysid),
                (formatter)default_f },
        { DX_FIELD, "pool", "pool", STR, 20, 14, addrof(pool_sbag),
                offsetof(statistic_bag_t, sb_name),
                (formatter)default_f },
        { DX_FIELD, "type", "type", STR, 4, 5, addrof(pset_sbag),
                offsetof(statistic_bag_t, sb_type),
                (formatter)default_f },
        { D_FIELD, "rid", "rid", LL, 3, 1, addrof(pset_sbag_s.bag),
                offsetof(pset_statistic_bag_t, pset_sb_sysid),
                (formatter)default_f },
        { DX_FIELD, "rset", "rset", STR, 20, 14, addrof(pset_sbag),
                offsetof(statistic_bag_t, sb_name),
                (formatter)default_f },
        { DX_FIELD, "min", "min", ULL, 4, 1, addrof(pset_sbag_s.bag),
                offsetof(pset_statistic_bag_t, pset_sb_min),
                (formatter)bigno_f },
        { DX_FIELD, "max", "max", ULL, 4, 1, addrof(pset_sbag_s.bag),
                offsetof(pset_statistic_bag_t, pset_sb_max),
                (formatter)bigno_f },
        { DX_FIELD, "size", "size", ULL, 4, 1, addrof(pset_sbag_s.bag),
                offsetof(pset_statistic_bag_t, pset_sb_size),
                (formatter)default_f },
        { DX_FIELD, "used", "used", FL, 4, -1, addrof(pset_sbag_s.bag),
                offsetof(pset_statistic_bag_t, pset_sb_used),
                (formatter)used_stat_f },
        { DX_FIELD, "load", "load", FL, 4, -1, addrof(pset_sbag_s.bag),
                offsetof(pset_statistic_bag_t, pset_sb_load),
                (formatter)default_f }
};
______unchanged_portion_omitted_
```

```
**********************************************************
   26813 Thu Feb 25 15:39:34 2016
new/usr/src/cmd/stat/common/acquire_iodevs.c
2976 remove useless offsetof() macros
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2004, 2010, Oracle and/or its affiliates. All rights reserved.
 */

#include "statcommon.h"
#include "dsr.h"

#include <sys/dklabel.h>
#include <sys/dktp/fdisk.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stddef.h>
#endif /* ! codereview */
#include <unistd.h>
#include <strings.h>
#include <errno.h>
#include <limits.h>

static void insert_iodev(struct snapshot *ss, struct iodev_snapshot *iodev);

static struct iodev_snapshot *
make_controller(int cid)
{
        struct iodev_snapshot *new;

        new = safe_alloc(sizeof (struct iodev_snapshot));
        (void) memset(new, 0, sizeof (struct iodev_snapshot));
        new->is_type = IODEV_CONTROLLER;
        new->is_id.id = cid;
        new->is_parent_id.id = IODEV_NO_ID;

        (void) snprintf(new->is_name, sizeof (new->is_name), "c%d", cid);

        return (new);
}

static struct iodev_snapshot *
find_iodev_by_name(struct iodev_snapshot *list, const char *name)
{
        struct iodev_snapshot *pos;
        struct iodev_snapshot *pos2;
```

```
        for (pos = list; pos; pos = pos->is_next) {
                if (strcmp(pos->is_name, name) == 0)
                        return (pos);

                pos2 = find_iodev_by_name(pos->is_children, name);
                if (pos2 != NULL)
                        return (pos2);
        }

        return (NULL);
}

static enum iodev_type
parent_iodev_type(enum iodev_type type)
{
        switch (type) {
                case IODEV_CONTROLLER: return (0);
                case IODEV_IOPATH_LT: return (0);
                case IODEV_IOPATH_LI: return (0);
                case IODEV_NFS: return (0);
                case IODEV_TAPE: return (0);
                case IODEV_IOPATH_LTI: return (IODEV_DISK);
                case IODEV_DISK: return (IODEV_CONTROLLER);
                case IODEV_PARTITION: return (IODEV_DISK);
        }
        return (IODEV_UNKNOWN);
}

static int
id_match(struct iodev_id *id1, struct iodev_id *id2)
{
        return (id1->id == id2->id &&
            strcmp(id1->tid, id2->tid) == 0);
}

static struct iodev_snapshot *
find_parent(struct snapshot *ss, struct iodev_snapshot *iodev)
{
        enum iodev_type parent_type = parent_iodev_type(iodev->is_type);
        struct iodev_snapshot *pos;
        struct iodev_snapshot *pos2;

        if (parent_type == 0 || parent_type == IODEV_UNKNOWN)
                return (NULL);

        if (iodev->is_parent_id.id == IODEV_NO_ID &&
            iodev->is_parent_id.tid[0] == '\0')
                return (NULL);

        if (parent_type == IODEV_CONTROLLER) {
                for (pos = ss->s_iodevs; pos; pos = pos->is_next) {
                        if (pos->is_type != IODEV_CONTROLLER)
                                continue;
                        if (pos->is_id.id != iodev->is_parent_id.id)
                                continue;
                        return (pos);
                }

                if (!(ss->s_types & SNAP_CONTROLLERS))
                        return (NULL);

                pos = make_controller(iodev->is_parent_id.id);
                insert_iodev(ss, pos);
                return (pos);
        }
        /* IODEV_DISK parent */
        for (pos = ss->s_iodevs; pos; pos = pos->is_next) {
                if (id_match(&iodev->is_parent_id, &pos->is_id) &&
                    pos->is_type == IODEV_DISK)
                        return (pos);
                if (pos->is_type != IODEV_CONTROLLER)
                        continue;
                for (pos2 = pos->is_children; pos2; pos2 = pos2->is_next) {
                        if (pos2->is_type != IODEV_DISK)
                                continue;
                        if (id_match(&iodev->is_parent_id, &pos2->is_id))
                                return (pos2);
                }
        }

        return (NULL);
}

/*
 * Introduce an index into the list to speed up insert_into looking for the
 * right position in the list. This index is an AVL tree of all the
 * iodev_snapshot in the list.
 */

#define offsetof(s, m)  (size_t)(&(((s *)0)->m))        /* for avl_create */

static int
avl_iodev_cmp(const void* is1, const void* is2)
{
        int c = iodev_cmp((struct iodev_snapshot *)is1,
            (struct iodev_snapshot *)is2);

        if (c > 0)
                return (1);

        if (c < 0)
                return (-1);

        return (0);
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   75112 Thu Feb 25 15:39:35 2016
new/usr/src/common/nvpair/nvpair.c
2976 remove useless offsetof() macros
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
 */

#include <sys/stropts.h>
#include <sys/debug.h>
#include <sys/isa_defs.h>
#include <sys/int_limits.h>
#include <sys/nvpair.h>
#include <sys/nvpair_impl.h>
#include <rpc/types.h>
#include <rpc/xdr.h>

#if defined(_KERNEL) && !defined(_BOOT)
#include <sys/varargs.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/sysmacros.h>
#endif /* ! codereview */
#else
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <stddef.h>
#endif /* ! codereview */
#endif

#ifndef offsetof
#define offsetof(s, m)          ((size_t)(&(((s *)0)->m)))
#endif
#define skip_whitespace(p)      while ((*(p) == ' ') || (*(p) == '\t')) p++

/*
 * nvpair.c - Provides kernel & userland interfaces for manipulating
 *      name-value pairs.
 *
 * Overview Diagram
 *
 *  +--------------+
```

```
 *  |  nvlist_t    |
 *  |--------------|
 *  | nvl_version  |
 *  | nvl_nvflag   |
 *  | nvl_priv    -+-+
 *  | nvl_flag     | |
 *  | nvl_pad      | |
 *  +--------------+ |
 *                   V
 *      +--------------+      last i_nvp in list
 *      | nvpriv_t     |  +--------------------->
 *      |--------------|  |
 *   +--+- nvp_list    |  |   +------------+
 *   |  |  nvp_last   -+--+   + nv_alloc_t |
 *   |  |  nvp_curr    |      |------------|
 *   |  |  nvp_nva    -+----> | nva_ops    |
 *   |  |  nvp_stat    |      | nva_arg    |
 *   |  +--------------+      +------------+
 *   |
 *   +-------+
 *           V
 *   +---------------------+      +-------------------+
 *   |  i_nvp_t            |  +-->|  i_nvp_t          |  +-->
 *   |---------------------|  |   |-------------------|  |
 *   | nvi_next           -+--+   | nvi_next         -+--+
 *   | nvi_prev (NULL)     | <----+ nvi_prev          |
 *   | . . . . . . . . . . |      | . . . . . . . . . |
 *   | nvp (nvpair_t)      |      | nvp (nvpair_t)    |
 *   |  - nvp_size         |      |  - nvp_size       |
 *   |  - nvp_name_sz      |      |  - nvp_name_sz    |
 *   |  - nvp_value_elem   |      |  - nvp_value_elem |
 *   |  - nvp_type         |      |  - nvp_type       |
 *   |  - data ...         |      |  - data ...       |
 *   +---------------------+      +-------------------+
 *
 *
 *
 *   +---------------------+              +---------------------+
 *   |  i_nvp_t            |  +-->    +-->|  i_nvp_t (last)     |
 *   |---------------------|  |       |   |---------------------|
 *   |  nvi_next          -+--+ ... --+   | nvi_next (NULL)     |
 * <-+- nvi_prev           |<-- ...  <----+ nvi_prev            |
 *   | . . . . . . . . .   |              | . . . . . . . . .   |
 *   | nvp (nvpair_t)      |              | nvp (nvpair_t)      |
 *   |  - nvp_size         |              |  - nvp_size         |
 *   |  - nvp_name_sz      |              |  - nvp_name_sz      |
 *   |  - nvp_value_elem   |              |  - nvp_value_elem   |
 *   |  - DATA_TYPE_NVLIST |              |  - nvp_type         |
 *   |  - data (embedded)  |              |  - data ...         |
 *   |    nvlist name      |              +---------------------+
 *   |  +--------------+   |
 *   |  |  nvlist_t    |   |
 *   |  |--------------|   |
 *   |  | nvl_version  |   |
 *   |  | nvl_nvflag   |   |
 *   |  | nvl_priv   --+---+---->
 *   |  | nvl_flag     |   |
 *   |  | nvl_pad      |   |
 *   |  +--------------+   |
 *   +---------------------+
 *
 *
 * N.B. nvpair_t may be aligned on 4 byte boundary, so +4 will
 * allow value to be aligned on 8 byte boundary
 *
 * name_len is the length of the name string including the null terminator
 * so it must be >= 1
 */
#define NVP_SIZE_CALC(name_len, data_len) \
        (NV_ALIGN((sizeof (nvpair_t)) + name_len) + NV_ALIGN(data_len))

static int i_get_value_size(data_type_t type, const void *data, uint_t nelem);
static int nvlist_add_common(nvlist_t *nvl, const char *name, data_type_t type,
    uint_t nelem, const void *data);

#define NV_STAT_EMBEDDED        0x1
#define EMBEDDED_NVL(nvp)       ((nvlist_t *)(void *)NVP_VALUE(nvp))
#define EMBEDDED_NVL_ARRAY(nvp) ((nvlist_t **)(void *)NVP_VALUE(nvp))

#define NVP_VALOFF(nvp) (NV_ALIGN(sizeof (nvpair_t) + (nvp)->nvp_name_sz))
#define NVPAIR2I_NVP(nvp) \
        ((i_nvp_t *)((size_t)(nvp) - offsetof(i_nvp_t, nvi_nvp)))


int
nv_alloc_init(nv_alloc_t *nva, const nv_alloc_ops_t *nvo, /* args */ ...)
{
        va_list valist;
        int err = 0;

        nva->nva_ops = nvo;
        nva->nva_arg = NULL;

        va_start(valist, nvo);
        if (nva->nva_ops->nv_ao_init != NULL)
                err = nva->nva_ops->nv_ao_init(nva, valist);
        va_end(valist);

        return (err);
}
```
______***unchanged_portion_omitted_***

****
2723 Thu Feb 25 15:39:35 2016
new/usr/src/head/iso/stddef_iso.h
3373 gcc >= 4.5 concerns about offsetof()
Portions contributed by: Igor Pashev <pashev.igor@gmail.com>
****
______unchanged_portion_omitted_

```
#endif /* end of namespace std */

#if __GNUC__ > 4 || (__GNUC__ == 4 && __GNUC_MINOR__ >= 5)
#define offsetof(s, m) __builtin_offsetof(s, m)
#else
#endif /* ! codereview */
#if __cplusplus >= 199711L
#define offsetof(s, m)  (std::size_t)(&(((s *)0)->m))
#else
#define offsetof(s, m)  (size_t)(&(((s *)0)->m))
#endif
#endif  /* GNUC, etc. */
#endif /* ! codereview */

#ifdef  __cplusplus
}
#endif

#endif  /* _ISO_STDDEF_ISO_H */
```

```
**********************************************************
    3536 Thu Feb 25 15:39:36 2016
new/usr/src/lib/libumem/common/misc.h
2976 remove useless offsetof() macros
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _MISC_H
#define _MISC_H

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/types.h>
#include <sys/time.h>
#include <thread.h>
#include <pthread.h>
#include <stdarg.h>
#include <stddef.h>
#endif /* ! codereview */

#ifdef  __cplusplus
extern "C" {
#endif

extern uint_t umem_abort;               /* abort when errors occur */
extern uint_t umem_output;              /* output error messages to stderr */
extern caddr_t umem_min_stack;          /* max stack address for audit log */
extern caddr_t umem_max_stack;          /* min stack address for audit log */

/*
 * various utility functions
 * These are globally implemented.
 */

#undef  offsetof
#define offsetof(s, m)  ((size_t)(&(((s *)0)->m)))

/*
 * a safe printf  -- do not use for error messages.
 */
void debug_printf(const char *format, ...);
```

```
/*
 * adds a message to the log without writing it out.
 */
void log_message(const char *format, ...);

/*
 * returns the index of the (high/low) bit + 1
 */
int highbit(ulong_t);
int lowbit(ulong_t);
#pragma no_side_effect(highbit, lowbit)

/*
 * Converts a hrtime_t to a timestruc_t
 */
void hrt2ts(hrtime_t hrt, timestruc_t *tsp);

/*
 * tries to print out the symbol and offset of a pointer using umem_error_info
 */
int print_sym(void *pointer);

/*
 * Information about the current error.  Can be called multiple times, should
 * be followed eventually with a call to umem_err or umem_err_recoverable.
 */
void umem_printf(const char *format, ...);
void umem_vprintf(const char *format, va_list);

void umem_printf_warn(void *ignored, const char *format, ...);

void umem_error_enter(const char *);

/*
 * prints error message and stack trace, then aborts.  Cannot return.
 */
void umem_panic(const char *format, ...) __NORETURN;
#pragma does_not_return(umem_panic)
#pragma rarely_called(umem_panic)

/*
 * like umem_err, but only aborts if umem_abort > 0
 */
void umem_err_recoverable(const char *format, ...);

/*
 * We define our own assertion handling since libc’s assert() calls malloc()
 */
#ifdef NDEBUG
#define ASSERT(assertion) (void)0
#else
#define ASSERT(assertion) (void)((assertion) || \
    __umem_assert_failed(#assertion, __FILE__, __LINE__))
#endif

int __umem_assert_failed(const char *assertion, const char *file, int line);
#pragma does_not_return(__umem_assert_failed)
#pragma rarely_called(__umem_assert_failed)
/*
 * These have architecture-specific implementations.
 */

/*
 * Returns the current function’s frame pointer.
 */
extern void *getfp(void);
```

```
/*
 * puts a pc-only stack trace of up to pcstack_limit frames into pcstack.
 * Returns the number of stacks written.
 *
 * if check_sighandler != 0, and we are in a signal context, calls
 * umem_err_recoverable.
 */
extern int getpcstack(uintptr_t *pcstack, int pcstack_limit,
    int check_sighandler);

#ifdef  __cplusplus
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   49098 Thu Feb 25 15:39:36 2016
new/usr/src/lib/lvm/libmeta/common/meta_statconcise.c
2976 remove useless offsetof() macros
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <meta.h>
#include <assert.h>
#include <ctype.h>
#include <mdiox.h>
#include <meta.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#endif /* ! codereview */
#include <strings.h>
#include <sys/lvm/md_mddb.h>
#include <sys/lvm/md_names.h>
#include <sys/lvm/md_crc.h>
#include <sys/lvm/md_convert.h>


/*
 * Design Notes:
 *
 * All of the code in this file supports the addition of metastat -c output
 * for the verbose option of metaimport.  Some of this code is also used by
 * the command metastat for concise output(cmd/lvm/util/metastat.c).
 * The code is designed to produce the same output as metastat -c does for a
 * given diskset--with a couple exceptions.
 * The primary differences between the output for the metastat -c command and
 * metastat output for metaimport -v are:
 *  - the set name is not printed next to each metadevice
 *  - top-level state information is not printed for some metadevices
 *  - the percent that a disk has completed resyncing is not listed
 * in metaimport -v.
 *
 *
 * The general layout of this file is as follows:
 *
```

```
 *  - report_metastat_info()
 *      This is the primary entry point for the functions in this file, with
 *      the exception of several functions that are also called from
 *      cmd/io/lvm/util/metastat.c
 *      report_metastat_info() calls functions to read in all the the
 *      Directory blocks and Record blocks and then process the information
 *      needed to print out the metadevice records in the same format as
 *      metastat -c.
 *
 *  - read_all_mdrecords()
 *      Reads in all the Directory blocks in the diskset and verifies their
 *      validity.  For each Directly block, it loops through all Directory
 *      Entries and for each one that contains a metadevice record calls
 *      read_md_record().  Because the output is designed to imitate the
 *      output of metastat -c, we ignore metadevice records for
 *      optimized resync, changelog, and translog.
 *
 *  - read_md_record()
 *      Reads in a Directory Entry and its associated Record block.  The
 *      revision information for the Record block is checked and it is
 *      determined whether or not it is a 64bit Record block or a 32bit record
 *      block.  For each valid Record block, it allocates an md_im_rec_t
 *      structure and calls extract_mduser_data().
 *
 *  - extract_mduser_data()
 *      Populates the md_im_rec_t data structure with information about the
 *      record's associated metadevice.  Also, the name of the metadevice is
 *      either copied from the NM namespace(if it exists there) or is generated
 *      from the record's un_self_id.
 *
 *  - process_toplevel_devices()
 *      For a given metadevice type, searchs through the md_im_rec_t **mdimpp,
 *      list of all metadevices in the set, to find all records of the
 *      specified type that do not have a parent and puts them on a temp list.
 *      The temp list is then iterated through and the associated processing
 *      function is called.
 *
 *  - process_(trans, hotspare, hotspare_pool, soft_part, mirror, stripe, raid)
 *      These functions are called by using the dfunc field in the mdimpp list.
 *      Each process function only understands its own type of metadevice. Once
 *      it processes the metadevice it was called for, it then loops through
 *      all of the underlying metadevices.  After printing the name of the
 *      underlying metadevice, it puts in on a list to be processed.  If the
 *      underlying device is a physical device, then print_physical_device is
 *      called.
 *      Once all information about the original metadevice is processed, it
 *      loops through the list of underlying metadevices and calls the
 *      appropriate function to process them.
 *
 *  - process_toplevel_softparts()
 *      To match the output for metastat -c, all top-level softpartions
 *      are printed out in groups based on their underlying metadevice--so that
 *      the underlying metadevice only needs to be processed once.
 *
 *  - meta_get_(sm_state, raid_col_state, stripe_state, hs_state)
 *      These functions are used to retrieve the metadevice state information.
 *      They are also used by the metastat concise routines in
 *      cmd/lvm/util/metastat.c.
 *
 */


/*
 * md_im_rec is a doubly linked list used to store the rb_data for each
 * directory entry that corresponds to a metadevice.
 * n_key: is set, if there is an associated entry in the NM namespace.
 * dfunc: is set to point to the function that processes the particular
 * metadevice associated with the record.
 * hs_record_id: is only set, if the metadevice is a hotspare.
 * un_self_id: is set for all other records. This is also used to generate
 * the name of the metadevice if there is no entry for the metadevice in
 * the NM namespace--n_key is not set.
 */
typedef struct md_im_rec {
        mdkey_t                 n_key; /* NM namespace key */
        struct md_im_rec        *next;
        struct md_im_rec        *prev;
        uint_t                  md_type;
        uint_t                  has_parent; /* either 0(no parent) or 1 */
        minor_t                 un_self_id;
        mddb_recid_t            hs_record_id; /* hotspare recid */
        char                    *n_name;  /* name of metadevice */
        void                    (*dfunc) ();
        ushort_t                record_len;
        /* pointer to the unit structure for the metadevice, e.g. rb_data[0] */
        void                    *record;
} md_im_rec_t;

/*
 * md_im_list is used to group toplevel metadevices by type and to group
 * the underlying devices for a particular metadevice.
 */
typedef struct md_im_list {
        struct md_im_list       *next;
        struct md_im_rec        *mdrec;
} md_im_list_t;


/*
 * MAXSIZEMDRECNAME is the value that has historically been used to allocate
 * space for the metadevice name
 */
#define MAXSIZEMDRECNAME        20
#define NAMEWIDTH               16
#define offsetof(s, m)  ((size_t)(&(((s *)0)->m)))
#define NOT_PHYSICAL_DEV        0
#define PHYSICAL_DEV            1


/*
 * strip_blacks()
 *
 * Strip blanks from string.  Used for size field in concise output.
 */
static char *
strip_blanks(char *s)
{
        char *p;

        for (p = s; *p; ) {
                if (*p == ' ') {
                        char *t;
                        for (t = p; *t; t++) {
                                *t = *(t + 1);
                        }
                } else {
                        p++;
                }
        }

        return (s);
}
```
______***unchanged_portion_omitted***_

```
**********************************************************
   26566 Thu Feb 25 15:39:37 2016
new/usr/src/uts/common/avs/ns/rdc/rdc.c
2976 remove useless offsetof() macros
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#define _RDC_
#include <sys/types.h>
#include <sys/ksynch.h>
#include <sys/kmem.h>
#include <sys/errno.h>
#include <sys/conf.h>
#include <sys/cmn_err.h>
#include <sys/modctl.h>
#include <sys/cred.h>
#include <sys/ddi.h>
#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/unistat/spcs_s.h>
#include <sys/unistat/spcs_s_k.h>
#include <sys/unistat/spcs_errors.h>

#include <sys/nsc_thread.h>
#ifdef DS_DDICT
#include "../contract.h"
#endif
#include <sys/nsctl/nsctl.h>
#include <sys/nsctl/nsvers.h>

#include <sys/sdt.h>            /* dtrace is S10 or later */

#include "rdc.h"
#include "rdc_io.h"
#include "rdc_bitmap.h"
#include "rdc_ioctl.h"
#include "rdcsrv.h"
#include "rdc_diskq.h"

#define DIDINIT         0x01
#define DIDNODES        0x02
#define DIDCONFIG       0x04
```

```
static int rdcopen(dev_t *devp, int flag, int otyp, cred_t *crp);
static int rdcclose(dev_t dev, int flag, int otyp, cred_t *crp);
static int rdcprint(dev_t dev, char *str);
static int rdcioctl(dev_t dev, int cmd, intptr_t arg, int mode, cred_t *crp,
        int *rvp);
static int rdcattach(dev_info_t *dip, ddi_attach_cmd_t cmd);
static int rdcdetach(dev_info_t *dip, ddi_detach_cmd_t cmd);
static int rdcgetinfo(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg,
        void **result);
#ifdef  DEBUG
static int rdc_clrkstat(void *);
#endif

/*
 * kstat interface
 */
static kstat_t *sndr_kstats;

int sndr_info_stats_update(kstat_t *ksp, int rw);

static sndr_m_stats_t sndr_info_stats = {
        {RDC_MKSTAT_MAXSETS,                    KSTAT_DATA_ULONG},
        {RDC_MKSTAT_MAXFBAS,                    KSTAT_DATA_ULONG},
        {RDC_MKSTAT_RPC_TIMEOUT,                KSTAT_DATA_ULONG},
        {RDC_MKSTAT_HEALTH_THRES,               KSTAT_DATA_ULONG},
        {RDC_MKSTAT_BITMAP_WRITES,              KSTAT_DATA_ULONG},
        {RDC_MKSTAT_CLNT_COTS_CALLS,            KSTAT_DATA_ULONG},
        {RDC_MKSTAT_CLNT_CLTS_CALLS,            KSTAT_DATA_ULONG},
        {RDC_MKSTAT_SVC_COTS_CALLS,             KSTAT_DATA_ULONG},
        {RDC_MKSTAT_SVC_CLTS_CALLS,             KSTAT_DATA_ULONG},
        {RDC_MKSTAT_BITMAP_REF_DELAY,           KSTAT_DATA_ULONG}
};

int rdc_info_stats_update(kstat_t *ksp, int rw);

static rdc_info_stats_t rdc_info_stats = {
        {RDC_IKSTAT_FLAGS,              KSTAT_DATA_ULONG},
        {RDC_IKSTAT_SYNCFLAGS,          KSTAT_DATA_ULONG},
        {RDC_IKSTAT_BMPFLAGS,           KSTAT_DATA_ULONG},
        {RDC_IKSTAT_SYNCPOS,            KSTAT_DATA_ULONG},
        {RDC_IKSTAT_VOLSIZE,            KSTAT_DATA_ULONG},
        {RDC_IKSTAT_BITSSET,            KSTAT_DATA_ULONG},
        {RDC_IKSTAT_AUTOSYNC,           KSTAT_DATA_ULONG},
        {RDC_IKSTAT_MAXQFBAS,           KSTAT_DATA_ULONG},
        {RDC_IKSTAT_MAXQITEMS,          KSTAT_DATA_ULONG},
        {RDC_IKSTAT_FILE,               KSTAT_DATA_STRING},
        {RDC_IKSTAT_SECFILE,            KSTAT_DATA_STRING},
        {RDC_IKSTAT_BITMAP,             KSTAT_DATA_STRING},
        {RDC_IKSTAT_PRIMARY_HOST,       KSTAT_DATA_STRING},
        {RDC_IKSTAT_SECONDARY_HOST,     KSTAT_DATA_STRING},
        {RDC_IKSTAT_TYPE_FLAG,          KSTAT_DATA_ULONG},
        {RDC_IKSTAT_BMP_SIZE,           KSTAT_DATA_ULONG},
        {RDC_IKSTAT_DISK_STATUS,        KSTAT_DATA_ULONG},
        {RDC_IKSTAT_IF_DOWN,            KSTAT_DATA_ULONG},
        {RDC_IKSTAT_IF_RPC_VERSION,     KSTAT_DATA_ULONG},
        {RDC_IKSTAT_ASYNC_BLOCK_HWM,    KSTAT_DATA_ULONG},
        {RDC_IKSTAT_ASYNC_ITEM_HWM,     KSTAT_DATA_ULONG},
        {RDC_IKSTAT_ASYNC_THROTTLE_DELAY,       KSTAT_DATA_ULONG},
        {RDC_IKSTAT_ASYNC_ITEMS,        KSTAT_DATA_ULONG},
        {RDC_IKSTAT_ASYNC_BLOCKS,       KSTAT_DATA_ULONG},
        {RDC_IKSTAT_QUEUE_TYPE,         KSTAT_DATA_CHAR}
};

static struct cb_ops rdc_cb_ops = {
        rdcopen,
        rdcclose,
        nulldev,                /* no strategy */
        rdcprint,
        nodev,                  /* no dump */
        nodev,                  /* no read */
        nodev,                  /* no write */
        rdcioctl,
        nodev,                  /* no devmap */
        nodev,                  /* no mmap */
        nodev,                  /* no segmap */
        nochpoll,
        ddi_prop_op,
        NULL,                   /* not STREAMS */
        D_NEW | D_MP | D_64BIT,
        CB_REV,
        nodev,                  /* no aread */
        nodev,                  /* no awrite */
};

static struct dev_ops rdc_ops = {
        DEVO_REV,
        0,
        rdcgetinfo,
        nulldev,                /* identify */
        nulldev,                /* probe */
        rdcattach,
        rdcdetach,
        nodev,                  /* no reset */
        &rdc_cb_ops,
        (struct bus_ops *)NULL
};

static struct modldrv rdc_ldrv = {
        &mod_driverops,
        "nws:Remote Mirror:" ISS_VERSION_STR,
        &rdc_ops
};

static struct modlinkage rdc_modlinkage = {
        MODREV_1,
        &rdc_ldrv,
        NULL
};

const   int sndr_major_rev = ISS_VERSION_MAJ;
const   int sndr_minor_rev = ISS_VERSION_MIN;
const   int sndr_micro_rev = ISS_VERSION_MIC;
const   int sndr_baseline_rev = ISS_VERSION_NUM;
static  char sndr_version[16];

static void *rdc_dip;

extern int _rdc_init_dev();
extern void _rdc_deinit_dev();
extern void rdc_link_down_free();

int rdc_bitmap_mode;
int rdc_auto_sync;
int rdc_max_sets;
extern int rdc_health_thres;

kmutex_t rdc_sync_mutex;
rdc_sync_event_t rdc_sync_event;
clock_t rdc_sync_event_timeout;

static void
rdc_sync_event_init()
```

```
{
        mutex_init(&rdc_sync_mutex, NULL, MUTEX_DRIVER, NULL);
        mutex_init(&rdc_sync_event.mutex, NULL, MUTEX_DRIVER, NULL);
        cv_init(&rdc_sync_event.cv, NULL, CV_DRIVER, NULL);
        cv_init(&rdc_sync_event.done_cv, NULL, CV_DRIVER, NULL);
        rdc_sync_event.master[0] = 0;
        rdc_sync_event.lbolt = (clock_t)0;
        rdc_sync_event_timeout = RDC_SYNC_EVENT_TIMEOUT;
}


static void
rdc_sync_event_destroy()
{
        mutex_destroy(&rdc_sync_mutex);
        mutex_destroy(&rdc_sync_event.mutex);
        cv_destroy(&rdc_sync_event.cv);
        cv_destroy(&rdc_sync_event.done_cv);
}



int
_init(void)
{
        return (mod_install(&rdc_modlinkage));
}

int
_fini(void)
{
        return (mod_remove(&rdc_modlinkage));
}

int
_info(struct modinfo *modinfop)
{
        return (mod_info(&rdc_modlinkage, modinfop));
}

static int
rdcattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        intptr_t flags;
        int instance;
        int i;

        /*CONSTCOND*/
        ASSERT(sizeof (u_longlong_t) == 8);

        if (cmd != DDI_ATTACH)
                return (DDI_FAILURE);

        (void) strncpy(sndr_version, _VERSION_, sizeof (sndr_version));

        instance = ddi_get_instance(dip);
        rdc_dip = dip;

        flags = 0;

        rdc_sync_event_init();

        /*
         * rdc_max_sets must be set before calling _rdc_load().
         */
```

```
	rdc_max_sets = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
	    DDI_PROP_DONTPASS | DDI_PROP_NOTPROM, "rdc_max_sets", 64);

	if (_rdc_init_dev()) {
		cmn_err(CE_WARN, "!rdc: _rdc_init_dev failed");
		goto out;
	}
	flags |= DIDINIT;

	if (_rdc_load() != 0) {
		cmn_err(CE_WARN, "!rdc: _rdc_load failed");
		goto out;
	}

	if (_rdc_configure()) {
		cmn_err(CE_WARN, "!rdc: _rdc_configure failed");
		goto out;
	}
	flags |= DIDCONFIG;

	if (ddi_create_minor_node(dip, "rdc", S_IFCHR, instance, DDI_PSEUDO, 0)
	    != DDI_SUCCESS) {
		cmn_err(CE_WARN, "!rdc: could not create node.");
		goto out;
	}
	flags |= DIDNODES;

	rdc_bitmap_mode = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
	    DDI_PROP_DONTPASS | DDI_PROP_NOTPROM,
	    "rdc_bitmap_mode", 0);

	switch (rdc_bitmap_mode) {
	case RDC_BMP_AUTO:		/* 0 */
		break;
	case RDC_BMP_ALWAYS:		/* 1 */
		break;
	case RDC_BMP_NEVER:		/* 2 */
		cmn_err(CE_NOTE, "!SNDR bitmap mode override");
		cmn_err(CE_CONT,
		    "!SNDR: bitmaps will only be written on shutdown\n");
		break;
	default:			/* unknown */
		cmn_err(CE_NOTE,
		    "!SNDR: unknown bitmap mode %d - autodetecting mode",
		    rdc_bitmap_mode);
		rdc_bitmap_mode = RDC_BMP_AUTO;
		break;
	}

	rdc_bitmap_init();

	rdc_auto_sync = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
	    DDI_PROP_DONTPASS | DDI_PROP_NOTPROM,
	    "rdc_auto_sync", 0);

	i = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
	    DDI_PROP_DONTPASS | DDI_PROP_NOTPROM,
	    "rdc_health_thres", RDC_HEALTH_THRESHOLD);
	if (i >= RDC_MIN_HEALTH_THRES)
		rdc_health_thres = i;
	else
		cmn_err(CE_WARN, "!value rdc_heath_thres from rdc.conf ignored "
		    "as it is smaller than the min value of %d",
		    RDC_MIN_HEALTH_THRES);

	ddi_set_driver_private(dip, (caddr_t)flags);
```

```
	ddi_report_dev(dip);

	sndr_kstats = kstat_create(RDC_KSTAT_MODULE, 0,
	    RDC_KSTAT_MINFO, RDC_KSTAT_CLASS, KSTAT_TYPE_NAMED,
	    sizeof (sndr_m_stats_t) / sizeof (kstat_named_t),
	    KSTAT_FLAG_VIRTUAL);

	if (sndr_kstats) {
		sndr_kstats->ks_data = &sndr_info_stats;
		sndr_kstats->ks_update = sndr_info_stats_update;
		sndr_kstats->ks_private = &rdc_k_info[0];
		kstat_install(sndr_kstats);
	} else
			cmn_err(CE_WARN, "!SNDR: module kstats failed");

	return (DDI_SUCCESS);

out:
	DTRACE_PROBE(rdc_attach_failed);
	ddi_set_driver_private(dip, (caddr_t)flags);
	(void) rdcdetach(dip, DDI_DETACH);
	return (DDI_FAILURE);
}

static int
rdcdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
	rdc_k_info_t *krdc;
	rdc_u_info_t *urdc;
	int rdcd;
	intptr_t flags;


	if (cmd != DDI_DETACH) {
		DTRACE_PROBE(rdc_detach_unknown_cmd);
		return (DDI_FAILURE);
	}

	if (rdc_k_info == NULL || rdc_u_info == NULL)
		goto cleanup;

	mutex_enter(&rdc_conf_lock);

	for (rdcd = 0; rdcd < rdc_max_sets; rdcd++) {
		krdc = &rdc_k_info[rdcd];
		urdc = &rdc_u_info[rdcd];

		if (IS_ENABLED(urdc) || krdc->devices) {
#ifdef DEBUG
			cmn_err(CE_WARN,
			    "!rdc: cannot detach, rdcd %d still in use", rdcd);
#endif
			mutex_exit(&rdc_conf_lock);
			DTRACE_PROBE(rdc_detach_err_busy);
			return (DDI_FAILURE);
		}
	}

	mutex_exit(&rdc_conf_lock);

cleanup:
	flags = (intptr_t)ddi_get_driver_private(dip);

	if (flags & DIDNODES)
		ddi_remove_minor_node(dip, NULL);
```

```c
	if (sndr_kstats) {
		kstat_delete(sndr_kstats);
	}
	if (flags & DIDINIT)
		_rdc_deinit_dev();

	if (flags & DIDCONFIG) {
		(void) _rdc_deconfigure();
		(void) _rdc_unload();
		rdcsrv_unload();
	}

	rdc_sync_event_destroy();
	rdc_link_down_free();

	rdc_dip = NULL;
	return (DDI_SUCCESS);
}

/* ARGSUSED */
static int
rdcgetinfo(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
{
	int rc = DDI_FAILURE;

	switch (infocmd) {

	case DDI_INFO_DEVT2DEVINFO:
		*result = rdc_dip;
		rc = DDI_SUCCESS;
		break;

	case DDI_INFO_DEVT2INSTANCE:
		/* We only have a single instance */
		*result = 0;
		rc = DDI_SUCCESS;
		break;

	default:
		break;
	}

	return (rc);
}


/* ARGSUSED */

static int
rdcopen(dev_t *devp, int flag, int otyp, cred_t *crp)
{
	return (0);
}


/* ARGSUSED */

static int
rdcclose(dev_t dev, int flag, int otyp, cred_t *crp)
{
	return (0);
}

/* ARGSUSED */

static int
```

```c
rdcprint(dev_t dev, char *str)
{
	int instance = 0;

	cmn_err(CE_WARN, "!rdc%d: %s", instance, str);
	return (0);
}


static int
convert_ioctl_args(int cmd, intptr_t arg, int mode, _rdc_ioctl_t *args)
{
	_rdc_ioctl32_t args32;

	if (ddi_copyin((void *)arg, &args32, sizeof (_rdc_ioctl32_t), mode))
		return (EFAULT);

	bzero((void *)args, sizeof (_rdc_ioctl_t));

	switch (cmd) {
	case RDC_CONFIG:
		args->arg0 = (uint32_t)args32.arg0; /* _rdc_config_t * */
		args->arg1 = (uint32_t)args32.arg1; /* pointer */
		args->arg2 = (uint32_t)args32.arg2; /* size */
		args->ustatus = (spcs_s_info_t)args32.ustatus;
		break;

	case RDC_STATUS:
		args->arg0 = (uint32_t)args32.arg0; /* pointer */
		args->ustatus = (spcs_s_info_t)args32.ustatus;
		break;

	case RDC_ENABLE_SVR:
		args->arg0 = (uint32_t)args32.arg0; /* _rdc_svc_args *  */
		break;

	case RDC_VERSION:
		args->arg0 = (uint32_t)args32.arg0; /* _rdc_version_t *  */
		args->ustatus = (spcs_s_info_t)args32.ustatus;
		break;

	case RDC_SYNC_EVENT:
		args->arg0 = (uint32_t)args32.arg0; /* char *  */
		args->arg1 = (uint32_t)args32.arg1; /* char *  */
		args->ustatus = (spcs_s_info_t)args32.ustatus;
		break;

	case RDC_LINK_DOWN:
		args->arg0 = (uint32_t)args32.arg0; /* char *  */
		args->ustatus = (spcs_s_info_t)args32.ustatus;
		break;
	case RDC_POOL_CREATE:
		args->arg0 = (uint32_t)args32.arg0; /* svcpool_args *  */
		break;
	case RDC_POOL_WAIT:
		args->arg0 = (uint32_t)args32.arg0; /* int */
		break;
	case RDC_POOL_RUN:
		args->arg0 = (uint32_t)args32.arg0; /* int */
		break;

	default:
		return (EINVAL);
	}

	return (0);
}


/*
 * Yet another standard thing that is not standard ...
 */
#ifndef offsetof
#define offsetof(s, m)  ((size_t)(&((s *)0)->m))
#endif

/*
 * Build a 32bit rdc_set structure and copyout to the user level.
 */
int
rdc_status_copy32(const void *arg, void *usetp, size_t size, int mode)
{
        rdc_u_info_t *urdc = (rdc_u_info_t *)arg;
        struct rdc_set32 set32;
        size_t tailsize;
#ifdef DEBUG
        size_t tailsize32;
#endif

        bzero(&set32, sizeof (set32));

        tailsize = sizeof (struct rdc_addr32) -
            offsetof(struct rdc_addr32, intf);

        /* primary address structure, avoiding netbuf */
        bcopy(&urdc->primary.intf[0], &set32.primary.intf[0], tailsize);

        /* secondary address structure, avoiding netbuf */
        bcopy(&urdc->secondary.intf[0], &set32.secondary.intf[0], tailsize);

        /*
         * the rest, avoiding netconfig
         * note: the tail must be the same size in both structures
         */
        tailsize = sizeof (struct rdc_set) - offsetof(struct rdc_set, flags);
#ifdef DEBUG
        /*
         * ASSERT is calling for debug reason, and tailsize32 is only declared
         * for ASSERT, put them under debug to avoid lint warning.
         */
        tailsize32 = sizeof (struct rdc_set32) -
            offsetof(struct rdc_set32, flags);
        ASSERT(tailsize == tailsize32);
#endif

        bcopy(&urdc->flags, &set32.flags, tailsize);

        /* copyout to user level */
        return (ddi_copyout(&set32, usetp, size, mode));
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
  160986 Thu Feb 25 15:39:37 2016
new/usr/src/uts/common/avs/ns/rdc/rdc_io.c
2976 remove useless offsetof() macros
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/types.h>
#include <sys/ksynch.h>
#include <sys/cmn_err.h>
#include <sys/kmem.h>
#include <sys/conf.h>
#include <sys/errno.h>
#include <sys/sysmacros.h>
#endif /* ! codereview */

#ifdef _SunOS_5_6
/*
 * on 2.6 both dki_lock.h and rpc/types.h define bool_t so we
 * define enum_t here as it is all we need from rpc/types.h
 * anyway and make it look like we included it. Yuck.
 */
#define _RPC_TYPES_H
typedef int enum_t;
#else
#ifndef DS_DDICT
#include <rpc/types.h>
#endif
#endif /* _SunOS_5_6 */

#include <sys/ddi.h>

#include <sys/nsc_thread.h>
#include <sys/nsctl/nsctl.h>

#include <sys/sdt.h>            /* dtrace is S10 or later */

#include "rdc_io.h"
#include "rdc_bitmap.h"
#include "rdc_update.h"
#include "rdc_ioctl.h"
#include "rdcsrv.h"
#include "rdc_diskq.h"
```

```
#include <sys/unistat/spcs_s.h>
#include <sys/unistat/spcs_s_k.h>
#include <sys/unistat/spcs_errors.h>

volatile int net_exit;
nsc_size_t MAX_RDC_FBAS;

#ifdef DEBUG
int RDC_MAX_SYNC_THREADS = 8;
int rdc_maxthreads_last = 8;
#endif

kmutex_t rdc_ping_lock;         /* Ping lock */
static kmutex_t net_blk_lock;

/*
 * rdc_conf_lock is used as a global device configuration lock.
 * It is also used by enable/resume and disable/suspend code to ensure that
 * the transition of an rdc set between configured and unconfigured is
 * atomic.
 *
 * krdc->group->lock is used to protect state changes of a configured rdc
 * set (e.g. changes to urdc->flags), such as enabled to disabled and vice
 * versa.
 *
 * rdc_many_lock is also used to protect changes in group membership. A group
 * linked list cannot change while this lock is held. The many list and the
 * multi-hop list are both protected by rdc_many_lock.
 */
kmutex_t rdc_conf_lock;
kmutex_t rdc_many_lock;                 /* Many/multi-list lock */

static kmutex_t rdc_net_hnd_id_lock;    /* Network handle id lock */
int rdc_debug = 0;
int rdc_debug_sleep = 0;

static int rdc_net_hnd_id = 1;

extern kmutex_t rdc_clnt_lock;

static void rdc_ditemsfree(rdc_net_dataset_t *);
void rdc_clnt_destroy(void);

rdc_k_info_t *rdc_k_info;
rdc_u_info_t *rdc_u_info;

unsigned long rdc_async_timeout;

nsc_size_t rdc_maxthres_queue = RDC_MAXTHRES_QUEUE;
int rdc_max_qitems = RDC_MAX_QITEMS;
int rdc_asyncthr = RDC_ASYNCTHR;
static nsc_svc_t *rdc_volume_update;
static int rdc_prealloc_handle = 1;

extern int _rdc_rsrv_diskq(rdc_group_t *group);
extern void _rdc_rlse_diskq(rdc_group_t *group);

/*
 * Forward declare all statics that are used before defined
 * to enforce parameter checking
 *
 * Some (if not all) of these could be removed if the code were reordered
 */

static void rdc_volume_update_svc(intptr_t);
```

```c
static void halt_sync(rdc_k_info_t *krdc);
void rdc_kstat_create(int index);
void rdc_kstat_delete(int index);
static int rdc_checkforbitmap(int, nsc_off_t);
static int rdc_installbitmap(int, void *, int, nsc_off_t, int, int *, int);
static rdc_group_t *rdc_newgroup();

int rdc_enable_diskq(rdc_k_info_t *krdc);
void rdc_close_diskq(rdc_group_t *group);
int rdc_suspend_diskq(rdc_k_info_t *krdc);
int rdc_resume_diskq(rdc_k_info_t *krdc);
void rdc_init_diskq_header(rdc_group_t *grp, dqheader *header);
void rdc_fail_diskq(rdc_k_info_t *krdc, int wait, int dolog);
void rdc_unfail_diskq(rdc_k_info_t *krdc);
void rdc_unintercept_diskq(rdc_group_t *grp);
int rdc_stamp_diskq(rdc_k_info_t *krdc, int rsrvd, int flags);
void rdc_qfiller_thr(rdc_k_info_t *krdc);

nstset_t *_rdc_ioset;
nstset_t *_rdc_flset;

/*
 * RDC threadset tunables
 */
int rdc_threads = 64;           /* default number of threads */
int rdc_threads_inc = 8;        /* increment for changing the size of the set */

/*
 * Private threadset manipulation variables
 */
static int rdc_threads_hysteresis = 2;
                                /* hysteresis for threadset resizing */
static int rdc_sets_active;     /* number of sets currently enabled */

#ifdef DEBUG
kmutex_t rdc_cntlock;
#endif

/*
 * rdc_thread_deconfigure - rdc is being deconfigured, stop any
 * thread activity.
 *
 * Inherently single-threaded by the Solaris module unloading code.
 */
static void
rdc_thread_deconfigure(void)
{
        nst_destroy(_rdc_ioset);
        _rdc_ioset = NULL;

        nst_destroy(_rdc_flset);
        _rdc_flset = NULL;

        nst_destroy(sync_info.rdc_syncset);
        sync_info.rdc_syncset = NULL;
}

/*
 * rdc_thread_configure - rdc is being configured, initialize the
 * threads we need for flushing aync volumes.
 *
 * Must be called with rdc_conf_lock held.
 */
static int
rdc_thread_configure(void)
{
```

```c
        ASSERT(MUTEX_HELD(&rdc_conf_lock));

        if ((_rdc_ioset = nst_init("rdc_thr", rdc_threads)) == NULL)
                return (EINVAL);

        if ((_rdc_flset = nst_init("rdc_flushthr", 2)) == NULL)
                return (EINVAL);

        if ((sync_info.rdc_syncset =
            nst_init("rdc_syncthr", RDC_MAX_SYNC_THREADS)) == NULL)
                return (EINVAL);

        return (0);
}


/*
 * rdc_thread_tune - called to tune the size of the rdc threadset.
 *
 * Called from the config code when an rdc_set has been enabled or disabled.
 * 'sets' is the increment to the number of active rdc_sets.
 *
 * Must be called with rdc_conf_lock held.
 */
static void
rdc_thread_tune(int sets)
{
        int incr = (sets > 0) ? 1 : -1;
        int change = 0;
        int nthreads;

        ASSERT(MUTEX_HELD(&rdc_conf_lock));

        if (sets < 0)
                sets = -sets;

        while (sets--) {
                nthreads = nst_nthread(_rdc_ioset);
                rdc_sets_active += incr;

                if (rdc_sets_active >= nthreads)
                        change += nst_add_thread(_rdc_ioset, rdc_threads_inc);
                else if ((rdc_sets_active <
                    (nthreads - (rdc_threads_inc + rdc_threads_hysteresis))) &&
                    ((nthreads - rdc_threads_inc) >= rdc_threads))
                        change -= nst_del_thread(_rdc_ioset, rdc_threads_inc);
        }

#ifdef DEBUG
        if (change) {
                cmn_err(CE_NOTE, "!rdc_thread_tune: "
                    "nsets %d, nthreads %d, nthreads change %d",
                    rdc_sets_active, nst_nthread(_rdc_ioset), change);
        }
#endif
}


/*
 * _rdc_unload() - cache is being unloaded,
 * deallocate any dual copy structures allocated during cache
 * loading.
 */
void
_rdc_unload(void)
{
	int i;
	rdc_k_info_t *krdc;

	if (rdc_volume_update) {
		(void) nsc_unregister_svc(rdc_volume_update);
		rdc_volume_update = NULL;
	}

	rdc_thread_deconfigure();

	if (rdc_k_info != NULL) {
		for (i = 0; i < rdc_max_sets; i++) {
			krdc = &rdc_k_info[i];
			mutex_destroy(&krdc->dc_sleep);
			mutex_destroy(&krdc->bmapmutex);
			mutex_destroy(&krdc->kstat_mutex);
			mutex_destroy(&krdc->bmp_kstat_mutex);
			mutex_destroy(&krdc->syncbitmutex);
			cv_destroy(&krdc->busycv);
			cv_destroy(&krdc->closingcv);
			cv_destroy(&krdc->haltcv);
			cv_destroy(&krdc->synccv);
		}
	}

	mutex_destroy(&sync_info.lock);
	mutex_destroy(&rdc_ping_lock);
	mutex_destroy(&net_blk_lock);
	mutex_destroy(&rdc_conf_lock);
	mutex_destroy(&rdc_many_lock);
	mutex_destroy(&rdc_net_hnd_id_lock);
	mutex_destroy(&rdc_clnt_lock);
#ifdef DEBUG
	mutex_destroy(&rdc_cntlock);
#endif
	net_exit = ATM_EXIT;

	if (rdc_k_info != NULL)
		kmem_free(rdc_k_info, sizeof (*rdc_k_info) * rdc_max_sets);
	if (rdc_u_info != NULL)
		kmem_free(rdc_u_info, sizeof (*rdc_u_info) * rdc_max_sets);
	rdc_k_info = NULL;
	rdc_u_info = NULL;
	rdc_max_sets = 0;
}


/*
 * _rdc_load() - rdc is being loaded, Allocate anything
 * that will be needed while the cache is loaded but doesn't really
 * depend on configuration parameters.
 *
 */
int
_rdc_load(void)
{
	int i;
	rdc_k_info_t *krdc;

	mutex_init(&rdc_ping_lock, NULL, MUTEX_DRIVER, NULL);
	mutex_init(&net_blk_lock, NULL, MUTEX_DRIVER, NULL);
	mutex_init(&rdc_conf_lock, NULL, MUTEX_DRIVER, NULL);
	mutex_init(&rdc_many_lock, NULL, MUTEX_DRIVER, NULL);
	mutex_init(&rdc_net_hnd_id_lock, NULL, MUTEX_DRIVER, NULL);
	mutex_init(&rdc_clnt_lock, NULL, MUTEX_DRIVER, NULL);
	mutex_init(&sync_info.lock, NULL, MUTEX_DRIVER, NULL);
```

```

#ifdef DEBUG
	mutex_init(&rdc_cntlock, NULL, MUTEX_DRIVER, NULL);
#endif

	if ((i = nsc_max_devices()) < rdc_max_sets)
		rdc_max_sets = i;
	/* following case for partial installs that may fail */
	if (!rdc_max_sets)
		rdc_max_sets = 1024;

	rdc_k_info = kmem_zalloc(sizeof (*rdc_k_info) * rdc_max_sets, KM_SLEEP);
	if (!rdc_k_info)
		return (ENOMEM);

	rdc_u_info = kmem_zalloc(sizeof (*rdc_u_info) * rdc_max_sets, KM_SLEEP);
	if (!rdc_u_info) {
		kmem_free(rdc_k_info, sizeof (*rdc_k_info) * rdc_max_sets);
		return (ENOMEM);
	}

	net_exit = ATM_NONE;
	for (i = 0; i < rdc_max_sets; i++) {
		krdc = &rdc_k_info[i];
		bzero(krdc, sizeof (*krdc));
		krdc->index = i;
		mutex_init(&krdc->dc_sleep, NULL, MUTEX_DRIVER, NULL);
		mutex_init(&krdc->bmapmutex, NULL, MUTEX_DRIVER, NULL);
		mutex_init(&krdc->kstat_mutex, NULL, MUTEX_DRIVER, NULL);
		mutex_init(&krdc->bmp_kstat_mutex, NULL, MUTEX_DRIVER, NULL);
		mutex_init(&krdc->syncbitmutex, NULL, MUTEX_DRIVER, NULL);
		cv_init(&krdc->busycv, NULL, CV_DRIVER, NULL);
		cv_init(&krdc->closingcv, NULL, CV_DRIVER, NULL);
		cv_init(&krdc->haltcv, NULL, CV_DRIVER, NULL);
		cv_init(&krdc->synccv, NULL, CV_DRIVER, NULL);
	}

	rdc_volume_update = nsc_register_svc("RDCVolumeUpdated",
	    rdc_volume_update_svc);

	return (0);
}

static void
rdc_u_init(rdc_u_info_t *urdc)
{
	const int index = (int)(urdc - &rdc_u_info[0]);

	if (urdc->secondary.addr.maxlen)
		free_rdc_netbuf(&urdc->secondary.addr);
	if (urdc->primary.addr.maxlen)
		free_rdc_netbuf(&urdc->primary.addr);

	bzero(urdc, sizeof (rdc_u_info_t));

	urdc->index = index;
	urdc->maxqfbas = rdc_maxthres_queue;
	urdc->maxqitems = rdc_max_qitems;
	urdc->asyncthr = rdc_asyncthr;
}

/*
 * _rdc_configure() - cache is being configured.
 *
 * Initialize dual copy structures
 */
```

```
int
_rdc_configure(void)
{
	int index;
	rdc_k_info_t *krdc;

	for (index = 0; index < rdc_max_sets; index++) {
		krdc = &rdc_k_info[index];

		krdc->remote_index = -1;
		krdc->dcio_bitmap = NULL;
		krdc->bitmap_ref = NULL;
		krdc->bitmap_size = 0;
		krdc->bitmap_write = 0;
		krdc->disk_status = 0;
		krdc->many_next = krdc;

		rdc_u_init(&rdc_u_info[index]);
	}

	rdc_async_timeout = 120 * HZ;   /* Seconds * HZ */
	MAX_RDC_FBAS = FBA_LEN(RDC_MAXDATA);
	if (net_exit != ATM_INIT) {
		net_exit = ATM_INIT;
		return (0);
	}
	return (0);
}

/*
 * _rdc_deconfigure - rdc is being deconfigured, shut down any
 * dual copy operations and return to an unconfigured state.
 */
void
_rdc_deconfigure(void)
{
	rdc_k_info_t *krdc;
	rdc_u_info_t *urdc;
	int index;

	for (index = 0; index < rdc_max_sets; index++) {
		krdc = &rdc_k_info[index];
		urdc = &rdc_u_info[index];

		krdc->remote_index = -1;
		krdc->dcio_bitmap = NULL;
		krdc->bitmap_ref = NULL;
		krdc->bitmap_size = 0;
		krdc->bitmap_write = 0;
		krdc->disk_status = 0;
		krdc->many_next = krdc;

		if (urdc->primary.addr.maxlen)
			free_rdc_netbuf(&(urdc->primary.addr));

		if (urdc->secondary.addr.maxlen)
			free_rdc_netbuf(&(urdc->secondary.addr));

		bzero(urdc, sizeof (rdc_u_info_t));
		urdc->index = index;
	}
	net_exit = ATM_EXIT;
	rdc_clnt_destroy();

}
```

```
/*
 * Lock primitives, containing checks that lock ordering isn't broken
 */
/*ARGSUSED*/
void
rdc_many_enter(rdc_k_info_t *krdc)
{
	ASSERT(!MUTEX_HELD(&krdc->bmapmutex));

	mutex_enter(&rdc_many_lock);
}

/* ARGSUSED */
void
rdc_many_exit(rdc_k_info_t *krdc)
{
	mutex_exit(&rdc_many_lock);
}

void
rdc_group_enter(rdc_k_info_t *krdc)
{
	ASSERT(!MUTEX_HELD(&rdc_many_lock));
	ASSERT(!MUTEX_HELD(&rdc_conf_lock));
	ASSERT(!MUTEX_HELD(&krdc->bmapmutex));

	mutex_enter(&krdc->group->lock);
}

void
rdc_group_exit(rdc_k_info_t *krdc)
{
	mutex_exit(&krdc->group->lock);
}

/*
 * Suspend and disable operations use this function to wait until it is safe
 * to do continue, without trashing data structures used by other ioctls.
 */
static void
wait_busy(rdc_k_info_t *krdc)
{
	ASSERT(MUTEX_HELD(&rdc_conf_lock));

	while (krdc->busy_count > 0)
		cv_wait(&krdc->busycv, &rdc_conf_lock);
}


/*
 * Other ioctls use this function to hold off disable and suspend.
 */
void
set_busy(rdc_k_info_t *krdc)
{
	ASSERT(MUTEX_HELD(&rdc_conf_lock));

	wait_busy(krdc);

	krdc->busy_count++;
}


/*
 * Other ioctls use this function to allow disable and suspend to continue.
 */
void
wakeup_busy(rdc_k_info_t *krdc)
{
        ASSERT(MUTEX_HELD(&rdc_conf_lock));

        if (krdc->busy_count <= 0)
                return;

        krdc->busy_count--;
        cv_broadcast(&krdc->busycv);
}


/*
 * Remove the rdc set from its group, and destroy the group if no longer in
 * use.
 */
static void
remove_from_group(rdc_k_info_t *krdc)
{
        rdc_k_info_t *p;
        rdc_group_t *group;

        ASSERT(MUTEX_HELD(&rdc_conf_lock));

        rdc_many_enter(krdc);
        group = krdc->group;

        group->count--;

        /*
         * lock queue while looking at thrnum
         */
        mutex_enter(&group->ra_queue.net_qlock);
        if ((group->rdc_thrnum == 0) && (group->count == 0)) {

                /*
                 * Assure the we've stopped and the flusher thread has not
                 * fallen back to sleep
                 */
                if (krdc->group->ra_queue.qfill_sleeping != RDC_QFILL_DEAD) {
                        group->ra_queue.qfflags |= RDC_QFILLSTOP;
                        while (krdc->group->ra_queue.qfflags & RDC_QFILLSTOP) {
                                if (krdc->group->ra_queue.qfill_sleeping ==
                                    RDC_QFILL_ASLEEP)
                                        cv_broadcast(&group->ra_queue.qfcv);
                                mutex_exit(&group->ra_queue.net_qlock);
                                delay(2);
                                mutex_enter(&group->ra_queue.net_qlock);
                        }
                }
                mutex_exit(&group->ra_queue.net_qlock);

                mutex_enter(&group->diskqmutex);
                rdc_close_diskq(group);
                mutex_exit(&group->diskqmutex);
                rdc_delgroup(group);
                rdc_many_exit(krdc);
                krdc->group = NULL;
                return;
        }
        mutex_exit(&group->ra_queue.net_qlock);
        /*
         * Always clear the group field.
         * no, you need it set in rdc_flush_memq().
```

```
         * to call rdc_group_log()
         * krdc->group = NULL;
         */

        /* Take this rdc structure off the group list */

        for (p = krdc->group_next; p->group_next != krdc; p = p->group_next)
        ;
        p->group_next = krdc->group_next;

        rdc_many_exit(krdc);
}


/*
 * Add the rdc set to its group, setting up a new group if it's the first one.
 */
static int
add_to_group(rdc_k_info_t *krdc, int options, int cmd)
{
        rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
        rdc_u_info_t *utmp;
        rdc_k_info_t *ktmp;
        int index;
        rdc_group_t *group;
        int rc = 0;
        nsthread_t *trc;

        ASSERT(MUTEX_HELD(&rdc_conf_lock));

        /*
         * Look for matching group name, primary host name and secondary
         * host name.
         */

        rdc_many_enter(krdc);
        for (index = 0; index < rdc_max_sets; index++) {
                utmp = &rdc_u_info[index];
                ktmp = &rdc_k_info[index];

                if (urdc->group_name[0] == 0)
                        break;

                if (!IS_CONFIGURED(ktmp))
                        continue;

                if (strncmp(utmp->group_name, urdc->group_name,
                    NSC_MAXPATH) != 0)
                        continue;
                if (strncmp(utmp->primary.intf, urdc->primary.intf,
                    MAX_RDC_HOST_SIZE) != 0) {
                        /* Same group name, different primary interface */
                        rdc_many_exit(krdc);
                        return (-1);
                }
                if (strncmp(utmp->secondary.intf, urdc->secondary.intf,
                    MAX_RDC_HOST_SIZE) != 0) {
                        /* Same group name, different secondary interface */
                        rdc_many_exit(krdc);
                        return (-1);
                }

                /* Group already exists, so add this set to the group */

                if (((options & RDC_OPT_ASYNC) == 0) &&
                    ((ktmp->type_flag & RDC_ASYNCMODE) != 0)) {
                        /* Must be same mode as existing group members */
                        rdc_many_exit(krdc);
                        return (-1);
                }
                if (((options & RDC_OPT_ASYNC) != 0) &&
                    ((ktmp->type_flag & RDC_ASYNCMODE) == 0)) {
                        /* Must be same mode as existing group members */
                        rdc_many_exit(krdc);
                        return (-1);
                }

                /* cannont reconfigure existing group into new queue this way */
                if ((cmd != RDC_CMD_RESUME) &&
                    !RDC_IS_DISKQ(ktmp->group) && urdc->disk_queue[0] != '\0') {
                        rdc_many_exit(krdc);
                        return (RDC_EQNOADD);
                }

                ktmp->group->count++;
                krdc->group = ktmp->group;
                krdc->group_next = ktmp->group_next;
                ktmp->group_next = krdc;

                urdc->autosync = utmp->autosync;        /* Same as rest */

                (void) strncpy(urdc->disk_queue, utmp->disk_queue, NSC_MAXPATH);

                rdc_many_exit(krdc);
                return (0);
        }

        /* This must be a new group */
        group = rdc_newgroup();
        krdc->group = group;
        krdc->group_next = krdc;
        urdc->autosync = -1;    /* Unknown */

        /*
         * Tune the thread set by one for each thread created
         */
        rdc_thread_tune(1);

        trc = nst_create(_rdc_ioset, rdc_qfiller_thr, (void *)krdc, NST_SLEEP);
        if (trc == NULL) {
                rc = -1;
                cmn_err(CE_NOTE, "!unable to create queue filler daemon");
                goto fail;
        }

        if (urdc->disk_queue[0] == '\0') {
                krdc->group->flags |= RDC_MEMQUE;
        } else {
                krdc->group->flags |= RDC_DISKQUE;

                /* XXX check here for resume or enable and act accordingly */

                if (cmd == RDC_CMD_RESUME) {
                        rc = rdc_resume_diskq(krdc);

                } else if (cmd == RDC_CMD_ENABLE) {
                        rc = rdc_enable_diskq(krdc);
                        if ((rc == RDC_EQNOADD) && (cmd != RDC_CMD_ENABLE)) {
                                cmn_err(CE_WARN, "!disk queue %s enable failed,"
                                    " enabling memory queue",
                                    urdc->disk_queue);
                                krdc->group->flags &= ~RDC_DISKQUE;
```

```
                                krdc->group->flags |= RDC_MEMQUE;
                                bzero(urdc->disk_queue, NSC_MAXPATH);
                        }
                }
        }
fail:
        rdc_many_exit(krdc);
        return (rc);
}


/*
 * Move the set to a new group if possible
 */
static int
change_group(rdc_k_info_t *krdc, int options)
{
        rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
        rdc_u_info_t *utmp;
        rdc_k_info_t *ktmp;
        rdc_k_info_t *next;
        char tmpq[NSC_MAXPATH];
        int index;
        int rc = -1;
        rdc_group_t *group, *old_group;
        nsthread_t *trc;

        ASSERT(MUTEX_HELD(&rdc_conf_lock));

        /*
         * Look for matching group name, primary host name and secondary
         * host name.
         */

        bzero(&tmpq, sizeof (tmpq));
        rdc_many_enter(krdc);

        old_group = krdc->group;
        next = krdc->group_next;

        if (RDC_IS_DISKQ(old_group)) { /* can't keep your own queue */
                (void) strncpy(tmpq, urdc->disk_queue, NSC_MAXPATH);
                bzero(urdc->disk_queue, sizeof (urdc->disk_queue));
        }
        for (index = 0; index < rdc_max_sets; index++) {
                utmp = &rdc_u_info[index];
                ktmp = &rdc_k_info[index];

                if (ktmp == krdc)
                        continue;

                if (urdc->group_name[0] == 0)
                        break;

                if (!IS_CONFIGURED(ktmp))
                        continue;

                if (strncmp(utmp->group_name, urdc->group_name,
                    NSC_MAXPATH) != 0)
                        continue;
                if (strncmp(utmp->primary.intf, urdc->primary.intf,
                    MAX_RDC_HOST_SIZE) != 0)
                        goto bad;
                if (strncmp(utmp->secondary.intf, urdc->secondary.intf,
                    MAX_RDC_HOST_SIZE) != 0)
                        goto bad;
```

```
		/* Group already exists, so add this set to the group */

		if (((options & RDC_OPT_ASYNC) == 0) &&
		    ((ktmp->type_flag & RDC_ASYNCMODE) != 0)) {
			/* Must be same mode as existing group members */
			goto bad;
		}
		if (((options & RDC_OPT_ASYNC) != 0) &&
		    ((ktmp->type_flag & RDC_ASYNCMODE) == 0)) {
			/* Must be same mode as existing group members */
			goto bad;
		}

		ktmp->group->count++;
		krdc->group = ktmp->group;
		krdc->group_next = ktmp->group_next;
		ktmp->group_next = krdc;
		bzero(urdc->disk_queue, sizeof (urdc->disk_queue));
		(void) strncpy(urdc->disk_queue, utmp->disk_queue, NSC_MAXPATH);

		goto good;
	}

	/* This must be a new group */
	group = rdc_newgroup();
	krdc->group = group;
	krdc->group_next = krdc;

	trc = nst_create(_rdc_ioset, rdc_qfiller_thr, (void *)krdc, NST_SLEEP);
	if (trc == NULL) {
		rc = -1;
		cmn_err(CE_NOTE, "!unable to create queue filler daemon");
		goto bad;
	}

	if (urdc->disk_queue[0] == 0) {
		krdc->group->flags |= RDC_MEMQUE;
	} else {
		krdc->group->flags |= RDC_DISKQUE;
		if ((rc = rdc_enable_diskq(krdc)) < 0)
			goto bad;
	}
good:
	if (options & RDC_OPT_ASYNC) {
		krdc->type_flag |= RDC_ASYNCMODE;
		rdc_set_flags(urdc, RDC_ASYNC);
	} else {
		krdc->type_flag &= ~RDC_ASYNCMODE;
		rdc_clr_flags(urdc, RDC_ASYNC);
	}

	old_group->count--;
	if (!old_group->rdc_writer && old_group->count == 0) {
		/* Group now empty, so destroy */
		if (RDC_IS_DISKQ(old_group)) {
			rdc_unintercept_diskq(old_group);
			mutex_enter(&old_group->diskqmutex);
			rdc_close_diskq(old_group);
			mutex_exit(&old_group->diskqmutex);
		}

		mutex_enter(&old_group->ra_queue.net_qlock);

		/*
		 * Assure the we’ve stopped and the flusher thread has not
```

```
		 * fallen back to sleep
		 */
		if (old_group->ra_queue.qfill_sleeping != RDC_QFILL_DEAD) {
			old_group->ra_queue.qfflags |= RDC_QFILLSTOP;
			while (old_group->ra_queue.qfflags & RDC_QFILLSTOP) {
				if (old_group->ra_queue.qfill_sleeping ==
				    RDC_QFILL_ASLEEP)
					cv_broadcast(&old_group->ra_queue.qfcv);
				mutex_exit(&old_group->ra_queue.net_qlock);
				delay(2);
				mutex_enter(&old_group->ra_queue.net_qlock);
			}
		}
		mutex_exit(&old_group->ra_queue.net_qlock);

		rdc_delgroup(old_group);
		rdc_many_exit(krdc);
		return (0);
	}

	/* Take this rdc structure off the old group list */

	for (ktmp = next; ktmp->group_next != krdc; ktmp = ktmp->group_next)
	;
	ktmp->group_next = next;

	rdc_many_exit(krdc);
	return (0);

bad:
	/* Leave existing group status alone */
	(void) strncpy(urdc->disk_queue, tmpq, NSC_MAXPATH);
	rdc_many_exit(krdc);
	return (rc);
}


/*
 * Set flags for an rdc set, setting the group flags as necessary.
 */
void
rdc_set_flags(rdc_u_info_t *urdc, int flags)
{
	rdc_k_info_t *krdc = &rdc_k_info[urdc->index];
	int vflags, sflags, bflags, ssflags;

	DTRACE_PROBE2(rdc_set_flags, int, krdc->index, int, flags);
	vflags = flags & RDC_VFLAGS;
	sflags = flags & RDC_SFLAGS;
	bflags = flags & RDC_BFLAGS;
	ssflags = flags & RDC_SYNC_STATE_FLAGS;

	if (vflags) {
		/* normal volume flags */
		ASSERT(MUTEX_HELD(&rdc_conf_lock) ||
		    MUTEX_HELD(&krdc->group->lock));
		if (ssflags)
			mutex_enter(&krdc->bmapmutex);

		urdc->flags |= vflags;

		if (ssflags)
			mutex_exit(&krdc->bmapmutex);
	}

	if (sflags) {
		/* Sync state flags that are protected by a different lock */
		ASSERT(MUTEX_HELD(&rdc_many_lock));
		urdc->sync_flags |= sflags;
	}

	if (bflags) {
		/* Bmap state flags that are protected by a different lock */
		ASSERT(MUTEX_HELD(&krdc->bmapmutex));
		urdc->bmap_flags |= bflags;
	}

}


/*
 * Clear flags for an rdc set, clearing the group flags as necessary.
 */
void
rdc_clr_flags(rdc_u_info_t *urdc, int flags)
{
	rdc_k_info_t *krdc = &rdc_k_info[urdc->index];
	int vflags, sflags, bflags;

	DTRACE_PROBE2(rdc_clr_flags, int, krdc->index, int, flags);
	vflags = flags & RDC_VFLAGS;
	sflags = flags & RDC_SFLAGS;
	bflags = flags & RDC_BFLAGS;

	if (vflags) {
		/* normal volume flags */
		ASSERT(MUTEX_HELD(&rdc_conf_lock) ||
		    MUTEX_HELD(&krdc->group->lock));
		urdc->flags &= ~vflags;

	}

	if (sflags) {
		/* Sync state flags that are protected by a different lock */
		ASSERT(MUTEX_HELD(&rdc_many_lock));
		urdc->sync_flags &= ~sflags;
	}

	if (bflags) {
		/* Bmap state flags that are protected by a different lock */
		ASSERT(MUTEX_HELD(&krdc->bmapmutex));
		urdc->bmap_flags &= ~bflags;
	}
}


/*
 * Get the flags for an rdc set.
 */
int
rdc_get_vflags(rdc_u_info_t *urdc)
{
	return (urdc->flags | urdc->sync_flags | urdc->bmap_flags);
}


/*
 * Initialise flags for an rdc set.
 */
static void
rdc_init_flags(rdc_u_info_t *urdc)
{
```

```
	urdc->flags = 0;
	urdc->mflags = 0;
	urdc->sync_flags = 0;
	urdc->bmap_flags = 0;
}


/*
 * Set flags for a many group.
 */
void
rdc_set_mflags(rdc_u_info_t *urdc, int flags)
{
	rdc_k_info_t *krdc = &rdc_k_info[urdc->index];
	rdc_k_info_t *this = krdc;

	ASSERT(!(flags & ~RDC_MFLAGS));

	if (flags == 0)
		return;

	ASSERT(MUTEX_HELD(&rdc_many_lock));

	rdc_set_flags(urdc, flags);	/* set flags on local urdc */

	urdc->mflags |= flags;
	for (krdc = krdc->many_next; krdc != this; krdc = krdc->many_next) {
		urdc = &rdc_u_info[krdc->index];
		if (!IS_ENABLED(urdc))
			continue;
		urdc->mflags |= flags;
	}
}


/*
 * Clear flags for a many group.
 */
void
rdc_clr_mflags(rdc_u_info_t *urdc, int flags)
{
	rdc_k_info_t *krdc = &rdc_k_info[urdc->index];
	rdc_k_info_t *this = krdc;
	rdc_u_info_t *utmp;

	ASSERT(!(flags & ~RDC_MFLAGS));

	if (flags == 0)
		return;

	ASSERT(MUTEX_HELD(&rdc_many_lock));

	rdc_clr_flags(urdc, flags);	/* clear flags on local urdc */

	/*
	 * We must maintain the mflags based on the set of flags for
	 * all the urdc’s that are chained up.
	 */

	/*
	 * First look through all the urdc’s and remove bits from
	 * the ’flags’ variable that are in use elsewhere.
	 */

	for (krdc = krdc->many_next; krdc != this; krdc = krdc->many_next) {
		utmp = &rdc_u_info[krdc->index];
```

```c
                if (!IS_ENABLED(utmp))
                        continue;
                flags &= ~(rdc_get_vflags(utmp) & RDC_MFLAGS);
                if (flags == 0)
                        break;
        }

        /*
         * Now clear flags as necessary.
         */

        if (flags != 0) {
                urdc->mflags &= ~flags;
                for (krdc = krdc->many_next; krdc != this;
                    krdc = krdc->many_next) {
                        utmp = &rdc_u_info[krdc->index];
                        if (!IS_ENABLED(utmp))
                                continue;
                        utmp->mflags &= ~flags;
                }
        }
}


int
rdc_get_mflags(rdc_u_info_t *urdc)
{
        return (urdc->mflags);
}


void
rdc_set_flags_log(rdc_u_info_t *urdc, int flags, char *why)
{
        DTRACE_PROBE2(rdc_set_flags_log, int, urdc->index, int, flags);

        rdc_set_flags(urdc, flags);

        if (why == NULL)
                return;

        if (flags & RDC_LOGGING)
                cmn_err(CE_NOTE, "!sndr: %s:%s entered logging mode: %s",
                    urdc->secondary.intf, urdc->secondary.file, why);
        if (flags & RDC_VOL_FAILED)
                cmn_err(CE_NOTE, "!sndr: %s:%s volume failed: %s",
                    urdc->secondary.intf, urdc->secondary.file, why);
        if (flags & RDC_BMP_FAILED)
                cmn_err(CE_NOTE, "!sndr: %s:%s bitmap failed: %s",
                    urdc->secondary.intf, urdc->secondary.file, why);
}
/*
 * rdc_lor(source, dest, len)
 * logically OR memory pointed to by source and dest, copying result into dest.
 */
void
rdc_lor(const uchar_t *source, uchar_t *dest, int len)
{
        int i;

        if (source == NULL)
                return;

        for (i = 0; i < len; i++)
                *dest++ |= *source++;
}
```

```c

static int
check_filesize(int index, spcs_s_info_t kstatus)
{
        uint64_t remote_size;
        char tmp1[16], tmp2[16];
        rdc_u_info_t *urdc = &rdc_u_info[index];
        int status;

        status = rdc_net_getsize(index, &remote_size);
        if (status) {
                (void) spcs_s_inttostring(status, tmp1, sizeof (tmp1), 0);
                spcs_s_add(kstatus, RDC_EGETSIZE, urdc->secondary.intf,
                    urdc->secondary.file, tmp1);
                (void) rdc_net_state(index, CCIO_ENABLELOG);
                return (RDC_EGETSIZE);
        }
        if (remote_size < (unsigned long long)urdc->volume_size) {
                (void) spcs_s_inttostring(
                    urdc->volume_size, tmp1, sizeof (tmp1), 0);
                /*
                 * Cheat, and covert to int, until we have
                 * spcs_s_unsignedlonginttostring().
                 */
                status = (int)remote_size;
                (void) spcs_s_inttostring(status, tmp2, sizeof (tmp2), 0);
                spcs_s_add(kstatus, RDC_ESIZE, urdc->primary.intf,
                    urdc->primary.file, tmp1, urdc->secondary.intf,
                    urdc->secondary.file, tmp2);
                (void) rdc_net_state(index, CCIO_ENABLELOG);
                return (RDC_ESIZE);
        }
        return (0);
}


static void
rdc_volume_update_svc(intptr_t arg)
{
        rdc_update_t *update = (rdc_update_t *)arg;
        rdc_k_info_t *krdc;
        rdc_k_info_t *this;
        rdc_u_info_t *urdc;
        struct net_bdata6 bd;
        int index;
        int rc;

#ifdef DEBUG_IIUPDATE
        cmn_err(CE_NOTE, "!SNDR received update request for %s",
            update->volume);
#endif

        if ((update->protocol != RDC_SVC_ONRETURN) &&
            (update->protocol != RDC_SVC_VOL_ENABLED)) {
                /* don't understand what the client intends to do */
                update->denied = 1;
                spcs_s_add(update->status, RDC_EVERSION);
                return;
        }

        index = rdc_lookup_enabled(update->volume, 0);
        if (index < 0)
                return;

        /*
```

```
         * warn II that this volume is in use by sndr so
         * II can validate the sizes of the master vs shadow
         * and avoid trouble later down the line with
         * size mis-matches between urdc->volume_size and
         * what is returned from nsc_partsize() which may
         * be the size of the master when replicating the shadow
         */
        if (update->protocol == RDC_SVC_VOL_ENABLED) {
                if (index >= 0)
                        update->denied = 1;
                return;
        }

        krdc = &rdc_k_info[index];
        urdc = &rdc_u_info[index];
        this = krdc;

        do {
                if (!(rdc_get_vflags(urdc) & RDC_LOGGING)) {
#ifdef DEBUG_IIUPDATE
                cmn_err(CE_NOTE, "!SNDR refused update request for %s",
                    update->volume);
#endif
                update->denied = 1;
                spcs_s_add(update->status, RDC_EMIRRORUP);
                return;
                }
                /* 1->many - all must be logging */
                if (IS_MANY(krdc) && IS_STATE(urdc, RDC_PRIMARY)) {
                        rdc_many_enter(krdc);
                        for (krdc = krdc->many_next; krdc != this;
                            krdc = krdc->many_next) {
                                urdc = &rdc_u_info[krdc->index];
                                if (!IS_ENABLED(urdc))
                                        continue;
                                break;
                        }
                        rdc_many_exit(krdc);
                }
        } while (krdc != this);

#ifdef DEBUG_IIUPDATE
        cmn_err(CE_NOTE, "!SNDR allowed update request for %s", update->volume);
#endif
        urdc = &rdc_u_info[krdc->index];
        do {

                bd.size = min(krdc->bitmap_size, (nsc_size_t)update->size);
                bd.data.data_val = (char *)update->bitmap;
                bd.offset = 0;
                bd.cd = index;

                if ((rc = RDC_OR_BITMAP(&bd)) != 0) {
                        update->denied = 1;
                        spcs_s_add(update->status, rc);
                        return;
                }
                urdc = &rdc_u_info[index];
                urdc->bits_set = RDC_COUNT_BITMAP(krdc);
                if (IS_MANY(krdc) && IS_STATE(urdc, RDC_PRIMARY)) {
                        rdc_many_enter(krdc);
                        for (krdc = krdc->many_next; krdc != this;
                            krdc = krdc->many_next) {
                                index = krdc->index;
                                if (!IS_ENABLED(urdc))
                                        continue;
```

```
                                break;
                        }
                        rdc_many_exit(krdc);
                }
        } while (krdc != this);


        /* II (or something else) has updated us, so no need for a sync */
        if (rdc_get_vflags(urdc) & (RDC_SYNC_NEEDED | RDC_RSYNC_NEEDED)) {
                rdc_many_enter(krdc);
                rdc_clr_flags(urdc, RDC_SYNC_NEEDED | RDC_RSYNC_NEEDED);
                rdc_many_exit(krdc);
        }

        if (krdc->bitmap_write > 0)
                (void) rdc_write_bitmap(krdc);
}


/*
 * rdc_check()
 *
 * Return 0 if the set is configured, enabled and the supplied
 * addressing information matches the in-kernel config, otherwise
 * return 1.
 */
static int
rdc_check(rdc_k_info_t *krdc, rdc_set_t *rdc_set)
{
        rdc_u_info_t *urdc = &rdc_u_info[krdc->index];

        ASSERT(MUTEX_HELD(&krdc->group->lock));

        if (!IS_ENABLED(urdc))
                return (1);

        if (strncmp(urdc->primary.file, rdc_set->primary.file,
            NSC_MAXPATH) != 0) {
#ifdef DEBUG
                cmn_err(CE_WARN, "!rdc_check: primary file mismatch %s vs %s",
                    urdc->primary.file, rdc_set->primary.file);
#endif
                return (1);
        }

        if (rdc_set->primary.addr.len != 0 &&
            bcmp(urdc->primary.addr.buf, rdc_set->primary.addr.buf,
            urdc->primary.addr.len) != 0) {
#ifdef DEBUG
                cmn_err(CE_WARN, "!rdc_check: primary address mismatch for %s",
                    urdc->primary.file);
#endif
                return (1);
        }

        if (strncmp(urdc->secondary.file, rdc_set->secondary.file,
            NSC_MAXPATH) != 0) {
#ifdef DEBUG
                cmn_err(CE_WARN, "!rdc_check: secondary file mismatch %s vs %s",
                    urdc->secondary.file, rdc_set->secondary.file);
#endif
                return (1);
        }

        if (rdc_set->secondary.addr.len != 0 &&
            bcmp(urdc->secondary.addr.buf, rdc_set->secondary.addr.buf,
	    urdc->secondary.addr.len) != 0) {
#ifdef DEBUG
		cmn_err(CE_WARN, "!rdc_check: secondary addr mismatch for %s",
		    urdc->secondary.file);
#endif
		return (1);
	}

	return (0);
}


/*
 * Lookup enabled sets for a bitmap match
 */

int
rdc_lookup_bitmap(char *pathname)
{
	rdc_u_info_t *urdc;
#ifdef DEBUG
	rdc_k_info_t *krdc;
#endif
	int index;

	for (index = 0; index < rdc_max_sets; index++) {
		urdc = &rdc_u_info[index];
#ifdef DEBUG
		krdc = &rdc_k_info[index];
#endif
		ASSERT(krdc->index == index);
		ASSERT(urdc->index == index);

		if (!IS_ENABLED(urdc))
			continue;

		if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
			if (strncmp(pathname, urdc->primary.bitmap,
			    NSC_MAXPATH) == 0)
				return (index);
		} else {
			if (strncmp(pathname, urdc->secondary.bitmap,
			    NSC_MAXPATH) == 0)
				return (index);
		}
	}

	return (-1);
}


/*
 * Translate a pathname to index into rdc_k_info[].
 * Returns first match that is enabled.
 */

int
rdc_lookup_enabled(char *pathname, int allow_disabling)
{
	rdc_u_info_t *urdc;
	rdc_k_info_t *krdc;
	int index;

restart:
	for (index = 0; index < rdc_max_sets; index++) {
		urdc = &rdc_u_info[index];
```

```
		krdc = &rdc_k_info[index];

		ASSERT(krdc->index == index);
		ASSERT(urdc->index == index);

		if (!IS_ENABLED(urdc))
			continue;

		if (allow_disabling == 0 && krdc->type_flag & RDC_UNREGISTER)
			continue;

		if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
			if (strncmp(pathname, urdc->primary.file,
			    NSC_MAXPATH) == 0)
				return (index);
		} else {
			if (strncmp(pathname, urdc->secondary.file,
			    NSC_MAXPATH) == 0)
				return (index);
		}
	}

	if (allow_disabling == 0) {
		/* None found, or only a disabling one found, so try again */
		allow_disabling = 1;
		goto restart;
	}

	return (-1);
}


/*
 * Translate a pathname to index into rdc_k_info[].
 * Returns first match that is configured.
 *
 * Used by enable & resume code.
 * Must be called with rdc_conf_lock held.
 */

int
rdc_lookup_configured(char *pathname)
{
	rdc_u_info_t *urdc;
	rdc_k_info_t *krdc;
	int index;

	ASSERT(MUTEX_HELD(&rdc_conf_lock));

	for (index = 0; index < rdc_max_sets; index++) {
		urdc = &rdc_u_info[index];
		krdc = &rdc_k_info[index];

		ASSERT(krdc->index == index);
		ASSERT(urdc->index == index);

		if (!IS_CONFIGURED(krdc))
			continue;

		if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
			if (strncmp(pathname, urdc->primary.file,
			    NSC_MAXPATH) == 0)
				return (index);
		} else {
			if (strncmp(pathname, urdc->secondary.file,
			    NSC_MAXPATH) == 0)
```

```
                                return (index);
                        }
        }

        return (-1);
}


/*
 * Looks up a configured set with matching secondary interface:volume
 * to check for illegal many-to-one volume configs.  To be used during
 * enable and resume processing.
 *
 * Must be called with rdc_conf_lock held.
 */

static int
rdc_lookup_many2one(rdc_set_t *rdc_set)
{
        rdc_u_info_t *urdc;
        rdc_k_info_t *krdc;
        int index;

        ASSERT(MUTEX_HELD(&rdc_conf_lock));

        for (index = 0; index < rdc_max_sets; index++) {
                urdc = &rdc_u_info[index];
                krdc = &rdc_k_info[index];

                if (!IS_CONFIGURED(krdc))
                        continue;

                if (strncmp(urdc->secondary.file,
                    rdc_set->secondary.file, NSC_MAXPATH) != 0)
                        continue;
                if (strncmp(urdc->secondary.intf,
                    rdc_set->secondary.intf, MAX_RDC_HOST_SIZE) != 0)
                        continue;

                break;
        }

        if (index < rdc_max_sets)
                return (index);
        else
                return (-1);
}


/*
 * Looks up an rdc set to check if it is already configured, to be used from
 * functions called from the config ioctl where the interface names can be
 * used for comparison.
 *
 * Must be called with rdc_conf_lock held.
 */

int
rdc_lookup_byname(rdc_set_t *rdc_set)
{
        rdc_u_info_t *urdc;
        rdc_k_info_t *krdc;
        int index;

        ASSERT(MUTEX_HELD(&rdc_conf_lock));
```

```
        for (index = 0; index < rdc_max_sets; index++) {
                urdc = &rdc_u_info[index];
                krdc = &rdc_k_info[index];

                ASSERT(krdc->index == index);
                ASSERT(urdc->index == index);

                if (!IS_CONFIGURED(krdc))
                        continue;

                if (strncmp(urdc->primary.file, rdc_set->primary.file,
                    NSC_MAXPATH) != 0)
                        continue;
                if (strncmp(urdc->primary.intf, rdc_set->primary.intf,
                    MAX_RDC_HOST_SIZE) != 0)
                        continue;
                if (strncmp(urdc->secondary.file, rdc_set->secondary.file,
                    NSC_MAXPATH) != 0)
                        continue;
                if (strncmp(urdc->secondary.intf, rdc_set->secondary.intf,
                    MAX_RDC_HOST_SIZE) != 0)
                        continue;

                break;
        }

        if (index < rdc_max_sets)
                return (index);
        else
                return (-1);
}

/*
 * Looks up a secondary hostname and device, to be used from
 * functions called from the config ioctl where the interface names can be
 * used for comparison.
 *
 * Must be called with rdc_conf_lock held.
 */

int
rdc_lookup_byhostdev(char *intf, char *file)
{
        rdc_u_info_t *urdc;
        rdc_k_info_t *krdc;
        int index;

        ASSERT(MUTEX_HELD(&rdc_conf_lock));

        for (index = 0; index < rdc_max_sets; index++) {
                urdc = &rdc_u_info[index];
                krdc = &rdc_k_info[index];

                ASSERT(krdc->index == index);
                ASSERT(urdc->index == index);

                if (!IS_CONFIGURED(krdc))
                        continue;

                if (strncmp(urdc->secondary.file, file,
                    NSC_MAXPATH) != 0)
                        continue;
                if (strncmp(urdc->secondary.intf, intf,
                    MAX_RDC_HOST_SIZE) != 0)
                        continue;
                break;
	}

	if (index < rdc_max_sets)
		return (index);
	else
		return (-1);
}


/*
 * Looks up an rdc set to see if it is currently enabled, to be used on the
 * server so that the interface addresses must be used for comparison, as
 * the interface names may differ from those used on the client.
 *
 */

int
rdc_lookup_byaddr(rdc_set_t *rdc_set)
{
	rdc_u_info_t *urdc;
#ifdef DEBUG
	rdc_k_info_t *krdc;
#endif
	int index;

	for (index = 0; index < rdc_max_sets; index++) {
		urdc = &rdc_u_info[index];
#ifdef DEBUG
		krdc = &rdc_k_info[index];
#endif
		ASSERT(krdc->index == index);
		ASSERT(urdc->index == index);

		if (!IS_ENABLED(urdc))
			continue;

		if (strcmp(urdc->primary.file, rdc_set->primary.file) != 0)
			continue;

		if (strcmp(urdc->secondary.file, rdc_set->secondary.file) != 0)
			continue;

		if (bcmp(urdc->primary.addr.buf, rdc_set->primary.addr.buf,
		    urdc->primary.addr.len) != 0) {
			continue;
		}

		if (bcmp(urdc->secondary.addr.buf, rdc_set->secondary.addr.buf,
		    urdc->secondary.addr.len) != 0) {
			continue;
		}

		break;
	}

	if (index < rdc_max_sets)
		return (index);
	else
		return (-1);
}


/*
 * Return index of first multihop or 1-to-many
 * Behavior controlled by setting ismany.
 * ismany TRUE (one-to-many)
```

```
 * ismany FALSE (multihops)
 *
 */
static int
rdc_lookup_multimany(rdc_k_info_t *krdc, const int ismany)
{
	rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
	rdc_u_info_t *utmp;
	rdc_k_info_t *ktmp;
	char *pathname;
	int index;
	int role;

	ASSERT(MUTEX_HELD(&rdc_conf_lock));
	ASSERT(MUTEX_HELD(&rdc_many_lock));

	if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
		/* this host is the primary of the krdc set */
		pathname = urdc->primary.file;
		if (ismany) {
			/*
			 * 1-many sets are linked by primary :
			 * look for matching primary on this host
			 */
			role = RDC_PRIMARY;
		} else {
			/*
			 * multihop sets link primary to secondary :
			 * look for matching secondary on this host
			 */
			role = 0;
		}
	} else {
		/* this host is the secondary of the krdc set */
		pathname = urdc->secondary.file;
		if (ismany) {
			/*
			 * 1-many sets are linked by primary, so if
			 * this host is the secondary of the set this
			 * cannot require 1-many linkage.
			 */
			return (-1);
		} else {
			/*
			 * multihop sets link primary to secondary :
			 * look for matching primary on this host
			 */
			role = RDC_PRIMARY;
		}
	}

	for (index = 0; index < rdc_max_sets; index++) {
		utmp = &rdc_u_info[index];
		ktmp = &rdc_k_info[index];

		if (!IS_CONFIGURED(ktmp)) {
			continue;
		}

		if (role == RDC_PRIMARY) {
			/*
			 * Find a primary that is this host and is not
			 * krdc but shares the same data volume as krdc.
			 */
			if ((rdc_get_vflags(utmp) & RDC_PRIMARY) &&
			    strncmp(utmp->primary.file, pathname,
			    NSC_MAXPATH) == 0 && (krdc != ktmp)) {
				break;
			}
		} else {
			/*
			 * Find a secondary that is this host and is not
			 * krdc but shares the same data volume as krdc.
			 */
			if (!(rdc_get_vflags(utmp) & RDC_PRIMARY) &&
			    strncmp(utmp->secondary.file, pathname,
			    NSC_MAXPATH) == 0 && (krdc != ktmp)) {
				break;
			}
		}
	}

	if (index < rdc_max_sets)
		return (index);
	else
		return (-1);
}

/*
 * Returns secondary match that is configured.
 *
 * Used by enable & resume code.
 * Must be called with rdc_conf_lock held.
 */

static int
rdc_lookup_secondary(char *pathname)
{
	rdc_u_info_t *urdc;
	rdc_k_info_t *krdc;
	int index;

	ASSERT(MUTEX_HELD(&rdc_conf_lock));

	for (index = 0; index < rdc_max_sets; index++) {
		urdc = &rdc_u_info[index];
		krdc = &rdc_k_info[index];

		ASSERT(krdc->index == index);
		ASSERT(urdc->index == index);

		if (!IS_CONFIGURED(krdc))
			continue;

		if (!IS_STATE(urdc, RDC_PRIMARY)) {
			if (strncmp(pathname, urdc->secondary.file,
			    NSC_MAXPATH) == 0)
			return (index);
		}
	}

	return (-1);
}


static nsc_fd_t *
rdc_open_direct(rdc_k_info_t *krdc)
{
	rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
	int rc;

	if (krdc->remote_fd == NULL)
```

```
		krdc->remote_fd = nsc_open(urdc->direct_file,
		    NSC_RDCHR_ID|NSC_DEVICE|NSC_RDWR, 0, 0, &rc);
	return (krdc->remote_fd);
}

static void
rdc_close_direct(rdc_k_info_t *krdc)
{
	rdc_u_info_t *urdc = &rdc_u_info[krdc->index];

	urdc->direct_file[0] = 0;
	if (krdc->remote_fd) {
		if (nsc_close(krdc->remote_fd) == 0) {
			krdc->remote_fd = NULL;
		}
	}
}


#ifdef DEBUG_MANY
static void
print_many(rdc_k_info_t *start)
{
	rdc_k_info_t *p = start;
	rdc_u_info_t *q = &rdc_u_info[p->index];

	do {
		cmn_err(CE_CONT, "!krdc %p, %s %s (many_nxt %p multi_nxt %p)\n",
		    p, q->primary.file, q->secondary.file, p->many_next,
		    p->multi_next);
		delay(10);
		p = p->many_next;
		q = &rdc_u_info[p->index];
	} while (p && p != start);
}
#endif /* DEBUG_MANY */


static int
add_to_multi(rdc_k_info_t *krdc)
{
	rdc_u_info_t *urdc;
	rdc_k_info_t *ktmp;
	rdc_u_info_t *utmp;
	int mindex;
	int domulti;

	urdc = &rdc_u_info[krdc->index];

	ASSERT(MUTEX_HELD(&rdc_conf_lock));
	ASSERT(MUTEX_HELD(&rdc_many_lock));

	/* Now find companion krdc */
	mindex = rdc_lookup_multimany(krdc, FALSE);

#ifdef DEBUG_MANY
	cmn_err(CE_NOTE,
	    "!add_to_multi: lookup_multimany: mindex %d prim %s sec %s",
	    mindex, urdc->primary.file, urdc->secondary.file);
#endif

	if (mindex >= 0) {
		ktmp = &rdc_k_info[mindex];
		utmp = &rdc_u_info[mindex];

		domulti = 1;
		if ((rdc_get_vflags(urdc) & RDC_PRIMARY) &&
		    ktmp->multi_next != NULL) {
			/*
			 * We are adding a new primary to a many
			 * group that is the target of a multihop, just
			 * ignore it since we are linked in elsewhere.
			 */
			domulti = 0;
		}

		if (domulti) {
			if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
				/* Is previous leg using direct file I/O? */
				if (utmp->direct_file[0] != 0) {
					/* It is, so cannot proceed */
					return (-1);
				}
			} else {
				/* Is this leg using direct file I/O? */
				if (urdc->direct_file[0] != 0) {
					/* It is, so cannot proceed */
					return (-1);
				}
			}
			krdc->multi_next = ktmp;
			ktmp->multi_next = krdc;
		}
	} else {
		krdc->multi_next = NULL;
#ifdef DEBUG_MANY
		cmn_err(CE_NOTE, "!add_to_multi: NULL multi_next index %d",
		    krdc->index);
#endif
	}

	return (0);
}


/*
 * Add a new set to the circular list of 1-to-many primaries and chain
 * up any multihop as well.
 */
static int
add_to_many(rdc_k_info_t *krdc)
{
	rdc_k_info_t *okrdc;
	int oindex;

	ASSERT(MUTEX_HELD(&rdc_conf_lock));

	rdc_many_enter(krdc);

	if (add_to_multi(krdc) < 0) {
		rdc_many_exit(krdc);
		return (-1);
	}

	oindex = rdc_lookup_multimany(krdc, TRUE);
	if (oindex < 0) {
#ifdef DEBUG_MANY
		print_many(krdc);
#endif
		rdc_many_exit(krdc);
		return (0);
```

```
	}

	okrdc = &rdc_k_info[oindex];

#ifdef DEBUG_MANY
	print_many(okrdc);
#endif
	krdc->many_next = okrdc->many_next;
	okrdc->many_next = krdc;

#ifdef DEBUG_MANY
	print_many(okrdc);
#endif
	rdc_many_exit(krdc);
	return (0);
}


/*
 * Remove a set from the circular list of 1-to-many primaries.
 */
static void
remove_from_many(rdc_k_info_t *old)
{
	rdc_u_info_t *uold = &rdc_u_info[old->index];
	rdc_k_info_t *p, *q;

	ASSERT(MUTEX_HELD(&rdc_conf_lock));

	rdc_many_enter(old);

#ifdef DEBUG_MANY
	cmn_err(CE_NOTE, "!rdc: before remove_from_many");
	print_many(old);
#endif

	if (old->many_next == old) {
		/* remove from multihop */
		if ((q = old->multi_next) != NULL) {
			ASSERT(q->multi_next == old);
			q->multi_next = NULL;
			old->multi_next = NULL;
		}

		rdc_many_exit(old);
		return;
	}

	/* search */
	for (p = old->many_next; p->many_next != old; p = p->many_next)
	;

	p->many_next = old->many_next;
	old->many_next = old;

	if ((q = old->multi_next) != NULL) {
		/*
		 * old was part of a multihop, so switch multi pointers
		 * to someone remaining on the many chain
		 */
		ASSERT(p->multi_next == NULL);

		q->multi_next = p;
		p->multi_next = q;
		old->multi_next = NULL;
	}
```

```
#ifdef DEBUG_MANY
        if (p == old) {
                cmn_err(CE_NOTE, "!rdc: after remove_from_many empty");
        } else {
                cmn_err(CE_NOTE, "!rdc: after remove_from_many");
                print_many(p);
        }
#endif

        rdc_clr_mflags(&rdc_u_info[p->index],
            (rdc_get_vflags(uold) & RDC_MFLAGS));

        rdc_many_exit(old);
}


static int
_rdc_enable(rdc_set_t *rdc_set, int options, spcs_s_info_t kstatus)
{
        int index;
        char *rhost;
        struct netbuf *addrp;
        rdc_k_info_t *krdc;
        rdc_u_info_t *urdc;
        rdc_srv_t *svp = NULL;
        char *local_file;
        char *local_bitmap;
        char *diskq;
        int rc;
        nsc_size_t maxfbas;
        rdc_group_t *grp;

        if ((rdc_set->primary.intf[0] == 0) ||
            (rdc_set->primary.addr.len == 0) ||
            (rdc_set->primary.file[0] == 0) ||
            (rdc_set->primary.bitmap[0] == 0) ||
            (rdc_set->secondary.intf[0] == 0) ||
            (rdc_set->secondary.addr.len == 0) ||
            (rdc_set->secondary.file[0] == 0) ||
            (rdc_set->secondary.bitmap[0] == 0)) {
                spcs_s_add(kstatus, RDC_EEMPTY);
                return (RDC_EEMPTY);
        }

        /* Next check there aren't any enabled rdc sets which match. */

        mutex_enter(&rdc_conf_lock);

        if (rdc_lookup_byname(rdc_set) >= 0) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EENABLED, rdc_set->primary.intf,
                    rdc_set->primary.file, rdc_set->secondary.intf,
                    rdc_set->secondary.file);
                return (RDC_EENABLED);
        }

        if (rdc_lookup_many2one(rdc_set) >= 0) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EMANY2ONE, rdc_set->primary.intf,
                    rdc_set->primary.file, rdc_set->secondary.intf,
                    rdc_set->secondary.file);
                return (RDC_EMANY2ONE);
        }

        if (rdc_set->netconfig->knc_proto == NULL) {
```

```
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_ENETCONFIG);
                return (RDC_ENETCONFIG);
        }

        if (rdc_set->primary.addr.len == 0) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_ENETBUF, rdc_set->primary.file);
                return (RDC_ENETBUF);
        }

        if (rdc_set->secondary.addr.len == 0) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_ENETBUF, rdc_set->secondary.file);
                return (RDC_ENETBUF);
        }

        /* Check that the local data volume isn't in use as a bitmap */
        if (options & RDC_OPT_PRIMARY)
                local_file = rdc_set->primary.file;
        else
                local_file = rdc_set->secondary.file;
        if (rdc_lookup_bitmap(local_file) >= 0) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EVOLINUSE, local_file);
                return (RDC_EVOLINUSE);
        }

        /* check that the secondary data volume isn't in use */
        if (!(options & RDC_OPT_PRIMARY)) {
                local_file = rdc_set->secondary.file;
                if (rdc_lookup_secondary(local_file) >= 0) {
                        mutex_exit(&rdc_conf_lock);
                        spcs_s_add(kstatus, RDC_EVOLINUSE, local_file);
                        return (RDC_EVOLINUSE);
                }
        }

        /* check that the local data vol is not in use as a diskqueue */
        if (options & RDC_OPT_PRIMARY) {
                if (rdc_lookup_diskq(rdc_set->primary.file) >= 0) {
                        mutex_exit(&rdc_conf_lock);
                        spcs_s_add(kstatus,
                            RDC_EVOLINUSE, rdc_set->primary.file);
                        return (RDC_EVOLINUSE);
                }
        }

        /* Check that the bitmap isn't in use as a data volume */
        if (options & RDC_OPT_PRIMARY)
                local_bitmap = rdc_set->primary.bitmap;
        else
                local_bitmap = rdc_set->secondary.bitmap;
        if (rdc_lookup_configured(local_bitmap) >= 0) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EBMPINUSE, local_bitmap);
                return (RDC_EBMPINUSE);
        }

        /* Check that the bitmap isn't already in use as a bitmap */
        if (rdc_lookup_bitmap(local_bitmap) >= 0) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EBMPINUSE, local_bitmap);
                return (RDC_EBMPINUSE);
        }
        /* check that the diskq (if here) is not in use */
        diskq = rdc_set->disk_queue;
        if (diskq[0] && rdc_diskq_inuse(rdc_set, diskq)) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EDISKQINUSE, diskq);
                return (RDC_EDISKQINUSE);
        }


        /* Set urdc->volume_size */
        index = rdc_dev_open(rdc_set, options);
        if (index < 0) {
                mutex_exit(&rdc_conf_lock);
                if (options & RDC_OPT_PRIMARY)
                        spcs_s_add(kstatus, RDC_EOPEN, rdc_set->primary.intf,
                            rdc_set->primary.file);
                else
                        spcs_s_add(kstatus, RDC_EOPEN, rdc_set->secondary.intf,
                            rdc_set->secondary.file);
                return (RDC_EOPEN);
        }

        urdc = &rdc_u_info[index];
        krdc = &rdc_k_info[index];

        /* copy relevant parts of rdc_set to urdc field by field */

        (void) strncpy(urdc->primary.intf, rdc_set->primary.intf,
            MAX_RDC_HOST_SIZE);
        (void) strncpy(urdc->secondary.intf, rdc_set->secondary.intf,
            MAX_RDC_HOST_SIZE);

        (void) strncpy(urdc->group_name, rdc_set->group_name, NSC_MAXPATH);
        (void) strncpy(urdc->disk_queue, rdc_set->disk_queue, NSC_MAXPATH);

        dup_rdc_netbuf(&rdc_set->primary.addr, &urdc->primary.addr);
        (void) strncpy(urdc->primary.file, rdc_set->primary.file, NSC_MAXPATH);
        (void) strncpy(urdc->primary.bitmap, rdc_set->primary.bitmap,
            NSC_MAXPATH);

        dup_rdc_netbuf(&rdc_set->secondary.addr, &urdc->secondary.addr);
        (void) strncpy(urdc->secondary.file, rdc_set->secondary.file,
            NSC_MAXPATH);
        (void) strncpy(urdc->secondary.bitmap, rdc_set->secondary.bitmap,
            NSC_MAXPATH);

        urdc->setid = rdc_set->setid;

        /*
         * before we try to add to group, or create one, check out
         * if we are doing the wrong thing with the diskq
         */

        if (urdc->disk_queue[0] && (options & RDC_OPT_SYNC)) {
                mutex_exit(&rdc_conf_lock);
                rdc_dev_close(krdc);
                spcs_s_add(kstatus, RDC_EQWRONGMODE);
                return (RDC_EQWRONGMODE);
        }

        if ((rc = add_to_group(krdc, options, RDC_CMD_ENABLE)) != 0) {
                mutex_exit(&rdc_conf_lock);
                rdc_dev_close(krdc);
                if (rc == RDC_EQNOADD) {
                        spcs_s_add(kstatus, RDC_EQNOADD, rdc_set->disk_queue);
                        return (RDC_EQNOADD);
```

```
                } else {
                        spcs_s_add(kstatus, RDC_EGROUP,
                            rdc_set->primary.intf, rdc_set->primary.file,
                            rdc_set->secondary.intf, rdc_set->secondary.file,
                            rdc_set->group_name);
                        return (RDC_EGROUP);
                }
        }

        /*
         * maxfbas was set in rdc_dev_open as primary's maxfbas.
         * If diskq's maxfbas is smaller, then use diskq's.
         */
        grp = krdc->group;
        if (grp && RDC_IS_DISKQ(grp) && (grp->diskqfd != 0)) {
                rc = _rdc_rsrv_diskq(grp);
                if (RDC_SUCCESS(rc)) {
                        rc = nsc_maxfbas(grp->diskqfd, 0, &maxfbas);
                        if (rc == 0) {
#ifdef DEBUG
                                if (krdc->maxfbas != maxfbas)
                                        cmn_err(CE_NOTE,
                                            "!_rdc_enable: diskq maxfbas = %"
                                            NSC_SZFMT ", primary maxfbas = %"
                                            NSC_SZFMT, maxfbas, krdc->maxfbas);
#endif
                                krdc->maxfbas = min(krdc->maxfbas, maxfbas);
                        } else {
                                cmn_err(CE_WARN,
                                    "!_rdc_enable: diskq maxfbas failed (%d)",
                                    rc);
                        }
                        _rdc_rlse_diskq(grp);
                } else {
                        cmn_err(CE_WARN,
                            "!_rdc_enable: diskq reserve failed (%d)", rc);
                }
        }

        rdc_init_flags(urdc);
        (void) strncpy(urdc->direct_file, rdc_set->direct_file, NSC_MAXPATH);
        if ((options & RDC_OPT_PRIMARY) && rdc_set->direct_file[0]) {
                if (rdc_open_direct(krdc) == NULL)
                        rdc_set_flags(urdc, RDC_FCAL_FAILED);
        }

        krdc->many_next = krdc;

        ASSERT(krdc->type_flag == 0);
        krdc->type_flag = RDC_CONFIGURED;

        if (options & RDC_OPT_PRIMARY)
                rdc_set_flags(urdc, RDC_PRIMARY);

        if (options & RDC_OPT_ASYNC)
                krdc->type_flag |= RDC_ASYNCMODE;

        set_busy(krdc);
        urdc->syshostid = rdc_set->syshostid;

        if (add_to_many(krdc) < 0) {
                mutex_exit(&rdc_conf_lock);

                rdc_group_enter(krdc);

                spcs_s_add(kstatus, RDC_EMULTI);
                rc = RDC_EMULTI;
                goto fail;
        }

        /* Configured but not enabled */
        ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

        mutex_exit(&rdc_conf_lock);

        rdc_group_enter(krdc);

        /* Configured but not enabled */
        ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

        /*
         * The rdc set is configured but not yet enabled. Other operations must
         * ignore this set until it is enabled.
         */

        urdc->sync_pos = 0;

        if (rdc_set->maxqfbas > 0)
                urdc->maxqfbas = rdc_set->maxqfbas;
        else
                urdc->maxqfbas = rdc_maxthres_queue;

        if (rdc_set->maxqitems > 0)
                urdc->maxqitems = rdc_set->maxqitems;
        else
                urdc->maxqitems = rdc_max_qitems;

        if (rdc_set->asyncthr > 0)
                urdc->asyncthr = rdc_set->asyncthr;
        else
                urdc->asyncthr = rdc_asyncthr;

        if (urdc->autosync == -1) {
                /* Still unknown */
                if (rdc_set->autosync > 0)
                        urdc->autosync = 1;
                else
                        urdc->autosync = 0;
        }

        urdc->netconfig = rdc_set->netconfig;

        if (options & RDC_OPT_PRIMARY) {
                rhost = rdc_set->secondary.intf;
                addrp = &rdc_set->secondary.addr;
        } else {
                rhost = rdc_set->primary.intf;
                addrp = &rdc_set->primary.addr;
        }

        if (options & RDC_OPT_ASYNC)
                rdc_set_flags(urdc, RDC_ASYNC);

        svp = rdc_create_svinfo(rhost, addrp, urdc->netconfig);
        if (svp == NULL) {
                spcs_s_add(kstatus, ENOMEM);
                rc = ENOMEM;
                goto fail;
        }
        urdc->netconfig = NULL;         /* This will be no good soon */

        rdc_kstat_create(index);
```

```
        /* Don't set krdc->intf here */

        if (rdc_enable_bitmap(krdc, options & RDC_OPT_SETBMP) < 0)
                goto bmpfail;

        RDC_ZERO_BITREF(krdc);
        if (krdc->lsrv == NULL)
                krdc->lsrv = svp;
        else {
#ifdef DEBUG
                cmn_err(CE_WARN, "!_rdc_enable: krdc->lsrv already set: %p",
                    (void *) krdc->lsrv);
#endif
                rdc_destroy_svinfo(svp);
        }
        svp = NULL;

        /* Configured but not enabled */
        ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

        /* And finally */

        krdc->remote_index = -1;
        /* Should we set the whole group logging? */
        rdc_set_flags(urdc, RDC_ENABLED | RDC_LOGGING);

        rdc_group_exit(krdc);

        if (rdc_intercept(krdc) != 0) {
                rdc_group_enter(krdc);
                rdc_clr_flags(urdc, RDC_ENABLED);
                if (options & RDC_OPT_PRIMARY)
                        spcs_s_add(kstatus, RDC_EREGISTER, urdc->primary.file);
                else
                        spcs_s_add(kstatus, RDC_EREGISTER,
                            urdc->secondary.file);
#ifdef DEBUG
                cmn_err(CE_NOTE, "!nsc_register_path failed %s",
                    urdc->primary.file);
#endif
                rc = RDC_EREGISTER;
                goto bmpfail;
        }
#ifdef DEBUG
        cmn_err(CE_NOTE, "!SNDR: enabled %s %s", urdc->primary.file,
            urdc->secondary.file);
#endif

        rdc_write_state(urdc);

        mutex_enter(&rdc_conf_lock);
        wakeup_busy(krdc);
        mutex_exit(&rdc_conf_lock);

        return (0);

bmpfail:
        if (options & RDC_OPT_PRIMARY)
                spcs_s_add(kstatus, RDC_EBITMAP, rdc_set->primary.bitmap);
        else
                spcs_s_add(kstatus, RDC_EBITMAP, rdc_set->secondary.bitmap);
        rc = RDC_EBITMAP;
        if (rdc_get_vflags(urdc) & RDC_ENABLED) {
                rdc_group_exit(krdc);
                (void) rdc_unintercept(krdc);
                rdc_group_enter(krdc);
        }

fail:
        rdc_kstat_delete(index);
        rdc_group_exit(krdc);
        if (krdc->intf) {
                rdc_if_t *ip = krdc->intf;
                mutex_enter(&rdc_conf_lock);
                krdc->intf = NULL;
                rdc_remove_from_if(ip);
                mutex_exit(&rdc_conf_lock);
        }
        rdc_group_enter(krdc);
        /* Configured but not enabled */
        ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

        rdc_dev_close(krdc);
        rdc_close_direct(krdc);
        rdc_destroy_svinfo(svp);

        /* Configured but not enabled */
        ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

        rdc_group_exit(krdc);

        mutex_enter(&rdc_conf_lock);

        /* Configured but not enabled */
        ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

        remove_from_group(krdc);

        if (IS_MANY(krdc) || IS_MULTI(krdc))
                remove_from_many(krdc);

        rdc_u_init(urdc);

        ASSERT(krdc->type_flag & RDC_CONFIGURED);
        krdc->type_flag = 0;
        wakeup_busy(krdc);

        mutex_exit(&rdc_conf_lock);

        return (rc);
}

static int
rdc_enable(rdc_config_t *uparms, spcs_s_info_t kstatus)
{
        int rc;
        char itmp[10];

        if (!(uparms->options & RDC_OPT_SYNC) &&
            !(uparms->options & RDC_OPT_ASYNC)) {
                rc = RDC_EEINVAL;
                (void) spcs_s_inttostring(
                    uparms->options, itmp, sizeof (itmp), 1);
                spcs_s_add(kstatus, RDC_EEINVAL, itmp);
                goto done;
        }

        if (!(uparms->options & RDC_OPT_PRIMARY) &&
            !(uparms->options & RDC_OPT_SECONDARY)) {
                rc = RDC_EEINVAL;
                (void) spcs_s_inttostring(
```

```
                    uparms->options, itmp, sizeof (itmp), 1);
                spcs_s_add(kstatus, RDC_EEINVAL, itmp);
                goto done;
        }

        if (!(uparms->options & RDC_OPT_SETBMP) &&
            !(uparms->options & RDC_OPT_CLRBMP)) {
                rc = RDC_EEINVAL;
                (void) spcs_s_inttostring(
                    uparms->options, itmp, sizeof (itmp), 1);
                spcs_s_add(kstatus, RDC_EEINVAL, itmp);
                goto done;
        }

        rc = _rdc_enable(uparms->rdc_set, uparms->options, kstatus);
done:
        return (rc);
}

/* ARGSUSED */
static int
_rdc_disable(rdc_k_info_t *krdc, rdc_config_t *uap, spcs_s_info_t kstatus)
{
        rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
        rdc_if_t *ip;
        int index = krdc->index;
        disk_queue *q;
        rdc_set_t *rdc_set = uap->rdc_set;

        ASSERT(krdc->group != NULL);
        rdc_group_enter(krdc);
#ifdef DEBUG
        ASSERT(rdc_check(krdc, rdc_set) == 0);
#else
        if (((uap->options & RDC_OPT_FORCE_DISABLE) == 0) &&
            rdc_check(krdc, rdc_set)) {
                rdc_group_exit(krdc);
                spcs_s_add(kstatus, RDC_EALREADY, rdc_set->primary.file,
                    rdc_set->secondary.file);
                return (RDC_EALREADY);
        }
#endif

        if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
                halt_sync(krdc);
                ASSERT(IS_ENABLED(urdc));
        }
        q = &krdc->group->diskq;

        if (IS_ASYNC(urdc) && RDC_IS_DISKQ(krdc->group) &&
            ((!IS_STATE(urdc, RDC_LOGGING)) && (!QEMPTY(q)))) {
                krdc->type_flag &= ~RDC_DISABLEPEND;
                rdc_group_exit(krdc);
                spcs_s_add(kstatus, RDC_EQNOTEMPTY, urdc->disk_queue);
                return (RDC_EQNOTEMPTY);
        }
        rdc_group_exit(krdc);
        (void) rdc_unintercept(krdc);

#ifdef DEBUG
        cmn_err(CE_NOTE, "!SNDR: disabled %s %s", urdc->primary.file,
            urdc->secondary.file);
#endif

        /* Configured but not enabled */
        ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));
```

```c
        /*
         * No new io can come in through the io provider.
         * Wait for the async flusher to finish.
         */

        if (IS_ASYNC(urdc) && !RDC_IS_DISKQ(krdc->group)) {
                int tries = 2; /* in case of hopelessly stuck flusher threads */
#ifdef DEBUG
                net_queue *qp = &krdc->group->ra_queue;
#endif
                do {
                        if (!krdc->group->rdc_writer)
                                (void) rdc_writer(krdc->index);

                        (void) rdc_drain_queue(krdc->index);

                } while (krdc->group->rdc_writer && tries--);

                /* ok, force it to happen... */
                if (rdc_drain_queue(krdc->index) != 0) {
                        do {
                                mutex_enter(&krdc->group->ra_queue.net_qlock);
                                krdc->group->asyncdis = 1;
                                cv_broadcast(&krdc->group->asyncqcv);
                                mutex_exit(&krdc->group->ra_queue.net_qlock);
                                cmn_err(CE_WARN,
                                    "!SNDR: async I/O pending and not flushed "
                                    "for %s during disable",
                                    urdc->primary.file);
#ifdef DEBUG
                                cmn_err(CE_WARN,
                                    "!nitems: %" NSC_SZFMT " nblocks: %"
                                    NSC_SZFMT " head: 0x%p tail: 0x%p",
                                    qp->nitems, qp->blocks,
                                    (void *)qp->net_qhead,
                                    (void *)qp->net_qtail);
#endif
                        } while (krdc->group->rdc_thrnum > 0);
                }
        }

        mutex_enter(&rdc_conf_lock);
        ip = krdc->intf;
        krdc->intf = 0;

        if (ip) {
                rdc_remove_from_if(ip);
        }

        mutex_exit(&rdc_conf_lock);

        rdc_group_enter(krdc);

        /* Configured but not enabled */
        ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

        /* Must not hold group lock during this function */
        rdc_group_exit(krdc);
        while (rdc_dump_alloc_bufs_cd(krdc->index) == EAGAIN)
                delay(2);
        rdc_group_enter(krdc);

        (void) rdc_clear_state(krdc);

        rdc_free_bitmap(krdc, RDC_CMD_DISABLE);
```

```c
        rdc_close_bitmap(krdc);

        rdc_dev_close(krdc);
        rdc_close_direct(krdc);

        /* Configured but not enabled */
        ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

        rdc_group_exit(krdc);

        /*
         * we should now unregister the queue, with no conflicting
         * locks held. This is the last(only) member of the group
         */
        if (krdc->group && RDC_IS_DISKQ(krdc->group) &&
            krdc->group->count == 1) { /* stop protecting queue */
                rdc_unintercept_diskq(krdc->group);
        }

        mutex_enter(&rdc_conf_lock);

        /* Configured but not enabled */
        ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

        wait_busy(krdc);

        if (IS_MANY(krdc) || IS_MULTI(krdc))
                remove_from_many(krdc);

        remove_from_group(krdc);

        krdc->remote_index = -1;
        ASSERT(krdc->type_flag & RDC_CONFIGURED);
        ASSERT(krdc->type_flag & RDC_DISABLEPEND);
        krdc->type_flag = 0;
#ifdef  DEBUG
        if (krdc->dcio_bitmap)
                cmn_err(CE_WARN, "!_rdc_disable: possible mem leak, "
                    "dcio_bitmap");
#endif
        krdc->dcio_bitmap = NULL;
        krdc->bitmap_ref = NULL;
        krdc->bitmap_size = 0;
        krdc->maxfbas = 0;
        krdc->bitmap_write = 0;
        krdc->disk_status = 0;
        rdc_destroy_svinfo(krdc->lsrv);
        krdc->lsrv = NULL;
        krdc->multi_next = NULL;

        rdc_u_init(urdc);

        mutex_exit(&rdc_conf_lock);
        rdc_kstat_delete(index);

        return (0);
}

static int
rdc_disable(rdc_config_t *uparms, spcs_s_info_t kstatus)
{
        rdc_k_info_t *krdc;
        int index;
        int rc;

        mutex_enter(&rdc_conf_lock);
        index = rdc_lookup_byname(uparms->rdc_set);
        if (index >= 0)
                krdc = &rdc_k_info[index];
        if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }

        krdc->type_flag |= RDC_DISABLEPEND;
        wait_busy(krdc);
        if (krdc->type_flag == 0) {
                /* A resume or enable failed */
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }
        mutex_exit(&rdc_conf_lock);

        rc = _rdc_disable(krdc, uparms, kstatus);
        return (rc);
}


/*
 * Checks whether the state of one of the other sets in the 1-many or
 * multi-hop config should prevent a sync from starting on this one.
 * Return NULL if no just cause or impediment is found, otherwise return
 * a pointer to the offending set.
 */
static rdc_u_info_t *
rdc_allow_pri_sync(rdc_u_info_t *urdc, int options)
{
        rdc_k_info_t *krdc = &rdc_k_info[urdc->index];
        rdc_k_info_t *ktmp;
        rdc_u_info_t *utmp;
        rdc_k_info_t *kmulti = NULL;

        ASSERT(rdc_get_vflags(urdc) & RDC_PRIMARY);

        rdc_many_enter(krdc);

        /*
         * In the reverse sync case we need to check the previous leg of
         * the multi-hop config. The link to that set can be from any of
         * the 1-many list, so as we go through we keep an eye open for it.
         */
        if ((options & RDC_OPT_REVERSE) && (IS_MULTI(krdc))) {
                /* This set links to the first leg */
                ktmp = krdc->multi_next;
                utmp = &rdc_u_info[ktmp->index];
                if (IS_ENABLED(utmp))
                        kmulti = ktmp;
        }

        if (IS_MANY(krdc)) {
                for (ktmp = krdc->many_next; ktmp != krdc;
                    ktmp = ktmp->many_next) {
                        utmp = &rdc_u_info[ktmp->index];

                        if (!IS_ENABLED(utmp))
                                continue;
```

```
                        if (options & RDC_OPT_FORWARD) {
                                /*
                                 * Reverse sync needed is bad, as it means a
                                 * reverse sync in progress or started and
                                 * didn’t complete, so this primary volume
                                 * is not consistent. So we shouldn’t copy
                                 * it to its secondary.
                                 */
                                if (rdc_get_mflags(utmp) & RDC_RSYNC_NEEDED) {
                                        rdc_many_exit(krdc);
                                        return (utmp);
                                }
                        } else {
                                /* Reverse, so see if we need to spot kmulti */
                                if ((kmulti == NULL) && (IS_MULTI(ktmp))) {
                                        /* This set links to the first leg */
                                        kmulti = ktmp->multi_next;
                                        if (!IS_ENABLED(
                                            &rdc_u_info[kmulti->index]))
                                                kmulti = NULL;
                                }

                                /*
                                 * Non-logging is bad, as the bitmap will
                                 * be updated with the bits for this sync.
                                 */
                                if (!(rdc_get_vflags(utmp) & RDC_LOGGING)) {
                                        rdc_many_exit(krdc);
                                        return (utmp);
                                }
                        }
                }
        }

        if (kmulti) {
                utmp = &rdc_u_info[kmulti->index];
                ktmp = kmulti;  /* In case we decide we do need to use ktmp */

                ASSERT(options & RDC_OPT_REVERSE);

                if (IS_REPLICATING(utmp)) {
                        /*
                         * Replicating is bad as data is already flowing to
                         * the target of the requested sync operation.
                         */
                        rdc_many_exit(krdc);
                        return (utmp);
                }

                if (rdc_get_vflags(utmp) & RDC_SYNCING) {
                        /*
                         * Forward sync in progress is bad, as data is
                         * already flowing to the target of the requested
                         * sync operation.
                         * Reverse sync in progress is bad, as the primary
                         * has already decided which data to copy.
                         */
                        rdc_many_exit(krdc);
                        return (utmp);
                }

                /*
                 * Clear the "sync needed" flags, as the multi-hop secondary
                 * will be updated via this requested sync operation, so does
                 * not need to complete its aborted forward sync.
                 */
```

```
		if (rdc_get_vflags(utmp) & RDC_SYNC_NEEDED)
			rdc_clr_flags(utmp, RDC_SYNC_NEEDED);
	}

	if (IS_MANY(krdc) && (options & RDC_OPT_REVERSE)) {
		for (ktmp = krdc->many_next; ktmp != krdc;
		    ktmp = ktmp->many_next) {
			utmp = &rdc_u_info[ktmp->index];
			if (!IS_ENABLED(utmp))
				continue;

			/*
			 * Clear any "reverse sync needed" flags, as the
			 * volume will be updated via this requested
			 * sync operation, so does not need to complete
			 * its aborted reverse sync.
			 */
			if (rdc_get_mflags(utmp) & RDC_RSYNC_NEEDED)
				rdc_clr_mflags(utmp, RDC_RSYNC_NEEDED);
		}
	}

	rdc_many_exit(krdc);

	return (NULL);
}

static void
_rdc_sync_wrthr(void *thrinfo)
{
	rdc_syncthr_t *syncinfo = (rdc_syncthr_t *)thrinfo;
	nsc_buf_t *handle = NULL;
	rdc_k_info_t *krdc = syncinfo->krdc;
	int rc;
	int tries = 0;

	DTRACE_PROBE2(rdc_sync_loop_netwrite_start, int, krdc->index,
	    nsc_buf_t *, handle);

retry:
	rc = nsc_alloc_buf(RDC_U_FD(krdc), syncinfo->offset, syncinfo->len,
	    NSC_READ | NSC_NOCACHE, &handle);

	if (!RDC_SUCCESS(rc) || krdc->remote_index < 0) {
		DTRACE_PROBE(rdc_sync_wrthr_alloc_buf_err);
		goto failed;
	}

	rdc_group_enter(krdc);
	if ((krdc->disk_status == 1) || (krdc->dcio_bitmap == NULL)) {
		rdc_group_exit(krdc);
		goto failed;
	}
	rdc_group_exit(krdc);

	if ((rc = rdc_net_write(krdc->index, krdc->remote_index, handle,
	    handle->sb_pos, handle->sb_len, RDC_NOSEQ, RDC_NOQUE, NULL)) > 0) {
		rdc_u_info_t *urdc = &rdc_u_info[krdc->index];

		/*
		 * The following is to handle
		 * the case where the secondary side
		 * has thrown our buffer handle token away in a
		 * attempt to preserve its health on restart
		 */
		if ((rc == EPROTO) && (tries < 3)) {
```

```
			(void) nsc_free_buf(handle);
			handle = NULL;
			tries++;
			delay(HZ >> 2);
			goto retry;
		}

		DTRACE_PROBE(rdc_sync_wrthr_remote_write_err);
		cmn_err(CE_WARN, "!rdc_sync_wrthr: remote write failed (%d) "
		    "0x%x", rc, rdc_get_vflags(urdc));

		goto failed;
	}
	(void) nsc_free_buf(handle);
	handle = NULL;

	return;
failed:
	(void) nsc_free_buf(handle);
	syncinfo->status->offset = syncinfo->offset;
}

/*
 * see above comments on _rdc_sync_wrthr
 */
static void
_rdc_sync_rdthr(void *thrinfo)
{
	rdc_syncthr_t *syncinfo = (rdc_syncthr_t *)thrinfo;
	nsc_buf_t *handle = NULL;
	rdc_k_info_t *krdc = syncinfo->krdc;
	rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
	int rc;

	rc = nsc_alloc_buf(RDC_U_FD(krdc), syncinfo->offset, syncinfo->len,
	    NSC_WRITE | NSC_WRTHRU | NSC_NOCACHE, &handle);

	if (!RDC_SUCCESS(rc) || krdc->remote_index < 0) {
		goto failed;
	}
	rdc_group_enter(krdc);
	if ((krdc->disk_status == 1) || (krdc->dcio_bitmap == NULL)) {
		rdc_group_exit(krdc);
		goto failed;
	}
	rdc_group_exit(krdc);

	rc = rdc_net_read(krdc->index, krdc->remote_index, handle,
	    handle->sb_pos, handle->sb_len);

	if (!RDC_SUCCESS(rc)) {
		cmn_err(CE_WARN, "!rdc_sync_rdthr: remote read failed(%d)", rc);
		goto failed;
	}
	if (!IS_STATE(urdc, RDC_FULL))
		rdc_set_bitmap_many(krdc, handle->sb_pos, handle->sb_len);

	rc = nsc_write(handle, handle->sb_pos, handle->sb_len, 0);

	if (!RDC_SUCCESS(rc)) {
		rdc_many_enter(krdc);
		rdc_set_flags_log(urdc, RDC_VOL_FAILED, "nsc_write failed");
		rdc_many_exit(krdc);
		rdc_write_state(urdc);
		goto failed;
	}
```

```
        (void) nsc_free_buf(handle);
        handle = NULL;

        return;
failed:
        (void) nsc_free_buf(handle);
        syncinfo->status->offset = syncinfo->offset;
}

/*
 * _rdc_sync_wrthr
 * sync loop write thread
 * if there are avail threads, we have not
 * used up the pipe, so the sync loop will, if
 * possible use these to multithread the write/read
 */
void
_rdc_sync_thread(void *thrinfo)
{
        rdc_syncthr_t *syncinfo = (rdc_syncthr_t *)thrinfo;
        rdc_k_info_t *krdc = syncinfo->krdc;
        rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
        rdc_thrsync_t *sync = &krdc->syncs;
        uint_t bitmask;
        int rc;

        rc = _rdc_rsrv_devs(krdc, RDC_RAW, RDC_INTERNAL);
        if (!RDC_SUCCESS(rc))
                goto failed;

        if (IS_STATE(urdc, RDC_SLAVE))
                _rdc_sync_rdthr(thrinfo);
        else
                _rdc_sync_wrthr(thrinfo);

        _rdc_rlse_devs(krdc, RDC_RAW);

        if (krdc->dcio_bitmap == NULL) {
#ifdef DEBUG
                cmn_err(CE_NOTE, "!_rdc_sync_wrthr: NULL bitmap");
#else
        /*EMPTY*/
#endif
        } else if (syncinfo->status->offset < 0) {

                RDC_SET_BITMASK(syncinfo->offset, syncinfo->len, &bitmask);
                RDC_CLR_BITMAP(krdc, syncinfo->offset, syncinfo->len, \
                    bitmask, RDC_BIT_FORCE);
        }

failed:
        /*
         * done with this, get rid of it.
         * the status is not freed, it should still be a status chain
         * that _rdc_sync() has the head of
         */
        kmem_free(syncinfo, sizeof (*syncinfo));

        /*
         * decrement the global sync thread num
         */
        mutex_enter(&sync_info.lock);
        sync_info.active_thr--;
        /* LINTED */
        RDC_AVAIL_THR_TUNE(sync_info);
```

```
        mutex_exit(&sync_info.lock);

        /*
         * krdc specific stuff
         */
        mutex_enter(&sync->lock);
        sync->complete++;
        cv_broadcast(&sync->cv);
        mutex_exit(&sync->lock);
}

int
_rdc_setup_syncthr(rdc_syncthr_t **synthr, nsc_off_t offset,
    nsc_size_t len, rdc_k_info_t *krdc, sync_status_t *stats)
{
        rdc_syncthr_t *tmp;
        /* alloc here, free in the sync thread */
        tmp =
            (rdc_syncthr_t *)kmem_zalloc(sizeof (rdc_syncthr_t), KM_NOSLEEP);

        if (tmp == NULL)
                return (-1);
        tmp->offset = offset;
        tmp->len = len;
        tmp->status = stats;
        tmp->krdc = krdc;

        *synthr = tmp;
        return (0);
}

sync_status_t *
_rdc_new_sync_status()
{
        sync_status_t *s;

        s = (sync_status_t *)kmem_zalloc(sizeof (*s), KM_NOSLEEP);
        s->offset = -1;
        return (s);
}

void
_rdc_free_sync_status(sync_status_t *status)
{
        sync_status_t *s;

        while (status) {
                s = status->next;
                kmem_free(status, sizeof (*status));
                status = s;
        }
}
int
_rdc_sync_status_ok(sync_status_t *status, int *offset)
{
#ifdef DEBUG_SYNCSTATUS
        int i = 0;
#endif
        while (status) {
                if (status->offset >= 0) {
                        *offset = status->offset;
                        return (-1);
                }
                status = status->next;
#ifdef DEBUG_SYNCSTATUS
                i++;
```

```
#endif
        }
#ifdef DEBUGSYNCSTATUS
        cmn_err(CE_NOTE, "!rdc_sync_status_ok: checked %d statuses", i);
#endif
        return (0);
}

int mtsync = 1;
/*
 * _rdc_sync() : rdc sync loop
 *
 */
static void
_rdc_sync(rdc_k_info_t *krdc)
{
        nsc_size_t size = 0;
        rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
        int rtype;
        int sts;
        int reserved = 0;
        nsc_buf_t *alloc_h = NULL;
        nsc_buf_t *handle = NULL;
        nsc_off_t mask;
        nsc_size_t maxbit;
        nsc_size_t len;
        nsc_off_t offset = 0;
        int sync_completed = 0;
        int tries = 0;
        int rc;
        int queuing = 0;
        uint_t bitmask;
        sync_status_t *ss, *sync_status = NULL;
        rdc_thrsync_t *sync = &krdc->syncs;
        rdc_syncthr_t *syncinfo;
        nsthread_t *trc = NULL;

        if (IS_STATE(urdc, RDC_QUEUING) && !IS_STATE(urdc, RDC_FULL)) {
                /* flusher is handling the sync in the update case */
                queuing = 1;
                goto sync_done;
        }

        /*
         * Main sync/resync loop
         */
        DTRACE_PROBE(rdc_sync_loop_start);

        rtype = RDC_RAW;
        sts = _rdc_rsrv_devs(krdc, rtype, RDC_INTERNAL);

        DTRACE_PROBE(rdc_sync_loop_rsrv);

        if (sts != 0)
                goto failed_noincr;

        reserved = 1;

        /*
         * pre-allocate a handle if we can - speeds up the sync.
         */

        if (rdc_prealloc_handle) {
                alloc_h = nsc_alloc_handle(RDC_U_FD(krdc), NULL, NULL, NULL);
#ifdef DEBUG
                if (!alloc_h) {
```

```
                        cmn_err(CE_WARN,
                            "!rdc sync: failed to pre-alloc handle");
                }
#endif
        } else {
                alloc_h = NULL;
        }

        ASSERT(urdc->volume_size != 0);
        size = urdc->volume_size;
        mask = ~(LOG_TO_FBA_NUM(1) - 1);
        maxbit = FBA_TO_LOG_NUM(size - 1);

        /*
         * as this while loop can also move data, it is counted as a
         * sync loop thread
         */
        rdc_group_enter(krdc);
        rdc_clr_flags(urdc, RDC_LOGGING);
        rdc_set_flags(urdc, RDC_SYNCING);
        krdc->group->synccount++;
        rdc_group_exit(krdc);
        mutex_enter(&sync_info.lock);
        sync_info.active_thr++;
        /* LINTED */
        RDC_AVAIL_THR_TUNE(sync_info);
        mutex_exit(&sync_info.lock);

        while (offset < size) {
                rdc_group_enter(krdc);
                ASSERT(krdc->aux_state & RDC_AUXSYNCIP);
                if (krdc->disk_status == 1 || krdc->dcio_bitmap == NULL) {
                        rdc_group_exit(krdc);
                        if (krdc->disk_status == 1) {
                                DTRACE_PROBE(rdc_sync_loop_disk_status_err);
                        } else {
                                DTRACE_PROBE(rdc_sync_loop_dcio_bitmap_err);
                        }
                        goto failed;            /* halt sync */
                }
                rdc_group_exit(krdc);

                if (!(rdc_get_vflags(urdc) & RDC_FULL)) {
                        mutex_enter(&krdc->syncbitmutex);
                        krdc->syncbitpos = FBA_TO_LOG_NUM(offset);
                        len = 0;

                        /* skip unnecessary chunks */

                        while (krdc->syncbitpos <= maxbit &&
                            !RDC_BIT_ISSET(krdc, krdc->syncbitpos)) {
                                offset += LOG_TO_FBA_NUM(1);
                                krdc->syncbitpos++;
                        }

                        /* check for boundary */

                        if (offset >= size) {
                                mutex_exit(&krdc->syncbitmutex);
                                goto sync_done;
                        }

                        /* find maximal length we can transfer */

                        while (krdc->syncbitpos <= maxbit &&
                            RDC_BIT_ISSET(krdc, krdc->syncbitpos)) {
```

```
				len += LOG_TO_FBA_NUM(1);
				krdc->syncbitpos++;
				/* we can only read maxfbas anyways */
				if (len >= krdc->maxfbas)
					break;
			}

			len = min(len, (size - offset));

		} else {
			len = size - offset;
		}

		/* truncate to the io provider limit */
		ASSERT(krdc->maxfbas != 0);
		len = min(len, krdc->maxfbas);

		if (len > LOG_TO_FBA_NUM(1)) {
			/*
			 * If the update is larger than a bitmap chunk,
			 * then truncate to a whole number of bitmap
			 * chunks.
			 *
			 * If the update is smaller than a bitmap
			 * chunk, this must be the last write.
			 */
			len &= mask;
		}

		if (!(rdc_get_vflags(urdc) & RDC_FULL)) {
			krdc->syncbitpos = FBA_TO_LOG_NUM(offset + len);
			mutex_exit(&krdc->syncbitmutex);
		}

		/*
		 * Find out if we can reserve a thread here ...
		 * note: skip the mutex for the first check, if the number
		 * is up there, why bother even grabbing the mutex to
		 * only realize that we can't have a thread anyways
		 */

		if (mtsync && sync_info.active_thr < RDC_MAX_SYNC_THREADS) {

			mutex_enter(&sync_info.lock);
			if (sync_info.avail_thr >= 1) {
				if (sync_status == NULL) {
					ss = sync_status =
					    _rdc_new_sync_status();
				} else {
					ss = ss->next = _rdc_new_sync_status();
				}
				if (ss == NULL) {
					mutex_exit(&sync_info.lock);
#ifdef DEBUG
					cmn_err(CE_WARN, "!rdc_sync: can't "
					    "allocate status for mt sync");
#endif
					goto retry;
				}
				/*
				 * syncinfo protected by sync_info lock but
				 * not part of the sync_info structure
				 * be careful if moving
				 */
				if (_rdc_setup_syncthr(&syncinfo,
				    offset, len, krdc, ss) < 0) {
```

```
					_rdc_free_sync_status(ss);
				}

				trc = nst_create(sync_info.rdc_syncset,
				    _rdc_sync_thread, syncinfo, NST_SLEEP);

				if (trc == NULL) {
					mutex_exit(&sync_info.lock);
#ifdef DEBUG
					cmn_err(CE_NOTE, "!rdc_sync: unable to "
					    "mt sync");
#endif
					_rdc_free_sync_status(ss);
					kmem_free(syncinfo, sizeof (*syncinfo));
					syncinfo = NULL;
					goto retry;
				} else {
					mutex_enter(&sync->lock);
					sync->threads++;
					mutex_exit(&sync->lock);
				}

				sync_info.active_thr++;
				/* LINTED */
				RDC_AVAIL_THR_TUNE(sync_info);

				mutex_exit(&sync_info.lock);
				goto threaded;
			}
			mutex_exit(&sync_info.lock);
		}
retry:
		handle = alloc_h;
		DTRACE_PROBE(rdc_sync_loop_allocbuf_start);
		if (rdc_get_vflags(urdc) & RDC_SLAVE)
			sts = nsc_alloc_buf(RDC_U_FD(krdc), offset, len,
			    NSC_WRITE | NSC_WRTHRU | NSC_NOCACHE, &handle);
		else
			sts = nsc_alloc_buf(RDC_U_FD(krdc), offset, len,
			    NSC_READ | NSC_NOCACHE, &handle);

		DTRACE_PROBE(rdc_sync_loop_allocbuf_end);
		if (sts > 0) {
			if (handle && handle != alloc_h) {
				(void) nsc_free_buf(handle);
			}

			handle = NULL;
			DTRACE_PROBE(rdc_sync_loop_allocbuf_err);
			goto failed;
		}

		if (rdc_get_vflags(urdc) & RDC_SLAVE) {
			/* overwrite buffer with remote data */
			sts = rdc_net_read(krdc->index, krdc->remote_index,
			    handle, handle->sb_pos, handle->sb_len);

			if (!RDC_SUCCESS(sts)) {
#ifdef DEBUG
				cmn_err(CE_WARN,
				    "!rdc sync: remote read failed (%d)", sts);
#endif
				DTRACE_PROBE(rdc_sync_loop_remote_read_err);
				goto failed;
			}
			if (!(rdc_get_vflags(urdc) & RDC_FULL))
```

```
                                rdc_set_bitmap_many(krdc, handle->sb_pos,
                                    handle->sb_len);

                        /* commit locally */

                        sts = nsc_write(handle, handle->sb_pos,
                            handle->sb_len, 0);

                        if (!RDC_SUCCESS(sts)) {
                                /* reverse sync needed already set */
                                rdc_many_enter(krdc);
                                rdc_set_flags_log(urdc, RDC_VOL_FAILED,
                                    "write failed during sync");
                                rdc_many_exit(krdc);
                                rdc_write_state(urdc);
                                DTRACE_PROBE(rdc_sync_loop_nsc_write_err);
                                goto failed;
                        }
                } else {
                        /* send local data to remote */
                        DTRACE_PROBE2(rdc_sync_loop_netwrite_start,
                            int, krdc->index, nsc_buf_t *, handle);

                        if ((sts = rdc_net_write(krdc->index,
                            krdc->remote_index, handle, handle->sb_pos,
                            handle->sb_len, RDC_NOSEQ, RDC_NOQUE, NULL)) > 0) {

                                /*
                                 * The following is to handle
                                 * the case where the secondary side
                                 * has thrown our buffer handle token away in a
                                 * attempt to preserve its health on restart
                                 */
                                if ((sts == EPROTO) && (tries < 3)) {
                                        (void) nsc_free_buf(handle);
                                        handle = NULL;
                                        tries++;
                                        delay(HZ >> 2);
                                        goto retry;
                                }
#ifdef DEBUG
                                cmn_err(CE_WARN,
                                    "!rdc sync: remote write failed (%d) 0x%x",
                                    sts, rdc_get_vflags(urdc));
#endif
                                DTRACE_PROBE(rdc_sync_loop_netwrite_err);
                                goto failed;
                        }
                        DTRACE_PROBE(rdc_sync_loop_netwrite_end);
                }

                (void) nsc_free_buf(handle);
                handle = NULL;

                if (krdc->dcio_bitmap == NULL) {
#ifdef DEBUG
                        cmn_err(CE_NOTE, "!_rdc_sync: NULL bitmap");
#else
                ;
                /*EMPTY*/
#endif
                } else {

                        RDC_SET_BITMASK(offset, len, &bitmask);
                        RDC_CLR_BITMAP(krdc, offset, len, bitmask, \
                            RDC_BIT_FORCE);
```

```
                        ASSERT(!IS_ASYNC(urdc));
                }

                /*
                 * Only release/reserve if someone is waiting
                 */
                if (krdc->devices->id_release || nsc_waiting(RDC_U_FD(krdc))) {
                        DTRACE_PROBE(rdc_sync_loop_rlse_start);
                        if (alloc_h) {
                                (void) nsc_free_handle(alloc_h);
                                alloc_h = NULL;
                        }

                        _rdc_rlse_devs(krdc, rtype);
                        reserved = 0;
                        delay(2);

                        rtype = RDC_RAW;
                        sts = _rdc_rsrv_devs(krdc, rtype, RDC_INTERNAL);
                        if (sts != 0) {
                                handle = NULL;
                                DTRACE_PROBE(rdc_sync_loop_rdc_rsrv_err);
                                goto failed;
                        }

                        reserved = 1;

                        if (rdc_prealloc_handle) {
                                alloc_h = nsc_alloc_handle(RDC_U_FD(krdc),
                                    NULL, NULL, NULL);
#ifdef DEBUG
                                if (!alloc_h) {
                                        cmn_err(CE_WARN, "!rdc_sync: "
                                            "failed to pre-alloc handle");
                                }
#endif
                        }
                        DTRACE_PROBE(rdc_sync_loop_rlse_end);
                }
threaded:
                offset += len;
                urdc->sync_pos = offset;
        }

sync_done:
        sync_completed = 1;

failed:
        krdc->group->synccount--;
failed_noincr:
        mutex_enter(&sync->lock);
        while (sync->complete != sync->threads) {
                cv_wait(&sync->cv, &sync->lock);
        }
        sync->complete = 0;
        sync->threads = 0;
        mutex_exit(&sync->lock);

        /*
         * if sync_completed is 0 here,
         * we know that the main sync thread failed anyway
         * so just free the statuses and fail
         */
        if (sync_completed && (_rdc_sync_status_ok(sync_status, &rc) < 0)) {
                urdc->sync_pos = rc;
                sync_completed = 0; /* at least 1 thread failed */
```

```
	}

	_rdc_free_sync_status(sync_status);

	/*
	 * we didn’t increment, we didn’t even sync,
	 * so don’t dec sync_info.active_thr
	 */
	if (!queuing) {
		mutex_enter(&sync_info.lock);
		sync_info.active_thr--;
		/* LINTED */
		RDC_AVAIL_THR_TUNE(sync_info);
		mutex_exit(&sync_info.lock);
	}

	if (handle) {
		(void) nsc_free_buf(handle);
	}

	if (alloc_h) {
		(void) nsc_free_handle(alloc_h);
	}

	if (reserved) {
		_rdc_rlse_devs(krdc, rtype);
	}

notstarted:
	rdc_group_enter(krdc);
	ASSERT(krdc->aux_state & RDC_AUXSYNCIP);
	if (IS_STATE(urdc, RDC_QUEUING))
		rdc_clr_flags(urdc, RDC_QUEUING);

	if (sync_completed) {
		(void) rdc_net_state(krdc->index, CCIO_DONE);
	} else {
		(void) rdc_net_state(krdc->index, CCIO_ENABLELOG);
	}

	rdc_clr_flags(urdc, RDC_SYNCING);
	if (rdc_get_vflags(urdc) & RDC_SLAVE) {
		rdc_many_enter(krdc);
		rdc_clr_mflags(urdc, RDC_SLAVE);
		rdc_many_exit(krdc);
	}
	if (krdc->type_flag & RDC_ASYNCMODE)
		rdc_set_flags(urdc, RDC_ASYNC);
	if (sync_completed) {
		rdc_many_enter(krdc);
		rdc_clr_mflags(urdc, RDC_RSYNC_NEEDED);
		rdc_many_exit(krdc);
	} else {
		krdc->remote_index = -1;
		rdc_set_flags_log(urdc, RDC_LOGGING, "sync failed to complete");
	}
	rdc_group_exit(krdc);
	rdc_write_state(urdc);

	mutex_enter(&net_blk_lock);
	if (sync_completed)
		krdc->sync_done = RDC_COMPLETED;
	else
		krdc->sync_done = RDC_FAILED;
	cv_broadcast(&krdc->synccv);
	mutex_exit(&net_blk_lock);
```

```

}


static int
rdc_sync(rdc_config_t *uparms, spcs_s_info_t kstatus)
{
	rdc_set_t *rdc_set = uparms->rdc_set;
	int options = uparms->options;
	int rc = 0;
	int busy = 0;
	int index;
	rdc_k_info_t *krdc;
	rdc_u_info_t *urdc;
	rdc_k_info_t *kmulti;
	rdc_u_info_t *umulti;
	rdc_group_t *group;
	rdc_srv_t *svp;
	int sm, um, md;
	int sync_completed = 0;
	int thrcount;

	mutex_enter(&rdc_conf_lock);
	index = rdc_lookup_byname(rdc_set);
	if (index >= 0)
		krdc = &rdc_k_info[index];
	if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
		mutex_exit(&rdc_conf_lock);
		spcs_s_add(kstatus, RDC_EALREADY, rdc_set->primary.file,
		    rdc_set->secondary.file);
		rc = RDC_EALREADY;
		goto notstarted;
	}

	urdc = &rdc_u_info[index];
	group = krdc->group;
	set_busy(krdc);
	busy = 1;
	if ((krdc->type_flag == 0) || (krdc->type_flag & RDC_DISABLEPEND)) {
		/* A resume or enable failed  or we raced with a teardown */
		mutex_exit(&rdc_conf_lock);
		spcs_s_add(kstatus, RDC_EALREADY, rdc_set->primary.file,
		    rdc_set->secondary.file);
		rc = RDC_EALREADY;
		goto notstarted;
	}
	mutex_exit(&rdc_conf_lock);
	rdc_group_enter(krdc);

	if (!IS_STATE(urdc, RDC_LOGGING)) {
		spcs_s_add(kstatus, RDC_ESETNOTLOGGING, urdc->secondary.intf,
		    urdc->secondary.file);
		rc = RDC_ENOTLOGGING;
		goto notstarted_unlock;
	}

	if (rdc_check(krdc, rdc_set)) {
		spcs_s_add(kstatus, RDC_EALREADY, rdc_set->primary.file,
		    rdc_set->secondary.file);
		rc = RDC_EALREADY;
		goto notstarted_unlock;
	}

	if (!(rdc_get_vflags(urdc) & RDC_PRIMARY)) {
		spcs_s_add(kstatus, RDC_ENOTPRIMARY, rdc_set->primary.intf,
		    rdc_set->primary.file, rdc_set->secondary.intf,
			    rdc_set->secondary.file);
			rc = RDC_ENOTPRIMARY;
			goto notstarted_unlock;
		}

		if ((options & RDC_OPT_REVERSE) && (IS_STATE(urdc, RDC_QUEUING))) {
			/*
			 * cannot reverse sync when queuing, need to go logging first
			 */
			spcs_s_add(kstatus, RDC_EQNORSYNC, rdc_set->primary.intf,
			    rdc_set->primary.file, rdc_set->secondary.intf,
			    rdc_set->secondary.file);
			rc = RDC_EQNORSYNC;
			goto notstarted_unlock;
		}

		svp = krdc->lsrv;
		krdc->intf = rdc_add_to_if(svp, &(urdc->primary.addr),
		    &(urdc->secondary.addr), 1);

		if (!krdc->intf) {
			spcs_s_add(kstatus, RDC_EADDTOIF, urdc->primary.intf,
			    urdc->secondary.intf);
			rc = RDC_EADDTOIF;
			goto notstarted_unlock;
		}

		if (urdc->volume_size == 0) {
			/* Implies reserve failed when previous resume was done */
			rdc_get_details(krdc);
		}
		if (urdc->volume_size == 0) {
			spcs_s_add(kstatus, RDC_ENOBMAP);
			rc = RDC_ENOBMAP;
			goto notstarted_unlock;
		}

		if (krdc->dcio_bitmap == NULL) {
			if (rdc_resume_bitmap(krdc) < 0) {
				spcs_s_add(kstatus, RDC_ENOBMAP);
				rc = RDC_ENOBMAP;
				goto notstarted_unlock;
			}
		}

		if ((rdc_get_vflags(urdc) & RDC_BMP_FAILED) && (krdc->bitmapfd)) {
			if (rdc_reset_bitmap(krdc)) {
				spcs_s_add(kstatus, RDC_EBITMAP);
				rc = RDC_EBITMAP;
				goto notstarted_unlock;
			}
		}

		if (IS_MANY(krdc) || IS_MULTI(krdc)) {
			rdc_u_info_t *ubad;

			if ((ubad = rdc_allow_pri_sync(urdc, options)) != NULL) {
				spcs_s_add(kstatus, RDC_ESTATE,
				    ubad->primary.intf, ubad->primary.file,
				    ubad->secondary.intf, ubad->secondary.file);
				rc = RDC_ESTATE;
				goto notstarted_unlock;
			}
		}

		/*
```

```
		 * there is a small window where _rdc_sync is still
		 * running, but has cleared the RDC_SYNCING flag.
		 * Use aux_state which is only cleared
		 * after _rdc_sync had done its 'death' broadcast.
		 */
		if (krdc->aux_state & RDC_AUXSYNCIP) {
#ifdef DEBUG
			if (!rdc_get_vflags(urdc) & RDC_SYNCING) {
				cmn_err(CE_WARN, "!rdc_sync: "
				    "RDC_AUXSYNCIP set, SYNCING off");
			}
#endif
			spcs_s_add(kstatus, RDC_ESYNCING, rdc_set->primary.file);
			rc = RDC_ESYNCING;
			goto notstarted_unlock;
		}
		if (krdc->disk_status == 1) {
			spcs_s_add(kstatus, RDC_ESYNCING, rdc_set->primary.file);
			rc = RDC_ESYNCING;
			goto notstarted_unlock;
		}

		if ((options & RDC_OPT_FORWARD) &&
		    (rdc_get_mflags(urdc) & RDC_RSYNC_NEEDED)) {
			/* cannot forward sync if a reverse sync is needed */
			spcs_s_add(kstatus, RDC_ERSYNCNEEDED, rdc_set->primary.intf,
			    rdc_set->primary.file, rdc_set->secondary.intf,
			    rdc_set->secondary.file);
			rc = RDC_ERSYNCNEEDED;
			goto notstarted_unlock;
		}

		urdc->sync_pos = 0;

		/* Check if the rdc set is accessible on the remote node */
		if (rdc_net_getstate(krdc, &sm, &um, &md, FALSE) < 0) {
			/*
			 * Remote end may be inaccessible, or the rdc set is not
			 * enabled at the remote end.
			 */
			spcs_s_add(kstatus, RDC_ECONNOPEN, urdc->secondary.intf,
			    urdc->secondary.file);
			rc = RDC_ECONNOPEN;
			goto notstarted_unlock;
		}
		if (options & RDC_OPT_REVERSE)
			krdc->remote_index = rdc_net_state(index, CCIO_RSYNC);
		else
			krdc->remote_index = rdc_net_state(index, CCIO_SLAVE);
		if (krdc->remote_index < 0) {
			/*
			 * Remote note probably not in a valid state to be synced,
			 * as the state was fetched OK above.
			 */
			spcs_s_add(kstatus, RDC_ERSTATE, urdc->secondary.intf,
			    urdc->secondary.file, urdc->primary.intf,
			    urdc->primary.file);
			rc = RDC_ERSTATE;
			goto notstarted_unlock;
		}

		rc = check_filesize(index, kstatus);
		if (rc != 0) {
			(void) rdc_net_state(krdc->index, CCIO_ENABLELOG);
			goto notstarted_unlock;
		}
```

```
        krdc->sync_done = 0;

        mutex_enter(&krdc->bmapmutex);
        krdc->aux_state |= RDC_AUXSYNCIP;
        mutex_exit(&krdc->bmapmutex);

        if (options & RDC_OPT_REVERSE) {
                rdc_many_enter(krdc);
                rdc_set_mflags(urdc, RDC_SLAVE | RDC_RSYNC_NEEDED);
                mutex_enter(&krdc->bmapmutex);
                rdc_clr_flags(urdc, RDC_VOL_FAILED);
                mutex_exit(&krdc->bmapmutex);
                rdc_write_state(urdc);
                /* LINTED */
                if (kmulti = krdc->multi_next) {
                        umulti = &rdc_u_info[kmulti->index];
                        if (IS_ENABLED(umulti) && (rdc_get_vflags(umulti) &
                            (RDC_VOL_FAILED | RDC_SYNC_NEEDED))) {
                                rdc_clr_flags(umulti, RDC_SYNC_NEEDED);
                                rdc_clr_flags(umulti, RDC_VOL_FAILED);
                                rdc_write_state(umulti);
                        }
                }
                rdc_many_exit(krdc);
        } else {
                rdc_clr_flags(urdc, RDC_FCAL_FAILED);
                rdc_write_state(urdc);
        }

        if (options & RDC_OPT_UPDATE) {
                ASSERT(urdc->volume_size != 0);
                if (rdc_net_getbmap(index,
                    BMAP_LOG_BYTES(urdc->volume_size)) > 0) {
                        spcs_s_add(kstatus, RDC_ENOBMAP);
                        rc = RDC_ENOBMAP;

                        (void) rdc_net_state(index, CCIO_ENABLELOG);

                        rdc_clr_flags(urdc, RDC_SYNCING);
                        if (options & RDC_OPT_REVERSE) {
                                rdc_many_enter(krdc);
                                rdc_clr_mflags(urdc, RDC_SLAVE);
                                rdc_many_exit(krdc);
                        }
                        if (krdc->type_flag & RDC_ASYNCMODE)
                                rdc_set_flags(urdc, RDC_ASYNC);
                        krdc->remote_index = -1;
                        rdc_set_flags_log(urdc, RDC_LOGGING,
                            "failed to read remote bitmap");
                        rdc_write_state(urdc);
                        goto failed;
                }
                rdc_clr_flags(urdc, RDC_FULL);
        } else {
                /*
                 * This is a full sync (not an update sync), mark the
                 * entire bitmap dirty
                 */
                (void) RDC_FILL_BITMAP(krdc, FALSE);

                rdc_set_flags(urdc, RDC_FULL);
        }

        rdc_group_exit(krdc);
```

```
        /*
         * allow diskq->memq flusher to wake up
         */
        mutex_enter(&krdc->group->ra_queue.net_qlock);
        krdc->group->ra_queue.qfflags &= ~RDC_QFILLSLEEP;
        mutex_exit(&krdc->group->ra_queue.net_qlock);

        /*
         * if this is a full sync on a non-diskq set or
         * a diskq set that has failed, clear the async flag
         */
        if (krdc->type_flag & RDC_ASYNCMODE) {
                if ((!(options & RDC_OPT_UPDATE)) ||
                    (!RDC_IS_DISKQ(krdc->group)) ||
                    (!(IS_STATE(urdc, RDC_QUEUING)))) {
                        /* full syncs, or core queue are synchronous */
                        rdc_group_enter(krdc);
                        rdc_clr_flags(urdc, RDC_ASYNC);
                        rdc_group_exit(krdc);
                }

                /*
                 * if the queue failed because it was full, lets see
                 * if we can restart it. After _rdc_sync() is done
                 * the modes will switch and we will begin disk
                 * queuing again. NOTE: this should only be called
                 * once per group, as it clears state for all group
                 * members, also clears the async flag for all members
                 */
                if (IS_STATE(urdc, RDC_DISKQ_FAILED)) {
                        rdc_unfail_diskq(krdc);
                } else {
                /* don’t add insult to injury by flushing a dead queue */

                        /*
                         * if we are updating, and a diskq and
                         * the async thread isn’t active, start
                         * it up.
                         */
                        if ((options & RDC_OPT_UPDATE) &&
                            (IS_STATE(urdc, RDC_QUEUING))) {
                                rdc_group_enter(krdc);
                                rdc_clr_flags(urdc, RDC_SYNCING);
                                rdc_group_exit(krdc);
                                mutex_enter(&krdc->group->ra_queue.net_qlock);
                                if (krdc->group->ra_queue.qfill_sleeping ==
                                    RDC_QFILL_ASLEEP)
                                        cv_broadcast(&group->ra_queue.qfcv);
                                mutex_exit(&krdc->group->ra_queue.net_qlock);
                                thrcount = urdc->asyncthr;
                                while ((thrcount-- > 0) &&
                                    !krdc->group->rdc_writer) {
                                        (void) rdc_writer(krdc->index);
                                }
                        }
                }
        }

        /*
         * For a reverse sync, merge the current bitmap with all other sets
         * that share this volume.
         */
        if (options & RDC_OPT_REVERSE) {
retry_many:
                rdc_many_enter(krdc);
                if (IS_MANY(krdc)) {
```

```
			rdc_k_info_t *kmany;
			rdc_u_info_t *umany;

			for (kmany = krdc->many_next; kmany != krdc;
			    kmany = kmany->many_next) {
				umany = &rdc_u_info[kmany->index];
				if (!IS_ENABLED(umany))
					continue;
				ASSERT(umany->flags & RDC_PRIMARY);

				if (!mutex_tryenter(&kmany->group->lock)) {
					rdc_many_exit(krdc);
					/* May merge more than once */
					goto retry_many;
				}
				rdc_merge_bitmaps(krdc, kmany);
				mutex_exit(&kmany->group->lock);
			}
		}
		rdc_many_exit(krdc);

retry_multi:
		rdc_many_enter(krdc);
		if (IS_MULTI(krdc)) {
			rdc_k_info_t *kmulti = krdc->multi_next;
			rdc_u_info_t *umulti = &rdc_u_info[kmulti->index];

			if (IS_ENABLED(umulti)) {
				ASSERT(!(umulti->flags & RDC_PRIMARY));

				if (!mutex_tryenter(&kmulti->group->lock)) {
					rdc_many_exit(krdc);
					goto retry_multi;
				}
				rdc_merge_bitmaps(krdc, kmulti);
				mutex_exit(&kmulti->group->lock);
			}
		}
		rdc_many_exit(krdc);
	}

	rdc_group_enter(krdc);

	if (krdc->bitmap_write == 0) {
		if (rdc_write_bitmap_fill(krdc) >= 0)
			krdc->bitmap_write = -1;
	}

	if (krdc->bitmap_write > 0)
		(void) rdc_write_bitmap(krdc);

	urdc->bits_set = RDC_COUNT_BITMAP(krdc);

	rdc_group_exit(krdc);

	if (options & RDC_OPT_REVERSE) {
		(void) _rdc_sync_event_notify(RDC_SYNC_START,
		    urdc->primary.file, urdc->group_name);
	}

	/* Now set off the sync itself */

	mutex_enter(&net_blk_lock);
	if (nsc_create_process(
	    (void (*)(void *))_rdc_sync, (void *)krdc, FALSE)) {
		mutex_exit(&net_blk_lock);
```

```
		spcs_s_add(kstatus, RDC_ENOPROC);
		/*
		 * We used to just return here,
		 * but we need to clear the AUXSYNCIP bit
		 * and there is a very small chance that
		 * someone may be waiting on the disk_status flag.
		 */
		rc = RDC_ENOPROC;
		/*
		 * need the group lock held at failed.
		 */
		rdc_group_enter(krdc);
		goto failed;
	}

	mutex_enter(&rdc_conf_lock);
	wakeup_busy(krdc);
	busy = 0;
	mutex_exit(&rdc_conf_lock);

	while (krdc->sync_done == 0)
		cv_wait(&krdc->synccv, &net_blk_lock);
	mutex_exit(&net_blk_lock);

	rdc_group_enter(krdc);

	if (krdc->sync_done == RDC_FAILED) {
		char siztmp1[16];
		(void) spcs_s_inttostring(
		    urdc->sync_pos, siztmp1, sizeof (siztmp1),
		    0);
		spcs_s_add(kstatus, RDC_EFAIL, siztmp1);
		rc = RDC_EFAIL;
	} else
		sync_completed = 1;

failed:
	/*
	 * We use this flag now to make halt_sync() wait for
	 * us to terminate and let us take the group lock.
	 */
	krdc->aux_state &= ~RDC_AUXSYNCIP;
	if (krdc->disk_status == 1) {
		krdc->disk_status = 0;
		cv_broadcast(&krdc->haltcv);
	}

notstarted_unlock:
	rdc_group_exit(krdc);

	if (sync_completed && (options & RDC_OPT_REVERSE)) {
		(void) _rdc_sync_event_notify(RDC_SYNC_DONE,
		    urdc->primary.file, urdc->group_name);
	}

notstarted:
	if (busy) {
		mutex_enter(&rdc_conf_lock);
		wakeup_busy(krdc);
		mutex_exit(&rdc_conf_lock);
	}

	return (rc);
}

/* ARGSUSED */
```

```
static int
_rdc_suspend(rdc_k_info_t *krdc, rdc_set_t *rdc_set, spcs_s_info_t kstatus)
{
        rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
        rdc_if_t *ip;
        int index = krdc->index;

        ASSERT(krdc->group != NULL);
        rdc_group_enter(krdc);
#ifdef DEBUG
        ASSERT(rdc_check(krdc, rdc_set) == 0);
#else
        if (rdc_check(krdc, rdc_set)) {
                rdc_group_exit(krdc);
                spcs_s_add(kstatus, RDC_EALREADY, rdc_set->primary.file,
                    rdc_set->secondary.file);
                return (RDC_EALREADY);
        }
#endif

        if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
                halt_sync(krdc);
                ASSERT(IS_ENABLED(urdc));
        }

        rdc_group_exit(krdc);
        (void) rdc_unintercept(krdc);

#ifdef DEBUG
        cmn_err(CE_NOTE, "!SNDR: suspended %s %s", urdc->primary.file,
            urdc->secondary.file);
#endif

        /* Configured but not enabled */
        ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));


        if (IS_ASYNC(urdc) && !RDC_IS_DISKQ(krdc->group)) {
                int tries = 2; /* in case of possibly stuck flusher threads */
#ifdef DEBUG
                net_queue *qp = &krdc->group->ra_queue;
#endif
                do {
                        if (!krdc->group->rdc_writer)
                                (void) rdc_writer(krdc->index);

                        (void) rdc_drain_queue(krdc->index);

                } while (krdc->group->rdc_writer && tries--);

                /* ok, force it to happen... */
                if (rdc_drain_queue(krdc->index) != 0) {
                        do {
                                mutex_enter(&krdc->group->ra_queue.net_qlock);
                                krdc->group->asyncdis = 1;
                                cv_broadcast(&krdc->group->asyncqcv);
                                mutex_exit(&krdc->group->ra_queue.net_qlock);
                                cmn_err(CE_WARN,
                                    "!SNDR: async I/O pending and not flushed "
                                    "for %s during suspend",
                                    urdc->primary.file);
#ifdef DEBUG
                                cmn_err(CE_WARN,
                                    "!nitems: %" NSC_SZFMT " nblocks: %"
                                    NSC_SZFMT " head: 0x%p tail: 0x%p",
                                    qp->nitems, qp->blocks,
```

```
                                    (void *)qp->net_qhead,
                                    (void *)qp->net_qtail);
#endif
                        } while (krdc->group->rdc_thrnum > 0);
                }
        }

        mutex_enter(&rdc_conf_lock);
        ip = krdc->intf;
        krdc->intf = 0;

        if (ip) {
                rdc_remove_from_if(ip);
        }

        mutex_exit(&rdc_conf_lock);

        rdc_group_enter(krdc);

        /* Configured but not enabled */
        ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

        rdc_group_exit(krdc);
        /* Must not hold group lock during this function */
        while (rdc_dump_alloc_bufs_cd(krdc->index) == EAGAIN)
                delay(2);
        rdc_group_enter(krdc);

        /* Don't rdc_clear_state, unlike _rdc_disable */

        rdc_free_bitmap(krdc, RDC_CMD_SUSPEND);
        rdc_close_bitmap(krdc);

        rdc_dev_close(krdc);
        rdc_close_direct(krdc);

        /* Configured but not enabled */
        ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

        rdc_group_exit(krdc);

        /*
         * we should now unregister the queue, with no conflicting
         * locks held. This is the last(only) member of the group
         */
        if (krdc->group && RDC_IS_DISKQ(krdc->group) &&
            krdc->group->count == 1) { /* stop protecting queue */
                rdc_unintercept_diskq(krdc->group);
        }

        mutex_enter(&rdc_conf_lock);

        /* Configured but not enabled */
        ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

        wait_busy(krdc);

        if (IS_MANY(krdc) || IS_MULTI(krdc))
                remove_from_many(krdc);

        remove_from_group(krdc);

        krdc->remote_index = -1;
        ASSERT(krdc->type_flag & RDC_CONFIGURED);
        ASSERT(krdc->type_flag & RDC_DISABLEPEND);
        krdc->type_flag = 0;
```

```
#ifdef  DEBUG
        if (krdc->dcio_bitmap)
                cmn_err(CE_WARN, "!_rdc_suspend: possible mem leak, "
                    "dcio_bitmap");
#endif
        krdc->dcio_bitmap = NULL;
        krdc->bitmap_ref = NULL;
        krdc->bitmap_size = 0;
        krdc->maxfbas = 0;
        krdc->bitmap_write = 0;
        krdc->disk_status = 0;
        rdc_destroy_svinfo(krdc->lsrv);
        krdc->lsrv = NULL;
        krdc->multi_next = NULL;

        rdc_u_init(urdc);

        mutex_exit(&rdc_conf_lock);
        rdc_kstat_delete(index);
        return (0);
}

static int
rdc_suspend(rdc_config_t *uparms, spcs_s_info_t kstatus)
{
        rdc_k_info_t *krdc;
        int index;
        int rc;

        mutex_enter(&rdc_conf_lock);

        index = rdc_lookup_byname(uparms->rdc_set);
        if (index >= 0)
                krdc = &rdc_k_info[index];
        if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }

        krdc->type_flag |= RDC_DISABLEPEND;
        wait_busy(krdc);
        if (krdc->type_flag == 0) {
                /* A resume or enable failed */
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }
        mutex_exit(&rdc_conf_lock);

        rc = _rdc_suspend(krdc, uparms->rdc_set, kstatus);
        return (rc);
}

static int
_rdc_resume(rdc_set_t *rdc_set, int options, spcs_s_info_t kstatus)
{
        int index;
        char *rhost;
        struct netbuf *addrp;
        rdc_k_info_t *krdc;
        rdc_u_info_t *urdc;
        rdc_srv_t *svp = NULL;
        char *local_file;
```

```
        char *local_bitmap;
        int rc, rc1;
        nsc_size_t maxfbas;
        rdc_group_t *grp;

        if ((rdc_set->primary.intf[0] == 0) ||
            (rdc_set->primary.addr.len == 0) ||
            (rdc_set->primary.file[0] == 0) ||
            (rdc_set->primary.bitmap[0] == 0) ||
            (rdc_set->secondary.intf[0] == 0) ||
            (rdc_set->secondary.addr.len == 0) ||
            (rdc_set->secondary.file[0] == 0) ||
            (rdc_set->secondary.bitmap[0] == 0)) {
                spcs_s_add(kstatus, RDC_EEMPTY);
                return (RDC_EEMPTY);
        }

        /* Next check there aren’t any enabled rdc sets which match. */

        mutex_enter(&rdc_conf_lock);

        if (rdc_lookup_byname(rdc_set) >= 0) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EENABLED, rdc_set->primary.intf,
                    rdc_set->primary.file, rdc_set->secondary.intf,
                    rdc_set->secondary.file);
                return (RDC_EENABLED);
        }

        if (rdc_lookup_many2one(rdc_set) >= 0) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EMANY2ONE, rdc_set->primary.intf,
                    rdc_set->primary.file, rdc_set->secondary.intf,
                    rdc_set->secondary.file);
                return (RDC_EMANY2ONE);
        }

        if (rdc_set->netconfig->knc_proto == NULL) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_ENETCONFIG);
                return (RDC_ENETCONFIG);
        }

        if (rdc_set->primary.addr.len == 0) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_ENETBUF, rdc_set->primary.file);
                return (RDC_ENETBUF);
        }

        if (rdc_set->secondary.addr.len == 0) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_ENETBUF, rdc_set->secondary.file);
                return (RDC_ENETBUF);
        }

        /* Check that the local data volume isn’t in use as a bitmap */
        if (options & RDC_OPT_PRIMARY)
                local_file = rdc_set->primary.file;
        else
                local_file = rdc_set->secondary.file;
        if (rdc_lookup_bitmap(local_file) >= 0) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EVOLINUSE, local_file);
                return (RDC_EVOLINUSE);
        }
        /* check that the secondary data volume isn't in use */
        if (!(options & RDC_OPT_PRIMARY)) {
                local_file = rdc_set->secondary.file;
                if (rdc_lookup_secondary(local_file) >= 0) {
                        mutex_exit(&rdc_conf_lock);
                        spcs_s_add(kstatus, RDC_EVOLINUSE, local_file);
                        return (RDC_EVOLINUSE);
                }
        }

        /* Check that the bitmap isn't in use as a data volume */
        if (options & RDC_OPT_PRIMARY)
                local_bitmap = rdc_set->primary.bitmap;
        else
                local_bitmap = rdc_set->secondary.bitmap;
        if (rdc_lookup_configured(local_bitmap) >= 0) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EBMPINUSE, local_bitmap);
                return (RDC_EBMPINUSE);
        }

        /* Check that the bitmap isn't already in use as a bitmap */
        if (rdc_lookup_bitmap(local_bitmap) >= 0) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EBMPINUSE, local_bitmap);
                return (RDC_EBMPINUSE);
        }

        /* Set urdc->volume_size */
        index = rdc_dev_open(rdc_set, options);
        if (index < 0) {
                mutex_exit(&rdc_conf_lock);
                if (options & RDC_OPT_PRIMARY)
                        spcs_s_add(kstatus, RDC_EOPEN, rdc_set->primary.intf,
                            rdc_set->primary.file);
                else
                        spcs_s_add(kstatus, RDC_EOPEN, rdc_set->secondary.intf,
                            rdc_set->secondary.file);
                return (RDC_EOPEN);
        }

        urdc = &rdc_u_info[index];
        krdc = &rdc_k_info[index];

        /* copy relevant parts of rdc_set to urdc field by field */

        (void) strncpy(urdc->primary.intf, rdc_set->primary.intf,
            MAX_RDC_HOST_SIZE);
        (void) strncpy(urdc->secondary.intf, rdc_set->secondary.intf,
            MAX_RDC_HOST_SIZE);

        (void) strncpy(urdc->group_name, rdc_set->group_name, NSC_MAXPATH);

        dup_rdc_netbuf(&rdc_set->primary.addr, &urdc->primary.addr);
        (void) strncpy(urdc->primary.file, rdc_set->primary.file, NSC_MAXPATH);
        (void) strncpy(urdc->primary.bitmap, rdc_set->primary.bitmap,
            NSC_MAXPATH);

        dup_rdc_netbuf(&rdc_set->secondary.addr, &urdc->secondary.addr);
        (void) strncpy(urdc->secondary.file, rdc_set->secondary.file,
            NSC_MAXPATH);
        (void) strncpy(urdc->secondary.bitmap, rdc_set->secondary.bitmap,
            NSC_MAXPATH);
        (void) strncpy(urdc->disk_queue, rdc_set->disk_queue, NSC_MAXPATH);
        urdc->setid = rdc_set->setid;
```

```
        if ((options & RDC_OPT_SYNC) && urdc->disk_queue[0]) {
                mutex_exit(&rdc_conf_lock);
                rdc_dev_close(krdc);
                spcs_s_add(kstatus, RDC_EQWRONGMODE);
                return (RDC_EQWRONGMODE);
        }

        /*
         * init flags now so that state left by failures in add_to_group()
         * are preserved.
         */
        rdc_init_flags(urdc);

        if ((rc1 = add_to_group(krdc, options, RDC_CMD_RESUME)) != 0) {
                if (rc1 == RDC_EQNOADD) { /* something went wrong with queue */
                        rdc_fail_diskq(krdc, RDC_WAIT, RDC_NOLOG);
                        /* don't return a failure here, continue with resume */

                } else { /* some other group add failure */
                        mutex_exit(&rdc_conf_lock);
                        rdc_dev_close(krdc);
                        spcs_s_add(kstatus, RDC_EGROUP,
                            rdc_set->primary.intf, rdc_set->primary.file,
                            rdc_set->secondary.intf, rdc_set->secondary.file,
                            rdc_set->group_name);
                        return (RDC_EGROUP);
                }
        }

        /*
         * maxfbas was set in rdc_dev_open as primary's maxfbas.
         * If diskq's maxfbas is smaller, then use diskq's.
         */
        grp = krdc->group;
        if (grp && RDC_IS_DISKQ(grp) && (grp->diskqfd != 0)) {
                rc = _rdc_rsrv_diskq(grp);
                if (RDC_SUCCESS(rc)) {
                        rc = nsc_maxfbas(grp->diskqfd, 0, &maxfbas);
                        if (rc == 0) {
#ifdef DEBUG
                                if (krdc->maxfbas != maxfbas)
                                        cmn_err(CE_NOTE,
                                            "!_rdc_resume: diskq maxfbas = %"
                                            NSC_SZFMT ", primary maxfbas = %"
                                            NSC_SZFMT, maxfbas, krdc->maxfbas);
#endif
                                        krdc->maxfbas = min(krdc->maxfbas,
                                            maxfbas);
                        } else {
                                cmn_err(CE_WARN,
                                    "!_rdc_resume: diskq maxfbas failed (%d)",
                                    rc);
                        }
                        _rdc_rlse_diskq(grp);
                } else {
                        cmn_err(CE_WARN,
                            "!_rdc_resume: diskq reserve failed (%d)", rc);
                }
        }

        (void) strncpy(urdc->direct_file, rdc_set->direct_file, NSC_MAXPATH);
        if ((options & RDC_OPT_PRIMARY) && rdc_set->direct_file[0]) {
                if (rdc_open_direct(krdc) == NULL)
                        rdc_set_flags(urdc, RDC_FCAL_FAILED);
        }
```

```
	krdc->many_next = krdc;

	ASSERT(krdc->type_flag == 0);
	krdc->type_flag = RDC_CONFIGURED;

	if (options & RDC_OPT_PRIMARY)
		rdc_set_flags(urdc, RDC_PRIMARY);

	if (options & RDC_OPT_ASYNC)
		krdc->type_flag |= RDC_ASYNCMODE;

	set_busy(krdc);

	urdc->syshostid = rdc_set->syshostid;

	if (add_to_many(krdc) < 0) {
		mutex_exit(&rdc_conf_lock);

		rdc_group_enter(krdc);

		spcs_s_add(kstatus, RDC_EMULTI);
		rc = RDC_EMULTI;
		goto fail;
	}

	/* Configured but not enabled */
	ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

	mutex_exit(&rdc_conf_lock);

	if (urdc->volume_size == 0) {
		rdc_many_enter(krdc);
		if (options & RDC_OPT_PRIMARY)
			rdc_set_mflags(urdc, RDC_RSYNC_NEEDED);
		else
			rdc_set_flags(urdc, RDC_SYNC_NEEDED);
		rdc_set_flags(urdc, RDC_VOL_FAILED);
		rdc_many_exit(krdc);
	}

	rdc_group_enter(krdc);

	/* Configured but not enabled */
	ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

	/*
	 * The rdc set is configured but not yet enabled. Other operations must
	 * ignore this set until it is enabled.
	 */

	urdc->sync_pos = 0;

	/* Set tunable defaults, we’ll pick up tunables from the header later */

	urdc->maxqfbas = rdc_maxthres_queue;
	urdc->maxqitems = rdc_max_qitems;
	urdc->autosync = 0;
	urdc->asyncthr = rdc_asyncthr;

	urdc->netconfig = rdc_set->netconfig;

	if (options & RDC_OPT_PRIMARY) {
		rhost = rdc_set->secondary.intf;
		addrp = &rdc_set->secondary.addr;
	} else {
		rhost = rdc_set->primary.intf;
```

```
		addrp = &rdc_set->primary.addr;
	}

	if (options & RDC_OPT_ASYNC)
		rdc_set_flags(urdc, RDC_ASYNC);

	svp = rdc_create_svinfo(rhost, addrp, urdc->netconfig);
	if (svp == NULL) {
		spcs_s_add(kstatus, ENOMEM);
		rc = ENOMEM;
		goto fail;
	}

	urdc->netconfig = NULL;		/* This will be no good soon */

	/* Don’t set krdc->intf here */
	rdc_kstat_create(index);

	/* if the bitmap resume isn’t clean, it will clear queuing flag */

	(void) rdc_resume_bitmap(krdc);

	if (RDC_IS_DISKQ(krdc->group)) {
		disk_queue *q = &krdc->group->diskq;
		if ((rc1 == RDC_EQNOADD) ||
		    IS_QSTATE(q, RDC_QBADRESUME)) {
			rdc_clr_flags(urdc, RDC_QUEUING);
			RDC_ZERO_BITREF(krdc);
		}
	}

	if (krdc->lsrv == NULL)
		krdc->lsrv = svp;
	else {
#ifdef DEBUG
		cmn_err(CE_WARN, "!_rdc_resume: krdc->lsrv already set: %p",
		    (void *) krdc->lsrv);
#endif
		rdc_destroy_svinfo(svp);
	}
	svp = NULL;

	/* Configured but not enabled */
	ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

	/* And finally */

	krdc->remote_index = -1;

	/* Should we set the whole group logging? */
	rdc_set_flags(urdc, RDC_ENABLED | RDC_LOGGING);

	rdc_group_exit(krdc);

	if (rdc_intercept(krdc) != 0) {
		rdc_group_enter(krdc);
		rdc_clr_flags(urdc, RDC_ENABLED);
		if (options & RDC_OPT_PRIMARY)
			spcs_s_add(kstatus, RDC_EREGISTER, urdc->primary.file);
		else
			spcs_s_add(kstatus, RDC_EREGISTER,
			    urdc->secondary.file);
#ifdef DEBUG
		cmn_err(CE_NOTE, "!nsc_register_path failed %s",
		    urdc->primary.file);
#endif
```

```
                rc = RDC_EREGISTER;
                goto bmpfail;
        }
#ifdef DEBUG
        cmn_err(CE_NOTE, "!SNDR: resumed %s %s", urdc->primary.file,
            urdc->secondary.file);
#endif

        rdc_write_state(urdc);

        mutex_enter(&rdc_conf_lock);
        wakeup_busy(krdc);
        mutex_exit(&rdc_conf_lock);

        return (0);

bmpfail:
        if (options & RDC_OPT_PRIMARY)
                spcs_s_add(kstatus, RDC_EBITMAP, urdc->primary.bitmap);
        else
                spcs_s_add(kstatus, RDC_EBITMAP, urdc->secondary.bitmap);
        rc = RDC_EBITMAP;
        if (rdc_get_vflags(urdc) & RDC_ENABLED) {
                rdc_group_exit(krdc);
                (void) rdc_unintercept(krdc);
                rdc_group_enter(krdc);
        }

fail:
        rdc_kstat_delete(index);
        /* Don’t unset krdc->intf here, unlike _rdc_enable */

        /* Configured but not enabled */
        ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

        rdc_dev_close(krdc);
        rdc_close_direct(krdc);
        rdc_destroy_svinfo(svp);

        /* Configured but not enabled */
        ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

        rdc_group_exit(krdc);

        mutex_enter(&rdc_conf_lock);

        /* Configured but not enabled */
        ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

        remove_from_group(krdc);

        if (IS_MANY(krdc) || IS_MULTI(krdc))
                remove_from_many(krdc);

        rdc_u_init(urdc);

        ASSERT(krdc->type_flag & RDC_CONFIGURED);
        krdc->type_flag = 0;
        wakeup_busy(krdc);

        mutex_exit(&rdc_conf_lock);

        return (rc);
}

static int
```

```
rdc_resume(rdc_config_t *uparms, spcs_s_info_t kstatus)
{
        char itmp[10];
        int rc;

        if (!(uparms->options & RDC_OPT_SYNC) &&
            !(uparms->options & RDC_OPT_ASYNC)) {
                (void) spcs_s_inttostring(
                    uparms->options, itmp, sizeof (itmp), 1);
                spcs_s_add(kstatus, RDC_EEINVAL, itmp);
                rc = RDC_EEINVAL;
                goto done;
        }

        if (!(uparms->options & RDC_OPT_PRIMARY) &&
            !(uparms->options & RDC_OPT_SECONDARY)) {
                (void) spcs_s_inttostring(
                    uparms->options, itmp, sizeof (itmp), 1);
                spcs_s_add(kstatus, RDC_EEINVAL, itmp);
                rc = RDC_EEINVAL;
                goto done;
        }

        rc = _rdc_resume(uparms->rdc_set, uparms->options, kstatus);
done:
        return (rc);
}

/*
 * if rdc_group_log is called because a volume has failed,
 * we must disgard the queue to preserve write ordering.
 * later perhaps, we can keep queuing, but we would have to
 * rewrite the i/o path to acommodate that. currently, if there
 * is a volume failure, the buffers are satisfied remotely and
 * there is no way to satisfy them from the current diskq config
 * phew, if we do that.. it will be difficult
 */
int
rdc_can_queue(rdc_k_info_t *krdc)
{
        rdc_k_info_t *p;
        rdc_u_info_t *q;

        for (p = krdc->group_next; ; p = p->group_next) {
                q = &rdc_u_info[p->index];
                if (IS_STATE(q, RDC_VOL_FAILED))
                        return (0);
                if (p == krdc)
                        break;
        }
        return (1);
}

/*
 * wait here, until all in flight async i/o’s have either
 * finished or failed. Avoid the race with r_net_state()
 * which tells remote end to log.
 */
void
rdc_inflwait(rdc_group_t *grp)
{
        int bail = RDC_CLNT_TMOUT * 2; /* to include retries */
        volatile int *inflitems;

        if (RDC_IS_DISKQ(grp))
                inflitems = (&(grp->diskq.inflitems));
        else
                inflitems = (&(grp->ra_queue.inflitems));

        while (*inflitems && (--bail > 0))
                delay(HZ);
}

void
rdc_group_log(rdc_k_info_t *krdc, int flag, char *why)
{
        rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
        rdc_k_info_t *p;
        rdc_u_info_t *q;
        int do_group;
        int sm, um, md;
        disk_queue *dq;

        void (*flag_op)(rdc_u_info_t *urdc, int flag);

        ASSERT(MUTEX_HELD(&krdc->group->lock));

        if (!IS_ENABLED(urdc))
                return;

        rdc_many_enter(krdc);

        if ((flag & RDC_QUEUING) && (!IS_STATE(urdc, RDC_SYNCING)) &&
            (rdc_can_queue(krdc))) {
                flag_op = rdc_set_flags; /* keep queuing, link error */
                flag &= ~RDC_FLUSH;
        } else {
                flag_op = rdc_clr_flags; /* stop queuing, user request */
        }

        do_group = 1;
        if (!(rdc_get_vflags(urdc) & RDC_PRIMARY))
                do_group = 0;
        else if ((urdc->group_name[0] == 0) ||
            (rdc_get_vflags(urdc) & RDC_LOGGING) ||
            (rdc_get_vflags(urdc) & RDC_SYNCING))
                do_group = 0;
        if (do_group) {
                for (p = krdc->group_next; p != krdc; p = p->group_next) {
                        q = &rdc_u_info[p->index];
                        if (!IS_ENABLED(q))
                                continue;
                        if ((rdc_get_vflags(q) & RDC_LOGGING) ||
                            (rdc_get_vflags(q) & RDC_SYNCING)) {
                                do_group = 0;
                                break;
                        }
                }
        }
        if (!do_group && (flag & RDC_FORCE_GROUP))
                do_group = 1;

        rdc_many_exit(krdc);
        dq = &krdc->group->diskq;
        if (do_group) {
#ifdef DEBUG
                cmn_err(CE_NOTE, "!SNDR:Group point-in-time for grp: %s %s:%s",
                    urdc->group_name, urdc->primary.intf, urdc->secondary.intf);
#endif
                DTRACE_PROBE(rdc_diskq_group_PIT);

                /* Set group logging at the same PIT under rdc_many_lock */
```

```
                rdc_many_enter(krdc);
                rdc_set_flags_log(urdc, RDC_LOGGING, why);
                if (RDC_IS_DISKQ(krdc->group))
                        flag_op(urdc, RDC_QUEUING);
                for (p = krdc->group_next; p != krdc; p = p->group_next) {
                        q = &rdc_u_info[p->index];
                        if (!IS_ENABLED(q))
                                continue;
                        rdc_set_flags_log(q, RDC_LOGGING,
                            "consistency group member following leader");
                        if (RDC_IS_DISKQ(p->group))
                                flag_op(q, RDC_QUEUING);
                }

                rdc_many_exit(krdc);

                /*
                 * This can cause the async threads to fail,
                 * which in turn will call rdc_group_log()
                 * again. Release the lock and re-aquire.
                 */
                rdc_group_exit(krdc);

                while (rdc_dump_alloc_bufs_cd(krdc->index) == EAGAIN)
                        delay(2);
                if (!RDC_IS_DISKQ(krdc->group))
                        RDC_ZERO_BITREF(krdc);

                rdc_inflwait(krdc->group);

                /*
                 * a little lazy, but neat. recall dump_alloc_bufs to
                 * ensure that the queue pointers & seq are reset properly
                 * after we have waited for inflight stuff
                 */
                while (rdc_dump_alloc_bufs_cd(krdc->index) == EAGAIN)
                        delay(2);

                rdc_group_enter(krdc);
                if (RDC_IS_DISKQ(krdc->group) && (!(flag & RDC_QUEUING))) {
                        /* fail or user request */
                        RDC_ZERO_BITREF(krdc);
                        mutex_enter(&krdc->group->diskq.disk_qlock);
                        rdc_init_diskq_header(krdc->group,
                            &krdc->group->diskq.disk_hdr);
                        SET_QNXTIO(dq, QHEAD(dq));
                        mutex_exit(&krdc->group->diskq.disk_qlock);
                }

                if (flag & RDC_ALLREMOTE) {
                        /* Tell other node to start logging */
                        if (krdc->lsrv && krdc->intf && !krdc->intf->if_down)
                                (void) rdc_net_state(krdc->index,
                                    CCIO_ENABLELOG);
                }

                if (flag & (RDC_ALLREMOTE | RDC_OTHERREMOTE)) {
                        rdc_many_enter(krdc);
                        for (p = krdc->group_next; p != krdc;
                            p = p->group_next) {
                                if (p->lsrv && krdc->intf &&
                                    !krdc->intf->if_down) {
                                        (void) rdc_net_state(p->index,
                                            CCIO_ENABLELOG);
                                }
                        }
                        rdc_many_exit(krdc);
                }

                rdc_write_state(urdc);
                for (p = krdc->group_next; p != krdc; p = p->group_next) {
                        q = &rdc_u_info[p->index];
                        if (!IS_ENABLED(q))
                                continue;
                        rdc_write_state(q);
                }
        } else {
                /* No point in time is possible, just deal with single set */

                if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
                        halt_sync(krdc);
                } else {
                        if (rdc_net_getstate(krdc, &sm, &um, &md, TRUE) < 0) {
                                rdc_clr_flags(urdc, RDC_SYNCING);
                                rdc_set_flags_log(urdc, RDC_LOGGING,
                                    "failed to read remote state");

                                rdc_write_state(urdc);
                                while (rdc_dump_alloc_bufs_cd(krdc->index)
                                    == EAGAIN)
                                        delay(2);
                                if ((RDC_IS_DISKQ(krdc->group)) &&
                                    (!(flag & RDC_QUEUING))) { /* fail! */
                                        mutex_enter(QLOCK(dq));
                                        rdc_init_diskq_header(krdc->group,
                                            &krdc->group->diskq.disk_hdr);
                                        SET_QNXTIO(dq, QHEAD(dq));
                                        mutex_exit(QLOCK(dq));
                                }

                                return;
                        }
                }

                if (rdc_get_vflags(urdc) & RDC_SYNCING)
                        return;

                if (RDC_IS_DISKQ(krdc->group))
                        flag_op(urdc, RDC_QUEUING);

                if ((RDC_IS_DISKQ(krdc->group)) &&
                    (!(flag & RDC_QUEUING))) { /* fail! */
                        RDC_ZERO_BITREF(krdc);
                        mutex_enter(QLOCK(dq));
                        rdc_init_diskq_header(krdc->group,
                            &krdc->group->diskq.disk_hdr);
                        SET_QNXTIO(dq, QHEAD(dq));
                        mutex_exit(QLOCK(dq));
                }

                if (!(rdc_get_vflags(urdc) & RDC_LOGGING)) {
                        rdc_set_flags_log(urdc, RDC_LOGGING, why);

                        rdc_write_state(urdc);

                        while (rdc_dump_alloc_bufs_cd(krdc->index) == EAGAIN)
                                delay(2);
                        if (!RDC_IS_DISKQ(krdc->group))
                                RDC_ZERO_BITREF(krdc);

                        rdc_inflwait(krdc->group);
                        /*
```

```
                         * a little lazy, but neat. recall dump_alloc_bufs to
                         * ensure that the queue pointers & seq are reset
                         * properly after we have waited for inflight stuff
                         */
                        while (rdc_dump_alloc_bufs_cd(krdc->index) == EAGAIN)
                                delay(2);

                        if (flag & RDC_ALLREMOTE) {
                                /* Tell other node to start logging */
                                if (krdc->lsrv && krdc->intf &&
                                    !krdc->intf->if_down) {
                                        (void) rdc_net_state(krdc->index,
                                            CCIO_ENABLELOG);
                                }
                        }
                }
        }
        /*
         * just in case any threads were in flight during log cleanup
         */
        if (RDC_IS_DISKQ(krdc->group)) {
                mutex_enter(QLOCK(dq));
                cv_broadcast(&dq->qfullcv);
                mutex_exit(QLOCK(dq));
        }
}

static int
_rdc_log(rdc_k_info_t *krdc, rdc_set_t *rdc_set, spcs_s_info_t kstatus)
{
        rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
        rdc_srv_t *svp;

        rdc_group_enter(krdc);
        if (rdc_check(krdc, rdc_set)) {
                rdc_group_exit(krdc);
                spcs_s_add(kstatus, RDC_EALREADY, rdc_set->primary.file,
                    rdc_set->secondary.file);
                return (RDC_EALREADY);
        }

        svp = krdc->lsrv;
        if (rdc_get_vflags(urdc) & RDC_PRIMARY)
                krdc->intf = rdc_add_to_if(svp, &(urdc->primary.addr),
                    &(urdc->secondary.addr), 1);
        else
                krdc->intf = rdc_add_to_if(svp, &(urdc->secondary.addr),
                    &(urdc->primary.addr), 0);

        if (!krdc->intf) {
                rdc_group_exit(krdc);
                spcs_s_add(kstatus, RDC_EADDTOIF, urdc->primary.intf,
                    urdc->secondary.intf);
                return (RDC_EADDTOIF);
        }

        rdc_group_log(krdc, RDC_FLUSH | RDC_ALLREMOTE, NULL);

        if (rdc_get_vflags(urdc) & RDC_SYNCING) {
                rdc_group_exit(krdc);
                spcs_s_add(kstatus, RDC_ESYNCING, urdc->primary.file);
                return (RDC_ESYNCING);
        }

        rdc_group_exit(krdc);
```

```c
        return (0);
}

static int
rdc_log(rdc_config_t *uparms, spcs_s_info_t kstatus)
{
        rdc_k_info_t *krdc;
        int rc = 0;
        int index;

        mutex_enter(&rdc_conf_lock);
        index = rdc_lookup_byname(uparms->rdc_set);
        if (index >= 0)
                krdc = &rdc_k_info[index];
        if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }

        set_busy(krdc);
        if (krdc->type_flag == 0) {
                /* A resume or enable failed */
                wakeup_busy(krdc);
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }
        mutex_exit(&rdc_conf_lock);

        rc = _rdc_log(krdc, uparms->rdc_set, kstatus);

        mutex_enter(&rdc_conf_lock);
        wakeup_busy(krdc);
        mutex_exit(&rdc_conf_lock);

        return (rc);
}


static int
rdc_wait(rdc_config_t *uparms, spcs_s_info_t kstatus)
{
        rdc_k_info_t *krdc;
        rdc_u_info_t *urdc;
        int index;
        int need_check = 0;

        mutex_enter(&rdc_conf_lock);
        index = rdc_lookup_byname(uparms->rdc_set);
        if (index >= 0)
                krdc = &rdc_k_info[index];
        if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }

        urdc = &rdc_u_info[index];
        if (!(rdc_get_vflags(urdc) & RDC_PRIMARY)) {
                mutex_exit(&rdc_conf_lock);
                return (0);
        }
```

```c

        set_busy(krdc);
        if (krdc->type_flag == 0) {
                /* A resume or enable failed */
                wakeup_busy(krdc);
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }
        mutex_exit(&rdc_conf_lock);

        rdc_group_enter(krdc);
        if (rdc_check(krdc, uparms->rdc_set)) {
                rdc_group_exit(krdc);
                mutex_enter(&rdc_conf_lock);
                wakeup_busy(krdc);
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }

        if ((rdc_get_vflags(urdc) & (RDC_SYNCING | RDC_PRIMARY)) !=
            (RDC_SYNCING | RDC_PRIMARY)) {
                rdc_group_exit(krdc);
                mutex_enter(&rdc_conf_lock);
                wakeup_busy(krdc);
                mutex_exit(&rdc_conf_lock);
                return (0);
        }
        if (rdc_get_vflags(urdc) & RDC_SYNCING) {
                need_check = 1;
        }
        rdc_group_exit(krdc);

        mutex_enter(&net_blk_lock);

        mutex_enter(&rdc_conf_lock);
        wakeup_busy(krdc);
        mutex_exit(&rdc_conf_lock);

        (void) cv_wait_sig(&krdc->synccv, &net_blk_lock);

        mutex_exit(&net_blk_lock);
        if (need_check) {
                if (krdc->sync_done == RDC_COMPLETED) {
                        return (0);
                } else if (krdc->sync_done == RDC_FAILED) {
                        return (EIO);
                }
        }
        return (0);
}


static int
rdc_health(rdc_config_t *uparms, spcs_s_info_t kstatus, int *rvp)
{
        rdc_k_info_t *krdc;
        rdc_u_info_t *urdc;
        int rc = 0;
        int index;

        mutex_enter(&rdc_conf_lock);
        index = rdc_lookup_byname(uparms->rdc_set);
        if (index >= 0)
                krdc = &rdc_k_info[index];
        if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }

        set_busy(krdc);
        if (krdc->type_flag == 0) {
                /* A resume or enable failed */
                wakeup_busy(krdc);
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }

        mutex_exit(&rdc_conf_lock);

        rdc_group_enter(krdc);
        if (rdc_check(krdc, uparms->rdc_set)) {
                rdc_group_exit(krdc);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                rc = RDC_EALREADY;
                goto done;
        }

        urdc = &rdc_u_info[index];
        if (rdc_isactive_if(&(urdc->primary.addr), &(urdc->secondary.addr)))
                *rvp = RDC_ACTIVE;
        else
                *rvp = RDC_INACTIVE;

        rdc_group_exit(krdc);

done:
        mutex_enter(&rdc_conf_lock);
        wakeup_busy(krdc);
        mutex_exit(&rdc_conf_lock);

        return (rc);
}


static int
rdc_reconfig(rdc_config_t *uparms, spcs_s_info_t kstatus)
{
        rdc_k_info_t *krdc;
        rdc_u_info_t *urdc;
        int rc = -2;
        int index;

        mutex_enter(&rdc_conf_lock);
        index = rdc_lookup_byname(uparms->rdc_set);
        if (index >= 0)
                krdc = &rdc_k_info[index];
        if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }
```

```
        urdc = &rdc_u_info[index];
        set_busy(krdc);
        if (krdc->type_flag == 0) {
                /* A resume or enable failed */
                wakeup_busy(krdc);
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }

        mutex_exit(&rdc_conf_lock);

        rdc_group_enter(krdc);
        if (rdc_check(krdc, uparms->rdc_set)) {
                rdc_group_exit(krdc);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                rc = RDC_EALREADY;
                goto done;
        }
        if ((rdc_get_vflags(urdc) & RDC_BMP_FAILED) && (krdc->bitmapfd))
                (void) rdc_reset_bitmap(krdc);

        /* Move to a new bitmap if necessary */
        if (strncmp(urdc->primary.bitmap, uparms->rdc_set->primary.bitmap,
            NSC_MAXPATH) != 0) {
                if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
                        rc = rdc_move_bitmap(krdc,
                            uparms->rdc_set->primary.bitmap);
                } else {
                        (void) strncpy(urdc->primary.bitmap,
                            uparms->rdc_set->primary.bitmap, NSC_MAXPATH);
                        /* simulate a succesful rdc_move_bitmap */
                        rc = 0;
                }
        }
        if (strncmp(urdc->secondary.bitmap, uparms->rdc_set->secondary.bitmap,
            NSC_MAXPATH) != 0) {
                if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
                        (void) strncpy(urdc->secondary.bitmap,
                            uparms->rdc_set->secondary.bitmap, NSC_MAXPATH);
                        /* simulate a succesful rdc_move_bitmap */
                        rc = 0;
                } else {
                        rc = rdc_move_bitmap(krdc,
                            uparms->rdc_set->secondary.bitmap);
                }
        }
        if (rc == -1) {
                rdc_group_exit(krdc);
                spcs_s_add(kstatus, RDC_EBMPRECONFIG,
                    uparms->rdc_set->secondary.intf,
                    uparms->rdc_set->secondary.file);
                rc = RDC_EBMPRECONFIG;
                goto done;
        }

        /*
         * At this point we fail any other type of reconfig
         * if not in logging mode and we did not do a bitmap reconfig
         */

        if (!(rdc_get_vflags(urdc) & RDC_LOGGING) && rc == -2) {
                /* no other changes possible unless logging */
                rdc_group_exit(krdc);
                spcs_s_add(kstatus, RDC_ENOTLOGGING,
                    uparms->rdc_set->primary.intf,
                    uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.intf,
                    uparms->rdc_set->secondary.file);
                rc = RDC_ENOTLOGGING;
                goto done;
        }
        rc = 0;
        /* Change direct file if necessary */
        if ((rdc_get_vflags(urdc) & RDC_PRIMARY) &&
            strncmp(urdc->direct_file, uparms->rdc_set->direct_file,
            NSC_MAXPATH)) {
                if (!(rdc_get_vflags(urdc) & RDC_LOGGING)) {
                        rdc_group_exit(krdc);
                        goto notlogging;
                }
                rdc_close_direct(krdc);
                (void) strncpy(urdc->direct_file, uparms->rdc_set->direct_file,
                    NSC_MAXPATH);

                if (urdc->direct_file[0]) {
                        if (rdc_open_direct(krdc) == NULL)
                                rdc_set_flags(urdc, RDC_FCAL_FAILED);
                        else
                                rdc_clr_flags(urdc, RDC_FCAL_FAILED);
                }
        }

        rdc_group_exit(krdc);

        /* Change group if necessary */
        if (strncmp(urdc->group_name, uparms->rdc_set->group_name,
            NSC_MAXPATH) != 0) {
                char orig_group[NSC_MAXPATH];
                if (!(rdc_get_vflags(urdc) & RDC_LOGGING))
                        goto notlogging;
                mutex_enter(&rdc_conf_lock);

                (void) strncpy(orig_group, urdc->group_name, NSC_MAXPATH);
                (void) strncpy(urdc->group_name, uparms->rdc_set->group_name,
                    NSC_MAXPATH);

                rc = change_group(krdc, uparms->options);
                if (rc == RDC_EQNOADD) {
                        mutex_exit(&rdc_conf_lock);
                        spcs_s_add(kstatus, RDC_EQNOADD,
                            uparms->rdc_set->disk_queue);
                        goto done;
                } else if (rc < 0) {
                        (void) strncpy(urdc->group_name, orig_group,
                            NSC_MAXPATH);
                        mutex_exit(&rdc_conf_lock);
                        spcs_s_add(kstatus, RDC_EGROUP,
                            urdc->primary.intf, urdc->primary.file,
                            urdc->secondary.intf, urdc->secondary.file,
                            uparms->rdc_set->group_name);
                        rc = RDC_EGROUP;
                        goto done;
                }

                mutex_exit(&rdc_conf_lock);

                if (rc >= 0) {
                        if (!(rdc_get_vflags(urdc) & RDC_LOGGING))
                                goto notlogging;
```

```
                        if (uparms->options & RDC_OPT_ASYNC) {
                                mutex_enter(&rdc_conf_lock);
                                krdc->type_flag |= RDC_ASYNCMODE;
                                mutex_exit(&rdc_conf_lock);
                                if (uparms->options & RDC_OPT_PRIMARY)
                                        krdc->bitmap_ref =
                                            (uchar_t *)kmem_zalloc(
                                            (krdc->bitmap_size * BITS_IN_BYTE *
                                            BMAP_REF_PREF_SIZE), KM_SLEEP);
                                rdc_group_enter(krdc);
                                rdc_set_flags(urdc, RDC_ASYNC);
                                rdc_group_exit(krdc);
                        } else {
                                mutex_enter(&rdc_conf_lock);
                                krdc->type_flag &= ~RDC_ASYNCMODE;
                                mutex_exit(&rdc_conf_lock);
                                rdc_group_enter(krdc);
                                rdc_clr_flags(urdc, RDC_ASYNC);
                                rdc_group_exit(krdc);
                                if (krdc->bitmap_ref) {
                                        kmem_free(krdc->bitmap_ref,
                                            (krdc->bitmap_size * BITS_IN_BYTE *
                                            BMAP_REF_PREF_SIZE));
                                        krdc->bitmap_ref = NULL;
                                }
                        }
                }
        } else {
                if ((((uparms->options & RDC_OPT_ASYNC) == 0) &&
                    ((krdc->type_flag & RDC_ASYNCMODE) != 0)) ||
                    (((uparms->options & RDC_OPT_ASYNC) != 0) &&
                    ((krdc->type_flag & RDC_ASYNCMODE) == 0))) {
                        if (!(rdc_get_vflags(urdc) & RDC_LOGGING))
                                goto notlogging;

                        if (krdc->group->count > 1) {
                                spcs_s_add(kstatus, RDC_EGROUPMODE);
                                rc = RDC_EGROUPMODE;
                                goto done;
                        }
                }

                /* Switch sync/async if necessary */
                if (krdc->group->count == 1) {
                        /* Only member of group. Can change sync/async */
                        if (((uparms->options & RDC_OPT_ASYNC) == 0) &&
                            ((krdc->type_flag & RDC_ASYNCMODE) != 0)) {
                                if (!(rdc_get_vflags(urdc) & RDC_LOGGING))
                                        goto notlogging;
                                /* switch to sync */
                                mutex_enter(&rdc_conf_lock);
                                krdc->type_flag &= ~RDC_ASYNCMODE;
                                if (RDC_IS_DISKQ(krdc->group)) {
                                        krdc->group->flags &= ~RDC_DISKQUE;
                                        krdc->group->flags |= RDC_MEMQUE;
                                        rdc_unintercept_diskq(krdc->group);
                                        mutex_enter(&krdc->group->diskqmutex);
                                        rdc_close_diskq(krdc->group);
                                        mutex_exit(&krdc->group->diskqmutex);
                                        bzero(&urdc->disk_queue,
                                            sizeof (urdc->disk_queue));
                                }
                                mutex_exit(&rdc_conf_lock);
                                rdc_group_enter(krdc);
                                rdc_clr_flags(urdc, RDC_ASYNC);
                                rdc_group_exit(krdc);
                                if (krdc->bitmap_ref) {
                                        kmem_free(krdc->bitmap_ref,
                                            (krdc->bitmap_size * BITS_IN_BYTE *
                                            BMAP_REF_PREF_SIZE));
                                        krdc->bitmap_ref = NULL;
                                }
                        } else if (((uparms->options & RDC_OPT_ASYNC) != 0) &&
                            ((krdc->type_flag & RDC_ASYNCMODE) == 0)) {
                                if (!(rdc_get_vflags(urdc) & RDC_LOGGING))
                                        goto notlogging;
                                /* switch to async */
                                mutex_enter(&rdc_conf_lock);
                                krdc->type_flag |= RDC_ASYNCMODE;
                                mutex_exit(&rdc_conf_lock);
                                if (uparms->options & RDC_OPT_PRIMARY)
                                        krdc->bitmap_ref =
                                            (uchar_t *)kmem_zalloc(
                                            (krdc->bitmap_size * BITS_IN_BYTE *
                                            BMAP_REF_PREF_SIZE), KM_SLEEP);
                                rdc_group_enter(krdc);
                                rdc_set_flags(urdc, RDC_ASYNC);
                                rdc_group_exit(krdc);
                        }
                }
        }
        /* Reverse concept of primary and secondary */
        if ((uparms->options & RDC_OPT_REVERSE_ROLE) != 0) {
                rdc_set_t rdc_set;
                struct netbuf paddr, saddr;

                mutex_enter(&rdc_conf_lock);

                /*
                 * Disallow role reversal for advanced configurations
                 */

                if (IS_MANY(krdc) || IS_MULTI(krdc)) {
                        mutex_exit(&rdc_conf_lock);
                        spcs_s_add(kstatus, RDC_EMASTER, urdc->primary.intf,
                            urdc->primary.file, urdc->secondary.intf,
                            urdc->secondary.file);
                        return (RDC_EMASTER);
                }
                bzero((void *) &rdc_set, sizeof (rdc_set_t));
                dup_rdc_netbuf(&urdc->primary.addr, &saddr);
                dup_rdc_netbuf(&urdc->secondary.addr, &paddr);
                free_rdc_netbuf(&urdc->primary.addr);
                free_rdc_netbuf(&urdc->secondary.addr);
                dup_rdc_netbuf(&saddr, &urdc->secondary.addr);
                dup_rdc_netbuf(&paddr, &urdc->primary.addr);
                free_rdc_netbuf(&paddr);
                free_rdc_netbuf(&saddr);
                /* copy primary parts of urdc to rdc_set field by field */
                (void) strncpy(rdc_set.primary.intf, urdc->primary.intf,
                    MAX_RDC_HOST_SIZE);
                (void) strncpy(rdc_set.primary.file, urdc->primary.file,
                    NSC_MAXPATH);
                (void) strncpy(rdc_set.primary.bitmap, urdc->primary.bitmap,
                    NSC_MAXPATH);

                /* Now overwrite urdc primary */
                (void) strncpy(urdc->primary.intf, urdc->secondary.intf,
                    MAX_RDC_HOST_SIZE);
                (void) strncpy(urdc->primary.file, urdc->secondary.file,
                    NSC_MAXPATH);
                (void) strncpy(urdc->primary.bitmap, urdc->secondary.bitmap,
```

```
                    NSC_MAXPATH);

                /* Now ovwewrite urdc secondary */
                (void) strncpy(urdc->secondary.intf, rdc_set.primary.intf,
                    MAX_RDC_HOST_SIZE);
                (void) strncpy(urdc->secondary.file, rdc_set.primary.file,
                    NSC_MAXPATH);
                (void) strncpy(urdc->secondary.bitmap, rdc_set.primary.bitmap,
                    NSC_MAXPATH);

                if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
                        rdc_clr_flags(urdc, RDC_PRIMARY);
                        if (krdc->intf) {
                                krdc->intf->issecondary = 1;
                                krdc->intf->isprimary = 0;
                                krdc->intf->if_down = 1;
                        }
                } else {
                        rdc_set_flags(urdc, RDC_PRIMARY);
                        if (krdc->intf) {
                                krdc->intf->issecondary = 0;
                                krdc->intf->isprimary = 1;
                                krdc->intf->if_down = 1;
                        }
                }

                if ((rdc_get_vflags(urdc) & RDC_PRIMARY) &&
                    ((krdc->type_flag & RDC_ASYNCMODE) != 0)) {
                        if (!krdc->bitmap_ref)
                                krdc->bitmap_ref =
                                    (uchar_t *)kmem_zalloc((krdc->bitmap_size *
                                    BITS_IN_BYTE * BMAP_REF_PREF_SIZE),
                                    KM_SLEEP);
                        if (krdc->bitmap_ref == NULL) {
                                cmn_err(CE_WARN,
                                    "!rdc_reconfig: bitmap_ref alloc %"
                                    NSC_SZFMT " failed",
                                    krdc->bitmap_size * BITS_IN_BYTE *
                                    BMAP_REF_PREF_SIZE);
                                mutex_exit(&rdc_conf_lock);
                                return (-1);
                        }
                }

                if ((rdc_get_vflags(urdc) & RDC_PRIMARY) &&
                    (rdc_get_vflags(urdc) & RDC_SYNC_NEEDED)) {
                        /* Primary, so reverse sync needed */
                        rdc_many_enter(krdc);
                        rdc_clr_flags(urdc, RDC_SYNC_NEEDED);
                        rdc_set_mflags(urdc, RDC_RSYNC_NEEDED);
                        rdc_many_exit(krdc);
                } else if (rdc_get_vflags(urdc) & RDC_RSYNC_NEEDED) {
                        /* Secondary, so forward sync needed */
                        rdc_many_enter(krdc);
                        rdc_clr_flags(urdc, RDC_RSYNC_NEEDED);
                        rdc_set_flags(urdc, RDC_SYNC_NEEDED);
                        rdc_many_exit(krdc);
                }

                /*
                 * rewrite bitmap header
                 */
                rdc_write_state(urdc);
                mutex_exit(&rdc_conf_lock);
        }
```

```
done:
        mutex_enter(&rdc_conf_lock);
        wakeup_busy(krdc);
        mutex_exit(&rdc_conf_lock);

        return (rc);

notlogging:
        /* no other changes possible unless logging */
        mutex_enter(&rdc_conf_lock);
        wakeup_busy(krdc);
        mutex_exit(&rdc_conf_lock);
        spcs_s_add(kstatus, RDC_ENOTLOGGING, urdc->primary.intf,
            urdc->primary.file, urdc->secondary.intf,
            urdc->secondary.file);
        return (RDC_ENOTLOGGING);
}

static int
rdc_reset(rdc_config_t *uparms, spcs_s_info_t kstatus)
{
        rdc_k_info_t *krdc;
        rdc_u_info_t *urdc;
        int rc = 0;
        int index;
        int cleared_error = 0;

        mutex_enter(&rdc_conf_lock);
        index = rdc_lookup_byname(uparms->rdc_set);
        if (index >= 0)
                krdc = &rdc_k_info[index];
        if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }

        urdc = &rdc_u_info[index];
        set_busy(krdc);
        if (krdc->type_flag == 0) {
                /* A resume or enable failed */
                wakeup_busy(krdc);
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }

        mutex_exit(&rdc_conf_lock);

        rdc_group_enter(krdc);
        if (rdc_check(krdc, uparms->rdc_set)) {
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                rc = RDC_EALREADY;
                goto done;
        }

        if ((rdc_get_vflags(urdc) & RDC_BMP_FAILED) && (krdc->bitmapfd)) {
                if (rdc_reset_bitmap(krdc) == 0)
                        cleared_error++;
        }

        /* Fix direct file if necessary */
        if ((rdc_get_vflags(urdc) & RDC_PRIMARY) && urdc->direct_file[0]) {
```

```
                if (rdc_open_direct(krdc) == NULL)
                        rdc_set_flags(urdc, RDC_FCAL_FAILED);
                else {
                        rdc_clr_flags(urdc, RDC_FCAL_FAILED);
                        cleared_error++;
                }
        }

        if ((rdc_get_vflags(urdc) & RDC_VOL_FAILED)) {
                rdc_many_enter(krdc);
                rdc_clr_flags(urdc, RDC_VOL_FAILED);
                cleared_error++;
                rdc_many_exit(krdc);
        }

        if (cleared_error) {
                /* cleared an error so we should be in logging mode */
                rdc_set_flags_log(urdc, RDC_LOGGING, "set reset");
        }
        rdc_group_exit(krdc);

        if ((rdc_get_vflags(urdc) & RDC_DISKQ_FAILED))
                rdc_unfail_diskq(krdc);

done:
        mutex_enter(&rdc_conf_lock);
        wakeup_busy(krdc);
        mutex_exit(&rdc_conf_lock);

        return (rc);
}


static int
rdc_tunable(rdc_config_t *uparms, spcs_s_info_t kstatus)
{
        rdc_k_info_t *krdc;
        rdc_u_info_t *urdc;
        rdc_k_info_t *p;
        rdc_u_info_t *q;
        int rc = 0;
        int index;

        mutex_enter(&rdc_conf_lock);
        index = rdc_lookup_byname(uparms->rdc_set);
        if (index >= 0)
                krdc = &rdc_k_info[index];
        if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }

        urdc = &rdc_u_info[index];
        set_busy(krdc);
        if (krdc->type_flag == 0) {
                /* A resume or enable failed */
                wakeup_busy(krdc);
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }

        mutex_exit(&rdc_conf_lock);
        rdc_group_enter(krdc);
        if (rdc_check(krdc, uparms->rdc_set)) {
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                rc = RDC_EALREADY;
                goto done;
        }

        if (uparms->rdc_set->maxqfbas > 0) {
                urdc->maxqfbas = uparms->rdc_set->maxqfbas;
                rdc_write_state(urdc);
                for (p = krdc->group_next; p != krdc; p = p->group_next) {
                        q = &rdc_u_info[p->index];
                        q->maxqfbas = urdc->maxqfbas;
                        rdc_write_state(q);
                }
        }

        if (uparms->rdc_set->maxqitems > 0) {
                urdc->maxqitems = uparms->rdc_set->maxqitems;
                rdc_write_state(urdc);
                for (p = krdc->group_next; p != krdc; p = p->group_next) {
                        q = &rdc_u_info[p->index];
                        q->maxqitems = urdc->maxqitems;
                        rdc_write_state(q);
                }
        }

        if (uparms->options & RDC_OPT_SET_QNOBLOCK) {
                disk_queue *que;

                if (!RDC_IS_DISKQ(krdc->group)) {
                        spcs_s_add(kstatus, RDC_EQNOQUEUE, urdc->primary.intf,
                            urdc->primary.file, urdc->secondary.intf,
                            urdc->secondary.file);
                        rc = RDC_EQNOQUEUE;
                        goto done;
                }

                que = &krdc->group->diskq;
                mutex_enter(QLOCK(que));
                SET_QSTATE(que, RDC_QNOBLOCK);
                /* queue will fail if this fails */
                (void) rdc_stamp_diskq(krdc, 0, RDC_GROUP_LOCKED);
                mutex_exit(QLOCK(que));

        }

        if (uparms->options & RDC_OPT_CLR_QNOBLOCK) {
                disk_queue *que;

                if (!RDC_IS_DISKQ(krdc->group)) {
                        spcs_s_add(kstatus, RDC_EQNOQUEUE, urdc->primary.intf,
                            urdc->primary.file, urdc->secondary.intf,
                            urdc->secondary.file);
                        rc = RDC_EQNOQUEUE;
                        goto done;
                }
                que = &krdc->group->diskq;
                mutex_enter(QLOCK(que));
                CLR_QSTATE(que, RDC_QNOBLOCK);
                /* queue will fail if this fails */
                (void) rdc_stamp_diskq(krdc, 0, RDC_GROUP_LOCKED);
                mutex_exit(QLOCK(que));

```

```
        }
        if (uparms->rdc_set->asyncthr > 0) {
                urdc->asyncthr = uparms->rdc_set->asyncthr;
                rdc_write_state(urdc);
                for (p = krdc->group_next; p != krdc; p = p->group_next) {
                        q = &rdc_u_info[p->index];
                        q->asyncthr = urdc->asyncthr;
                        rdc_write_state(q);
                }
        }

        if (uparms->rdc_set->autosync >= 0) {
                if (uparms->rdc_set->autosync == 0)
                        urdc->autosync = 0;
                else
                        urdc->autosync = 1;

                rdc_write_state(urdc);

                /* Changed autosync, so update rest of the group */

                for (p = krdc->group_next; p != krdc; p = p->group_next) {
                        q = &rdc_u_info[p->index];
                        q->autosync = urdc->autosync;
                        rdc_write_state(q);
                }
        }

done:
        rdc_group_exit(krdc);

        mutex_enter(&rdc_conf_lock);
        wakeup_busy(krdc);
        mutex_exit(&rdc_conf_lock);

        return (rc);
}

/*
 * Yet another standard thing that is not standard ...
 */
#ifndef offsetof
#define offsetof(s, m)  ((size_t)(&((s *)0)->m))
#endif

static int
rdc_status(void *arg, int mode, rdc_config_t *uparms, spcs_s_info_t kstatus)
{
        rdc_k_info_t *krdc;
        rdc_u_info_t *urdc;
        disk_queue *dqp;
        int rc = 0;
        int index;
        char *ptr;
        extern int rdc_status_copy32(const void *, void *, size_t, int);

        mutex_enter(&rdc_conf_lock);
        index = rdc_lookup_byname(uparms->rdc_set);
        if (index >= 0)
                krdc = &rdc_k_info[index];
        if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }
        set_busy(krdc);
        if (krdc->type_flag == 0) {
                /* A resume or enable failed */
                wakeup_busy(krdc);
                mutex_exit(&rdc_conf_lock);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                return (RDC_EALREADY);
        }

        mutex_exit(&rdc_conf_lock);

        rdc_group_enter(krdc);
        if (rdc_check(krdc, uparms->rdc_set)) {
                rdc_group_exit(krdc);
                spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
                    uparms->rdc_set->secondary.file);
                rc = RDC_EALREADY;
                goto done;
        }

        urdc = &rdc_u_info[index];

        /*
         * sneak out qstate in urdc->flags
         * this is harmless because it's value is not used
         * in urdc->flags. the real qstate is kept in
         * group->diskq->disk_hdr.h.state
         */
        if (RDC_IS_DISKQ(krdc->group)) {
                dqp = &krdc->group->diskq;
                if (IS_QSTATE(dqp, RDC_QNOBLOCK))
                urdc->flags |= RDC_QNOBLOCK;
        }

        if (ddi_model_convert_from(mode & FMODELS) == DDI_MODEL_ILP32) {
                ptr = (char *)arg + offsetof(struct rdc_config32, rdc_set);
                rc = rdc_status_copy32(urdc, ptr, sizeof (struct rdc_set32),
                    mode);
        } else {
                ptr = (char *)arg + offsetof(struct rdc_config, rdc_set);
                rc = ddi_copyout(urdc, ptr, sizeof (struct rdc_set), mode);
        }
        /* clear out qstate from flags */
        urdc->flags &= ~RDC_QNOBLOCK;

        if (rc)
                rc = EFAULT;

        rdc_group_exit(krdc);
done:
        mutex_enter(&rdc_conf_lock);
        wakeup_busy(krdc);
        mutex_exit(&rdc_conf_lock);

        return (rc);
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   60735 Thu Feb 25 15:39:38 2016
new/usr/src/uts/common/avs/ns/sv/sv.c
2976 remove useless offsetof() macros
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 *
 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
 */

/*
 * Storage Volume Character and Block Driver (SV)
 *
 * This driver implements a simplistic /dev/{r}dsk/ interface to a
 * specified disk volume that is otherwise managed by the Prism
 * software.  The SV driver layers itself onto the underlying disk
 * device driver by changing function pointers in the cb_ops
 * structure.
 *
 * CONFIGURATION:
 *
 * 1. Configure the driver using the svadm utility.
 * 2. Access the device as before through /dev/rdsk/c?t?d?s?
 *
 * LIMITATIONS:
 *
 * This driver should NOT be used to share a device between another
 * DataServices user interface module (e.g., STE) and a user accessing
 * the device through the block device in O_WRITE mode.  This is because
 * writes through the block device are asynchronous (due to the page
 * cache) and so consistency between the block device user and the
 * STE user cannot be guaranteed.
 *
 * Data is copied between system struct buf(9s) and nsc_vec_t.  This is
 * wasteful and slow.
 */

#include <sys/debug.h>
#include <sys/types.h>

#include <sys/ksynch.h>
#include <sys/kmem.h>
#include <sys/errno.h>
#include <sys/varargs.h>
```

```
#include <sys/file.h>
#include <sys/open.h>
#include <sys/conf.h>
#include <sys/cred.h>
#include <sys/buf.h>
#include <sys/uio.h>
#ifndef DS_DDICT
#include <sys/pathname.h>
#endif
#include <sys/aio_req.h>
#include <sys/dkio.h>
#include <sys/vtoc.h>
#include <sys/cmn_err.h>
#include <sys/modctl.h>
#include <sys/ddi.h>
#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/sunddi.h>
#include <sys/sunldi.h>
#include <sys/nsctl/nsvers.h>

#include <sys/nsc_thread.h>
#include <sys/unistat/spcs_s.h>
#include <sys/unistat/spcs_s_k.h>
#include <sys/unistat/spcs_errors.h>

#ifdef DS_DDICT
#include "../contract.h"
#endif

#include "../nsctl.h"


#include <sys/sdt.h>            /* dtrace is S10 or later */

#include "sv.h"
#include "sv_impl.h"
#include "sv_efi.h"

#define MAX_EINTR_COUNT 1000

/*
 * sv_mod_status
 */
#define SV_PREVENT_UNLOAD 1
#define SV_ALLOW_UNLOAD 2

static const int sv_major_rev = ISS_VERSION_MAJ;        /* Major number */
static const int sv_minor_rev = ISS_VERSION_MIN;        /* Minor number */
static const int sv_micro_rev = ISS_VERSION_MIC;        /* Micro number */
static const int sv_baseline_rev = ISS_VERSION_NUM;     /* Baseline number */

#ifdef DKIOCPARTITION
/*
 * CRC32 polynomial table needed for computing the checksums
 * in an EFI vtoc.
 */
static const uint32_t sv_crc32_table[256] = { CRC32_TABLE };
#endif

static clock_t sv_config_time;          /* Time of successful {en,dis}able */
static int sv_debug;                    /* Set non-zero for debug to syslog */
static int sv_mod_status;               /* Set to prevent modunload */

static dev_info_t *sv_dip;              /* Single DIP for driver */
static kmutex_t sv_mutex;               /* Protect global lists, etc. */
static nsc_mem_t        *sv_mem;        /* nsctl memory allocator token */


/*
 * Per device and per major state.
 */

#ifndef _SunOS_5_6
#define UNSAFE_ENTER()
#define UNSAFE_EXIT()
#else
#define UNSAFE_ENTER()  mutex_enter(&unsafe_driver)
#define UNSAFE_EXIT()   mutex_exit(&unsafe_driver)
#endif

                                        /* hash table of major dev structures */
static sv_maj_t *sv_majors[SV_MAJOR_HASH_CNT] = {0};
static sv_dev_t *sv_devs;               /* array of per device structures */
static int sv_max_devices;              /* SV version of nsc_max_devices() */
static int sv_ndevices;                 /* number of SV enabled devices */

/*
 * Threading.
 */

int sv_threads_max = 1024;              /* maximum # to dynamically alloc */
int sv_threads = 32;                    /* # to pre-allocate (see sv.conf) */
int sv_threads_extra = 0;               /* addl # we would have alloc'ed */

static nstset_t *sv_tset;               /* the threadset pointer */

static int sv_threads_hysteresis = 4;   /* hysteresis for threadset resizing */
static int sv_threads_dev = 2;          /* # of threads to alloc per device */
static int sv_threads_inc = 8;          /* increment for changing the set */
static int sv_threads_needed;           /* number of threads needed */
static int sv_no_threads;               /* number of nsc_create errors */
static int sv_max_nlive;                /* max number of threads running */



/*
 * nsctl fd callbacks.
 */

static int svattach_fd(blind_t);
static int svdetach_fd(blind_t);

static nsc_def_t sv_fd_def[] = {
        { "Attach",     (uintptr_t)svattach_fd, },
        { "Detach",     (uintptr_t)svdetach_fd, },
        { 0, 0, }
};

/*
 * cb_ops functions.
 */

static int svopen(dev_t *, int, int, cred_t *);
static int svclose(dev_t, int, int, cred_t *);
static int svioctl(dev_t, int, intptr_t, int, cred_t *, int *);
static int svprint(dev_t, char *);

/*
 * These next functions are layered into the underlying driver's devops.
 */
```

```
static int sv_lyr_open(dev_t *, int, int, cred_t *);
static int sv_lyr_close(dev_t, int, int, cred_t *);
static int sv_lyr_strategy(struct buf *);
static int sv_lyr_read(dev_t, struct uio *, cred_t *);
static int sv_lyr_write(dev_t, struct uio *, cred_t *);
static int sv_lyr_aread(dev_t, struct aio_req *, cred_t *);
static int sv_lyr_awrite(dev_t, struct aio_req *, cred_t *);
static int sv_lyr_ioctl(dev_t, int, intptr_t, int, cred_t *, int *);

static struct cb_ops sv_cb_ops = {
        svopen,         /* open */
        svclose,        /* close */
        nulldev,        /* strategy */
        svprint,
        nodev,          /* dump */
        nodev,          /* read */
        nodev,          /* write */
        svioctl,
        nodev,          /* devmap */
        nodev,          /* mmap */
        nodev,          /* segmap */
        nochpoll,       /* poll */
        ddi_prop_op,
        NULL,           /* NOT a stream */
        D_NEW | D_MP | D_64BIT,
        CB_REV,
        nodev,          /* aread */
        nodev,          /* awrite */
};


/*
 * dev_ops functions.
 */

static int sv_getinfo(dev_info_t *, ddi_info_cmd_t, void *, void **);
static int sv_attach(dev_info_t *, ddi_attach_cmd_t);
static int sv_detach(dev_info_t *, ddi_detach_cmd_t);

static struct dev_ops sv_ops = {
        DEVO_REV,
        0,
        sv_getinfo,
        nulldev,        /* identify */
        nulldev,        /* probe */
        sv_attach,
        sv_detach,
        nodev,          /* reset */
        &sv_cb_ops,
        (struct bus_ops *)0
};

/*
 * Module linkage.
 */

extern struct mod_ops mod_driverops;

static struct modldrv modldrv = {
        &mod_driverops,
        "nws:Storage Volume:" ISS_VERSION_STR,
        &sv_ops
};

static struct modlinkage modlinkage = {
        MODREV_1,
        &modldrv,
        0
};


int
_init(void)
{
        int error;

        mutex_init(&sv_mutex, NULL, MUTEX_DRIVER, NULL);

        if ((error = mod_install(&modlinkage)) != 0) {
                mutex_destroy(&sv_mutex);
                return (error);
        }

#ifdef DEBUG
        cmn_err(CE_CONT, "!sv (revision %d.%d.%d.%d, %s, %s)\n",
            sv_major_rev, sv_minor_rev, sv_micro_rev, sv_baseline_rev,
            ISS_VERSION_STR, BUILD_DATE_STR);
#else
        if (sv_micro_rev) {
                cmn_err(CE_CONT, "!sv (revision %d.%d.%d, %s, %s)\n",
                    sv_major_rev, sv_minor_rev, sv_micro_rev,
                    ISS_VERSION_STR, BUILD_DATE_STR);
        } else {
                cmn_err(CE_CONT, "!sv (revision %d.%d, %s, %s)\n",
                    sv_major_rev, sv_minor_rev,
                    ISS_VERSION_STR, BUILD_DATE_STR);
        }
#endif

        return (error);
}


int
_fini(void)
{
        int error;

        if ((error = mod_remove(&modlinkage)) != 0)
                return (error);

        mutex_destroy(&sv_mutex);

        return (error);
}


int
_info(struct modinfo *modinfop)
{
        return (mod_info(&modlinkage, modinfop));
}


/*
 * Locking & State.
 *
 * sv_mutex protects config information - sv_maj_t and sv_dev_t lists;
 * threadset creation and sizing; sv_ndevices.
 *
 * If we need to hold both sv_mutex and sv_lock, then the sv_mutex
```

```
 * must be acquired first.
 *
 * sv_lock protects the sv_dev_t structure for an individual device.
 *
 * sv_olock protects the otyp/open members of the sv_dev_t.  If we need
 * to hold both sv_lock and sv_olock, then the sv_lock must be acquired
 * first.
 *
 * nsc_reserve/nsc_release are used in NSC_MULTI mode to allow multiple
 * I/O operations to a device simultaneously, as above.
 *
 * All nsc_open/nsc_close/nsc_reserve/nsc_release operations that occur
 * with sv_lock write-locked must be done with (sv_state == SV_PENDING)
 * and (sv_pending == curthread) so that any recursion through
 * sv_lyr_open/sv_lyr_close can be detected.
 */


static int
sv_init_devs(void)
{
        int i;

        ASSERT(MUTEX_HELD(&sv_mutex));

        if (sv_max_devices > 0)
                return (0);

        sv_max_devices = nsc_max_devices();

        if (sv_max_devices <= 0) {
                /* nsctl is not attached (nskernd not running) */
                if (sv_debug > 0)
                        cmn_err(CE_CONT, "!sv: nsc_max_devices = 0\n");
                return (EAGAIN);
        }

        sv_devs = nsc_kmem_zalloc((sv_max_devices * sizeof (*sv_devs)),
            KM_NOSLEEP, sv_mem);

        if (sv_devs == NULL) {
                cmn_err(CE_WARN, "!sv: could not allocate sv_devs array");
                return (ENOMEM);
        }

        for (i = 0; i < sv_max_devices; i++) {
                mutex_init(&sv_devs[i].sv_olock, NULL, MUTEX_DRIVER, NULL);
                rw_init(&sv_devs[i].sv_lock, NULL, RW_DRIVER, NULL);
        }

        if (sv_debug > 0)
                cmn_err(CE_CONT, "!sv: sv_init_devs successful\n");

        return (0);
}


static int
sv_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        int rc;

        switch (cmd) {

        case DDI_ATTACH:
                sv_dip = dip;
```

```
		if (ddi_create_minor_node(dip, "sv", S_IFCHR,
		    0, DDI_PSEUDO, 0) != DDI_SUCCESS)
			goto failed;

		mutex_enter(&sv_mutex);

		sv_mem = nsc_register_mem("SV", NSC_MEM_LOCAL, 0);
		if (sv_mem == NULL) {
			mutex_exit(&sv_mutex);
			goto failed;
		}

		rc = sv_init_devs();
		if (rc != 0 && rc != EAGAIN) {
			mutex_exit(&sv_mutex);
			goto failed;
		}

		mutex_exit(&sv_mutex);


		ddi_report_dev(dip);

		sv_threads = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
		    DDI_PROP_DONTPASS | DDI_PROP_NOTPROM,
		    "sv_threads", sv_threads);

		if (sv_debug > 0)
			cmn_err(CE_CONT, "!sv: sv_threads=%d\n", sv_threads);

		if (sv_threads > sv_threads_max)
			sv_threads_max = sv_threads;

		return (DDI_SUCCESS);

	default:
		return (DDI_FAILURE);
	}

failed:
	DTRACE_PROBE(sv_attach_failed);
	(void) sv_detach(dip, DDI_DETACH);
	return (DDI_FAILURE);
}


static int
sv_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
	sv_dev_t *svp;
	int i;

	switch (cmd) {

	case DDI_DETACH:

		/*
		 * Check that everything is disabled.
		 */

		mutex_enter(&sv_mutex);

		if (sv_mod_status == SV_PREVENT_UNLOAD) {
			mutex_exit(&sv_mutex);
			DTRACE_PROBE(sv_detach_err_prevent);
```

```
			return (DDI_FAILURE);
		}

		for (i = 0; sv_devs && i < sv_max_devices; i++) {
			svp = &sv_devs[i];

			if (svp->sv_state != SV_DISABLE) {
				mutex_exit(&sv_mutex);
				DTRACE_PROBE(sv_detach_err_busy);
				return (DDI_FAILURE);
			}
		}


		for (i = 0; sv_devs && i < sv_max_devices; i++) {
			mutex_destroy(&sv_devs[i].sv_olock);
			rw_destroy(&sv_devs[i].sv_lock);
		}

		if (sv_devs) {
			nsc_kmem_free(sv_devs,
			    (sv_max_devices * sizeof (*sv_devs)));
			sv_devs = NULL;
		}
		sv_max_devices = 0;

		if (sv_mem) {
			nsc_unregister_mem(sv_mem);
			sv_mem = NULL;
		}

		mutex_exit(&sv_mutex);

		/*
		 * Remove all minor nodes.
		 */

		ddi_remove_minor_node(dip, NULL);
		sv_dip = NULL;

		return (DDI_SUCCESS);

	default:
		return (DDI_FAILURE);
	}
}

static sv_maj_t *
sv_getmajor(const dev_t dev)
{
	sv_maj_t **insert, *maj;
	major_t umaj = getmajor(dev);

	/*
	 * See if the hash table entry, or one of the hash chains
	 * is already allocated for this major number
	 */
	if ((maj = sv_majors[SV_MAJOR_HASH(umaj)]) != 0) {
		do {
			if (maj->sm_major == umaj)
				return (maj);
		} while ((maj = maj->sm_next) != 0);
	}

	/*
	 * If the sv_mutex is held, there is design flaw, as the only non-mutex
	 * held callers can be sv_enable() or sv_dev_to_sv()
	 * Return an error, instead of panicing the system
	 */
	if (MUTEX_HELD(&sv_mutex)) {
		cmn_err(CE_WARN, "!sv: could not allocate sv_maj_t");
		return (NULL);
	}

	/*
	 * Determine where to allocate a new element in the hash table
	 */
	mutex_enter(&sv_mutex);
	insert = &(sv_majors[SV_MAJOR_HASH(umaj)]);
	for (maj = *insert; maj; maj = maj->sm_next) {

		/* Did another thread beat us to it? */
		if (maj->sm_major == umaj)
			return (maj);

		/* Find a NULL insert point? */
		if (maj->sm_next == NULL)
			insert = &maj->sm_next;
	}

	/*
	 * Located the new insert point
	 */
	*insert = nsc_kmem_zalloc(sizeof (*maj), KM_NOSLEEP, sv_mem);
	if ((maj = *insert) != 0)
		maj->sm_major = umaj;
	else
		cmn_err(CE_WARN, "!sv: could not allocate sv_maj_t");

	mutex_exit(&sv_mutex);

	return (maj);
}

/* ARGSUSED */

static int
sv_getinfo(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
{
	int rc = DDI_FAILURE;

	switch (infocmd) {

	case DDI_INFO_DEVT2DEVINFO:
		*result = sv_dip;
		rc = DDI_SUCCESS;
		break;

	case DDI_INFO_DEVT2INSTANCE:
		/*
		 * We only have a single instance.
		 */
		*result = 0;
		rc = DDI_SUCCESS;
		break;

	default:
		break;
	}

	return (rc);
}
```

```
/*
 * Hashing of devices onto major device structures.
 *
 * Individual device structures are hashed onto one of the sm_hash[]
 * buckets in the relevant major device structure.
 *
 * Hash insertion and deletion -must- be done with sv_mutex held.  Hash
 * searching does not require the mutex because of the sm_seq member.
 * sm_seq is incremented on each insertion (-after- hash chain pointer
 * manipulation) and each deletion (-before- hash chain pointer
 * manipulation).  When searching the hash chain, the seq number is
 * checked before accessing each device structure, if the seq number has
 * changed, then we restart the search from the top of the hash chain.
 * If we restart more than SV_HASH_RETRY times, we take sv_mutex and search
 * the hash chain (we are guaranteed that this search cannot be
 * interrupted).
 */

#define SV_HASH_RETRY	16

static sv_dev_t *
sv_dev_to_sv(const dev_t dev, sv_maj_t **majpp)
{
	minor_t umin = getminor(dev);
	sv_dev_t **hb, *next, *svp;
	sv_maj_t *maj;
	int seq;
	int try;

	/* Get major hash table */
	maj = sv_getmajor(dev);
	if (majpp)
		*majpp = maj;
	if (maj == NULL)
		return (NULL);

	if (maj->sm_inuse == 0) {
		DTRACE_PROBE1(
		    sv_dev_to_sv_end,
		    dev_t, dev);
		return (NULL);
	}

	hb = &(maj->sm_hash[SV_MINOR_HASH(umin)]);
	try = 0;

retry:
	if (try > SV_HASH_RETRY)
		mutex_enter(&sv_mutex);

	seq = maj->sm_seq;
	for (svp = *hb; svp; svp = next) {
		next = svp->sv_hash;

		nsc_membar_stld();	/* preserve register load order */

		if (maj->sm_seq != seq) {
			DTRACE_PROBE1(sv_dev_to_sv_retry, dev_t, dev);
			try++;
			goto retry;
		}

		if (svp->sv_dev == dev)
			break;
```

```
        }

        if (try > SV_HASH_RETRY)
                mutex_exit(&sv_mutex);

        return (svp);
}


/*
 * Must be called with sv_mutex held.
 */

static int
sv_get_state(const dev_t udev, sv_dev_t **svpp)
{
        sv_dev_t **hb, **insert, *svp;
        sv_maj_t *maj;
        minor_t umin;
        int i;

        /* Get major hash table */
        if ((maj = sv_getmajor(udev)) == NULL)
                return (NULL);

        /* Determine which minor hash table */
        umin = getminor(udev);
        hb = &(maj->sm_hash[SV_MINOR_HASH(umin)]);

        /* look for clash */

        insert = hb;

        for (svp = *hb; svp; svp = svp->sv_hash) {
                if (svp->sv_dev == udev)
                        break;

                if (svp->sv_hash == NULL)
                        insert = &svp->sv_hash;
        }

        if (svp) {
                DTRACE_PROBE1(
                    sv_get_state_enabled,
                    dev_t, udev);
                return (SV_EENABLED);
        }

        /* look for spare sv_devs slot */

        for (i = 0; i < sv_max_devices; i++) {
                svp = &sv_devs[i];

                if (svp->sv_state == SV_DISABLE)
                        break;
        }

        if (i >= sv_max_devices) {
                DTRACE_PROBE1(
                    sv_get_state_noslots,
                    dev_t, udev);
                return (SV_ENOSLOTS);
        }

        svp->sv_state = SV_PENDING;
        svp->sv_pending = curthread;
```

```

        *insert = svp;
        svp->sv_hash = NULL;
        maj->sm_seq++;          /* must be after the store to the hash chain */

        *svpp = svp;

        /*
         * We do not know the size of the underlying device at
         * this stage, so initialise "nblocks" property to
         * zero, and update it whenever we succeed in
         * nsc_reserve'ing the underlying nsc_fd_t.
         */

        svp->sv_nblocks = 0;

        return (0);
}


/*
 * Remove a device structure from it's hash chain.
 * Must be called with sv_mutex held.
 */

static void
sv_rm_hash(sv_dev_t *svp)
{
        sv_dev_t **svpp;
        sv_maj_t *maj;

        /* Get major hash table */
        if ((maj = sv_getmajor(svp->sv_dev)) == NULL)
                return;

        /* remove svp from hash chain */

        svpp = &(maj->sm_hash[SV_MINOR_HASH(getminor(svp->sv_dev))]);
        while (*svpp) {
                if (*svpp == svp) {
                        /*
                         * increment of sm_seq must be before the
                         * removal from the hash chain
                         */
                        maj->sm_seq++;
                        *svpp = svp->sv_hash;
                        break;
                }

                svpp = &(*svpp)->sv_hash;
        }

        svp->sv_hash = NULL;
}

/*
 * Free (disable) a device structure.
 * Must be called with sv_lock(RW_WRITER) and sv_mutex held, and will
 * perform the exits during its processing.
 */

static int
sv_free(sv_dev_t *svp, const int error)
{
        struct cb_ops *cb_ops;
        sv_maj_t *maj;
	/* Get major hash table */
	if ((maj = sv_getmajor(svp->sv_dev)) == NULL)
		return (NULL);

	svp->sv_state = SV_PENDING;
	svp->sv_pending = curthread;

	/*
	 * Close the fd's before removing from the hash or swapping
	 * back the cb_ops pointers so that the cache flushes before new
	 * io can come in.
	 */

	if (svp->sv_fd) {
		(void) nsc_close(svp->sv_fd);
		svp->sv_fd = 0;
	}

	sv_rm_hash(svp);

	if (error != SV_ESDOPEN &&
	    error != SV_ELYROPEN && --maj->sm_inuse == 0) {

		if (maj->sm_dev_ops)
			cb_ops = maj->sm_dev_ops->devo_cb_ops;
		else
			cb_ops = NULL;

		if (cb_ops && maj->sm_strategy != NULL) {
			cb_ops->cb_strategy = maj->sm_strategy;
			cb_ops->cb_close = maj->sm_close;
			cb_ops->cb_ioctl = maj->sm_ioctl;
			cb_ops->cb_write = maj->sm_write;
			cb_ops->cb_open = maj->sm_open;
			cb_ops->cb_read = maj->sm_read;
			cb_ops->cb_flag = maj->sm_flag;

			if (maj->sm_awrite)
				cb_ops->cb_awrite = maj->sm_awrite;

			if (maj->sm_aread)
				cb_ops->cb_aread = maj->sm_aread;

			/*
			 * corbin XXX
			 * Leave backing device ops in maj->sm_*
			 * to handle any requests that might come
			 * in during the disable.  This could be
			 * a problem however if the backing device
			 * driver is changed while we process these
			 * requests.
			 *
			 * maj->sm_strategy = 0;
			 * maj->sm_awrite = 0;
			 * maj->sm_write = 0;
			 * maj->sm_ioctl = 0;
			 * maj->sm_close = 0;
			 * maj->sm_aread = 0;
			 * maj->sm_read = 0;
			 * maj->sm_open = 0;
			 * maj->sm_flag = 0;
			 *
			 */
		}
```

```
		if (maj->sm_dev_ops) {
			maj->sm_dev_ops = 0;
		}
	}

	if (svp->sv_lh) {
		cred_t *crp = ddi_get_cred();

		/*
		 * Close the protective layered driver open using the
		 * Sun Private layered driver i/f.
		 */

		(void) ldi_close(svp->sv_lh, FREAD|FWRITE, crp);
		svp->sv_lh = NULL;
	}

	svp->sv_timestamp = nsc_lbolt();
	svp->sv_state = SV_DISABLE;
	svp->sv_pending = NULL;
	rw_exit(&svp->sv_lock);
	mutex_exit(&sv_mutex);

	return (error);
}

/*
 * Reserve the device, taking into account the possibility that
 * the reserve might have to be retried.
 */
static int
sv_reserve(nsc_fd_t *fd, int flags)
{
	int eintr_count;
	int rc;

	eintr_count = 0;
	do {
		rc = nsc_reserve(fd, flags);
		if (rc == EINTR) {
			++eintr_count;
			delay(2);
		}
	} while ((rc == EINTR) && (eintr_count < MAX_EINTR_COUNT));

	return (rc);
}

static int
sv_enable(const caddr_t path, const int flag,
    const dev_t udev, spcs_s_info_t kstatus)
{
	struct dev_ops *dev_ops;
	struct cb_ops *cb_ops;
	sv_dev_t *svp;
	sv_maj_t *maj;
	nsc_size_t nblocks;
	int rc;
	cred_t *crp;
	ldi_ident_t	li;

	if (udev == (dev_t)-1 || udev == 0) {
		DTRACE_PROBE1(
		    sv_enable_err_baddev,
		    dev_t, udev);
		return (SV_EBADDEV);
```

```
        }

        if ((flag & ~(NSC_CACHE|NSC_DEVICE)) != 0) {
                DTRACE_PROBE1(sv_enable_err_amode, dev_t, udev);
                return (SV_EAMODE);
        }

        /* Get major hash table */
        if ((maj = sv_getmajor(udev)) == NULL)
                return (SV_EBADDEV);

        mutex_enter(&sv_mutex);

        rc = sv_get_state(udev, &svp);
        if (rc) {
                mutex_exit(&sv_mutex);
                DTRACE_PROBE1(sv_enable_err_state, dev_t, udev);
                return (rc);
        }

        rw_enter(&svp->sv_lock, RW_WRITER);

        /*
         * Get real fd used for io
         */

        svp->sv_dev = udev;
        svp->sv_flag = flag;

        /*
         * OR in NSC_DEVICE to ensure that nskern grabs the real strategy
         * function pointer before sv swaps them out.
         */

        svp->sv_fd = nsc_open(path, (svp->sv_flag | NSC_DEVICE),
            sv_fd_def, (blind_t)udev, &rc);

        if (svp->sv_fd == NULL) {
                if (kstatus)
                        spcs_s_add(kstatus, rc);
                DTRACE_PROBE1(sv_enable_err_fd, dev_t, udev);
                return (sv_free(svp, SV_ESDOPEN));
        }

        /*
         * Perform a layered driver open using the Sun Private layered
         * driver i/f to ensure that the cb_ops structure for the driver
         * is not detached out from under us whilst sv is enabled.
         *
         */

        crp = ddi_get_cred();
        svp->sv_lh = NULL;

        if ((rc = ldi_ident_from_dev(svp->sv_dev, &li)) == 0) {
                rc = ldi_open_by_dev(&svp->sv_dev,
                    OTYP_BLK, FREAD|FWRITE, crp, &svp->sv_lh, li);
        }

        if (rc != 0) {
                if (kstatus)
                        spcs_s_add(kstatus, rc);
                DTRACE_PROBE1(sv_enable_err_lyr_open, dev_t, udev);
                return (sv_free(svp, SV_ELYROPEN));
        }
```

```
        /*
         * Do layering if required - must happen after nsc_open().
         */

        if (maj->sm_inuse++ == 0) {
                maj->sm_dev_ops = nsc_get_devops(getmajor(udev));

                if (maj->sm_dev_ops == NULL ||
                    maj->sm_dev_ops->devo_cb_ops == NULL) {
                        DTRACE_PROBE1(sv_enable_err_load, dev_t, udev);
                        return (sv_free(svp, SV_ELOAD));
                }

                dev_ops = maj->sm_dev_ops;
                cb_ops = dev_ops->devo_cb_ops;

                if (cb_ops->cb_strategy == NULL ||
                    cb_ops->cb_strategy == nodev ||
                    cb_ops->cb_strategy == nulldev) {
                        DTRACE_PROBE1(sv_enable_err_nostrategy, dev_t, udev);
                        return (sv_free(svp, SV_ELOAD));
                }

                if (cb_ops->cb_strategy == sv_lyr_strategy) {
                        DTRACE_PROBE1(sv_enable_err_svstrategy, dev_t, udev);
                        return (sv_free(svp, SV_ESTRATEGY));
                }

                maj->sm_strategy = cb_ops->cb_strategy;
                maj->sm_close = cb_ops->cb_close;
                maj->sm_ioctl = cb_ops->cb_ioctl;
                maj->sm_write = cb_ops->cb_write;
                maj->sm_open = cb_ops->cb_open;
                maj->sm_read = cb_ops->cb_read;
                maj->sm_flag = cb_ops->cb_flag;

                cb_ops->cb_flag = cb_ops->cb_flag | D_MP;
                cb_ops->cb_strategy = sv_lyr_strategy;
                cb_ops->cb_close = sv_lyr_close;
                cb_ops->cb_ioctl = sv_lyr_ioctl;
                cb_ops->cb_write = sv_lyr_write;
                cb_ops->cb_open = sv_lyr_open;
                cb_ops->cb_read = sv_lyr_read;

                /*
                 * Check that the driver has async I/O entry points
                 * before changing them.
                 */

                if (dev_ops->devo_rev < 3 || cb_ops->cb_rev < 1) {
                        maj->sm_awrite = 0;
                        maj->sm_aread = 0;
                } else {
                        maj->sm_awrite = cb_ops->cb_awrite;
                        maj->sm_aread = cb_ops->cb_aread;

                        cb_ops->cb_awrite = sv_lyr_awrite;
                        cb_ops->cb_aread = sv_lyr_aread;
                }

                /*
                 * Bug 4645743
                 *
                 * Prevent sv from ever unloading after it has interposed
                 * on a major device because there is a race between
                 * sv removing its layered entry points from the target
```

```
		 * dev_ops, a client coming in and accessing the driver,
		 * and the kernel modunloading the sv text.
		 *
		 * To allow unload, do svboot -u, which only happens in
		 * pkgrm time.
		 */
		ASSERT(MUTEX_HELD(&sv_mutex));
		sv_mod_status = SV_PREVENT_UNLOAD;
	}


	svp->sv_timestamp = nsc_lbolt();
	svp->sv_state = SV_ENABLE;
	svp->sv_pending = NULL;
	rw_exit(&svp->sv_lock);

	sv_ndevices++;
	mutex_exit(&sv_mutex);

	nblocks = 0;
	if (sv_reserve(svp->sv_fd, NSC_READ|NSC_MULTI|NSC_PCATCH) == 0) {
		nblocks = svp->sv_nblocks;
		nsc_release(svp->sv_fd);
	}

	cmn_err(CE_CONT, "!sv: rdev 0x%lx, nblocks %" NSC_SZFMT "\n",
	    svp->sv_dev, nblocks);

	return (0);
}


static int
sv_prepare_unload()
{
	int rc = 0;

	mutex_enter(&sv_mutex);

	if (sv_mod_status == SV_PREVENT_UNLOAD) {
		if ((sv_ndevices != 0) || (sv_tset != NULL)) {
			rc = EBUSY;
		} else {
			sv_mod_status = SV_ALLOW_UNLOAD;
			delay(SV_WAIT_UNLOAD * drv_usectohz(1000000));
		}
	}

	mutex_exit(&sv_mutex);
	return (rc);
}

static int
svattach_fd(blind_t arg)
{
	dev_t dev = (dev_t)arg;
	sv_dev_t *svp = sv_dev_to_sv(dev, NULL);
	int rc;

	if (sv_debug > 0)
		cmn_err(CE_CONT, "!svattach_fd(%p, %p)\n", arg, (void *)svp);

	if (svp == NULL) {
		cmn_err(CE_WARN, "!svattach_fd: no state (arg %p)", arg);
		return (0);
	}
```

```

	if ((rc = nsc_partsize(svp->sv_fd, &svp->sv_nblocks)) != 0) {
		cmn_err(CE_WARN,
		    "!svattach_fd: nsc_partsize() failed, rc %d", rc);
		svp->sv_nblocks = 0;
	}

	if ((rc = nsc_maxfbas(svp->sv_fd, 0, &svp->sv_maxfbas)) != 0) {
		cmn_err(CE_WARN,
		    "!svattach_fd: nsc_maxfbas() failed, rc %d", rc);
		svp->sv_maxfbas = 0;
	}

	if (sv_debug > 0) {
		cmn_err(CE_CONT,
		    "!svattach_fd(%p): size %" NSC_SZFMT ", "
		    "maxfbas %" NSC_SZFMT "\n",
		    arg, svp->sv_nblocks, svp->sv_maxfbas);
	}

	return (0);
}


static int
svdetach_fd(blind_t arg)
{
	dev_t dev = (dev_t)arg;
	sv_dev_t *svp = sv_dev_to_sv(dev, NULL);

	if (sv_debug > 0)
		cmn_err(CE_CONT, "!svdetach_fd(%p, %p)\n", arg, (void *)svp);

	/* svp can be NULL during disable of an sv */
	if (svp == NULL)
		return (0);

	svp->sv_maxfbas = 0;
	svp->sv_nblocks = 0;
	return (0);
}


/*
 * Side effect: if called with (guard != 0), then expects both sv_mutex
 * and sv_lock(RW_WRITER) to be held, and will release them before returning.
 */

/* ARGSUSED */
static int
sv_disable(dev_t dev, spcs_s_info_t kstatus)
{
	sv_dev_t *svp = sv_dev_to_sv(dev, NULL);

	if (svp == NULL) {

		DTRACE_PROBE1(sv_disable_err_nodev, sv_dev_t *, svp);
		return (SV_ENODEV);
	}

	mutex_enter(&sv_mutex);
	rw_enter(&svp->sv_lock, RW_WRITER);

	if (svp->sv_fd == NULL || svp->sv_state != SV_ENABLE) {
		rw_exit(&svp->sv_lock);
		mutex_exit(&sv_mutex);
```

```
                DTRACE_PROBE1(sv_disable_err_disabled, sv_dev_t *, svp);
                return (SV_EDISABLED);
        }


        sv_ndevices--;
        return (sv_free(svp, 0));
}



static int
sv_lyr_open(dev_t *devp, int flag, int otyp, cred_t *crp)
{
        nsc_buf_t *tmph;
        sv_dev_t *svp;
        sv_maj_t *maj;
        int (*fn)();
        dev_t odev;
        int ret;
        int rc;

        svp = sv_dev_to_sv(*devp, &maj);

        if (svp) {
                if (svp->sv_state == SV_PENDING &&
                    svp->sv_pending == curthread) {
                        /*
                         * This is a recursive open from a call to
                         * ddi_lyr_open_by_devt and so we just want
                         * to pass it straight through to the
                         * underlying driver.
                         */
                        DTRACE_PROBE2(sv_lyr_open_recursive,
                            sv_dev_t *, svp,
                            dev_t, *devp);
                        svp = NULL;
                } else
                        rw_enter(&svp->sv_lock, RW_READER);
        }

        odev = *devp;

        if (maj && (fn = maj->sm_open) != 0) {
                if (!(maj->sm_flag & D_MP)) {
                        UNSAFE_ENTER();
                        ret = (*fn)(devp, flag, otyp, crp);
                        UNSAFE_EXIT();
                } else {
                        ret = (*fn)(devp, flag, otyp, crp);
                }

                if (ret == 0) {
                        /*
                         * Re-acquire svp if the driver changed *devp.
                         */

                        if (*devp != odev) {
                                if (svp != NULL)
                                        rw_exit(&svp->sv_lock);

                                svp = sv_dev_to_sv(*devp, NULL);

                                if (svp) {
                                        rw_enter(&svp->sv_lock, RW_READER);
```

```
                                }
                        }
                }
        } else {
                ret = ENODEV;
        }

        if (svp && ret != 0 && svp->sv_state == SV_ENABLE) {
                /*
                 * Underlying DDI open failed, but we have this
                 * device SV enabled.  If we can read some data
                 * from the device, fake a successful open (this
                 * probably means that this device is RDC'd and we
                 * are getting the data from the secondary node).
                 *
                 * The reserve must be done with NSC_TRY|NSC_NOWAIT to
                 * ensure that it does not deadlock if this open is
                 * coming from nskernd:get_bsize().
                 */
                rc = sv_reserve(svp->sv_fd,
                    NSC_TRY | NSC_NOWAIT | NSC_MULTI | NSC_PCATCH);
                if (rc == 0) {
                        tmph = NULL;

                        rc = nsc_alloc_buf(svp->sv_fd, 0, 1, NSC_READ, &tmph);
                        if (rc <= 0) {
                                /* success */
                                ret = 0;
                        }

                        if (tmph) {
                                (void) nsc_free_buf(tmph);
                                tmph = NULL;
                        }

                        nsc_release(svp->sv_fd);

                        /*
                         * Count the number of layered opens that we
                         * fake since we have to fake a matching number
                         * of closes (OTYP_LYR open/close calls must be
                         * paired).
                         */

                        if (ret == 0 && otyp == OTYP_LYR) {
                                mutex_enter(&svp->sv_olock);
                                svp->sv_openlcnt++;
                                mutex_exit(&svp->sv_olock);
                        }
                }
        }

        if (svp) {
                rw_exit(&svp->sv_lock);
        }

        return (ret);
}


static int
sv_lyr_close(dev_t dev, int flag, int otyp, cred_t *crp)
{
        sv_dev_t *svp;
        sv_maj_t *maj;
        int (*fn)();
```

```c
        int ret;

        svp = sv_dev_to_sv(dev, &maj);

        if (svp &&
            svp->sv_state == SV_PENDING &&
            svp->sv_pending == curthread) {
                /*
                 * This is a recursive open from a call to
                 * ddi_lyr_close and so we just want
                 * to pass it straight through to the
                 * underlying driver.
                 */
                DTRACE_PROBE2(sv_lyr_close_recursive, sv_dev_t *, svp,
                    dev_t, dev);
                svp = NULL;
        }

        if (svp) {
                rw_enter(&svp->sv_lock, RW_READER);

                if (otyp == OTYP_LYR) {
                        mutex_enter(&svp->sv_olock);

                        if (svp->sv_openlcnt) {
                                /*
                                 * Consume sufficient layered closes to
                                 * account for the opens that we faked
                                 * whilst the device was failed.
                                 */
                                svp->sv_openlcnt--;
                                mutex_exit(&svp->sv_olock);
                                rw_exit(&svp->sv_lock);

                                DTRACE_PROBE1(sv_lyr_close_end, dev_t, dev);

                                return (0);
                        }

                        mutex_exit(&svp->sv_olock);
                }
        }

        if (maj && (fn = maj->sm_close) != 0) {
                if (!(maj->sm_flag & D_MP)) {
                        UNSAFE_ENTER();
                        ret = (*fn)(dev, flag, otyp, crp);
                        UNSAFE_EXIT();
                } else {
                        ret = (*fn)(dev, flag, otyp, crp);
                }
        } else {
                ret = ENODEV;
        }

        if (svp) {
                rw_exit(&svp->sv_lock);
        }

        return (ret);
}


/*
 * Convert the specified dev_t into a locked and enabled sv_dev_t, or
 * return NULL.
```

```c
 */
static sv_dev_t *
sv_find_enabled(const dev_t dev, sv_maj_t **majpp)
{
        sv_dev_t *svp;

        while ((svp = sv_dev_to_sv(dev, majpp)) != NULL) {
                rw_enter(&svp->sv_lock, RW_READER);

                if (svp->sv_state == SV_ENABLE) {
                        /* locked and enabled */
                        break;
                }

                /*
                 * State was changed while waiting on the lock.
                 * Wait for a stable state.
                 */
                rw_exit(&svp->sv_lock);

                DTRACE_PROBE1(sv_find_enabled_retry, dev_t, dev);

                delay(2);
        }

        return (svp);
}


static int
sv_lyr_uio(dev_t dev, uio_t *uiop, cred_t *crp, int rw)
{
        sv_dev_t *svp;
        sv_maj_t *maj;
        int (*fn)();
        int rc;

        svp = sv_find_enabled(dev, &maj);
        if (svp == NULL) {
                if (maj) {
                        if (rw == NSC_READ)
                                fn = maj->sm_read;
                        else
                                fn = maj->sm_write;

                        if (fn != 0) {
                                if (!(maj->sm_flag & D_MP)) {
                                        UNSAFE_ENTER();
                                        rc = (*fn)(dev, uiop, crp);
                                        UNSAFE_EXIT();
                                } else {
                                        rc = (*fn)(dev, uiop, crp);
                                }
                        }

                        return (rc);
                } else {
                        return (ENODEV);
                }
        }

        ASSERT(RW_READ_HELD(&svp->sv_lock));

        if (svp->sv_flag == 0) {
                /*
                 * guard access mode
```

```
		 * - prevent user level access to the device
		 */
		DTRACE_PROBE1(sv_lyr_uio_err_guard, uio_t *, uiop);
		rc = EPERM;
		goto out;
	}

	if ((rc = sv_reserve(svp->sv_fd, NSC_MULTI|NSC_PCATCH)) != 0) {
		DTRACE_PROBE1(sv_lyr_uio_err_rsrv, uio_t *, uiop);
		goto out;
	}

	if (rw == NSC_READ)
		rc = nsc_uread(svp->sv_fd, uiop, crp);
	else
		rc = nsc_uwrite(svp->sv_fd, uiop, crp);

	nsc_release(svp->sv_fd);

out:
	rw_exit(&svp->sv_lock);

	return (rc);
}


static int
sv_lyr_read(dev_t dev, uio_t *uiop, cred_t *crp)
{
	return (sv_lyr_uio(dev, uiop, crp, NSC_READ));
}


static int
sv_lyr_write(dev_t dev, uio_t *uiop, cred_t *crp)
{
	return (sv_lyr_uio(dev, uiop, crp, NSC_WRITE));
}


/* ARGSUSED */

static int
sv_lyr_aread(dev_t dev, struct aio_req *aio, cred_t *crp)
{
	return (aphysio(sv_lyr_strategy,
	    anocancel, dev, B_READ, minphys, aio));
}


/* ARGSUSED */

static int
sv_lyr_awrite(dev_t dev, struct aio_req *aio, cred_t *crp)
{
	return (aphysio(sv_lyr_strategy,
	    anocancel, dev, B_WRITE, minphys, aio));
}


/*
 * Set up an array containing the list of raw path names
 * The array for the paths is svl and the size of the array is
 * in size.
 *
 * If there are more layered devices than will fit in the array,
```

```
 * the number of extra layered devices is returned.  Otherwise
 * zero is return.
 *
 * Input:
 *	svn	: array for paths
 *	size	: size of the array
 *
 * Output (extra):
 *	zero	: All paths fit in array
 *	>0	: Number of defined layered devices don't fit in array
 */

static int
sv_list(void *ptr, const int size, int *extra, const int ilp32)
{
	sv_name32_t *svn32;
	sv_name_t *svn;
	sv_dev_t *svp;
	int *mode, *nblocks;
	int i, index;
	char *path;

	*extra = 0;
	index = 0;

	if (ilp32)
		svn32 = ptr;
	else
		svn = ptr;

	mutex_enter(&sv_mutex);
	for (i = 0; i < sv_max_devices; i++) {
		svp = &sv_devs[i];

		rw_enter(&svp->sv_lock, RW_READER);

		if (svp->sv_state != SV_ENABLE) {
			rw_exit(&svp->sv_lock);
			continue;
		}

		if ((*extra) != 0 || ptr == NULL) {
			/* Another overflow entry */
			rw_exit(&svp->sv_lock);
			(*extra)++;
			continue;
		}

		if (ilp32) {
			nblocks = &svn32->svn_nblocks;
			mode = &svn32->svn_mode;
			path = svn32->svn_path;

			svn32->svn_timestamp = (uint32_t)svp->sv_timestamp;
			svn32++;
		} else {
			nblocks = &svn->svn_nblocks;
			mode = &svn->svn_mode;
			path = svn->svn_path;

			svn->svn_timestamp = svp->sv_timestamp;
			svn++;
		}

		(void) strcpy(path, nsc_pathname(svp->sv_fd));
		*nblocks = svp->sv_nblocks;
```

```
                *mode = svp->sv_flag;

                if (*nblocks == 0) {
                        if (sv_debug > 3)
                                cmn_err(CE_CONT, "!sv_list: need to reserve\n");

                        if (sv_reserve(svp->sv_fd, NSC_MULTI|NSC_PCATCH) == 0) {
                                *nblocks = svp->sv_nblocks;
                                nsc_release(svp->sv_fd);
                        }
                }

                if (++index >= size) {
                        /* Out of space */
                        (*extra)++;
                }

                rw_exit(&svp->sv_lock);
        }
        mutex_exit(&sv_mutex);

        if (index < size) {
                /* NULL terminated list */
                if (ilp32)
                        svn32->svn_path[0] = '\0';
                else
                        svn->svn_path[0] = '\0';
        }

        return (0);
}


static void
sv_thread_tune(int threads)
{
        int incr = (threads > 0) ? 1 : -1;
        int change = 0;
        int nthreads;

        ASSERT(MUTEX_HELD(&sv_mutex));

        if (sv_threads_extra) {
                /* keep track of any additional threads requested */
                if (threads > 0) {
                        sv_threads_extra += threads;
                        return;
                }
                threads = -threads;
                if (threads >= sv_threads_extra) {
                        threads -= sv_threads_extra;
                        sv_threads_extra = 0;
                        /* fall through to while loop */
                } else {
                        sv_threads_extra -= threads;
                        return;
                }
        } else if (threads > 0) {
                /*
                 * do not increase the number of threads beyond
                 * sv_threads_max when doing dynamic thread tuning
                 */
                nthreads = nst_nthread(sv_tset);
                if ((nthreads + threads) > sv_threads_max) {
                        sv_threads_extra = nthreads + threads - sv_threads_max;
                        threads = sv_threads_max - nthreads;
```

```
                        if (threads <= 0)
                                return;
                }
        }

        if (threads < 0)
                threads = -threads;

        while (threads--) {
                nthreads = nst_nthread(sv_tset);
                sv_threads_needed += incr;

                if (sv_threads_needed >= nthreads)
                        change += nst_add_thread(sv_tset, sv_threads_inc);
                else if ((sv_threads_needed <
                    (nthreads - (sv_threads_inc + sv_threads_hysteresis))) &&
                    ((nthreads - sv_threads_inc) >= sv_threads))
                        change -= nst_del_thread(sv_tset, sv_threads_inc);
        }

#ifdef DEBUG
        if (change) {
                cmn_err(CE_NOTE,
                    "!sv_thread_tune: threads needed %d, nthreads %d, "
                    "nthreads change %d",
                    sv_threads_needed, nst_nthread(sv_tset), change);
        }
#endif
}


/* ARGSUSED */
static int
svopen(dev_t *devp, int flag, int otyp, cred_t *crp)
{
        int rc;

        mutex_enter(&sv_mutex);
        rc = sv_init_devs();
        mutex_exit(&sv_mutex);

        return (rc);
}


/* ARGSUSED */
static int
svclose(dev_t dev, int flag, int otyp, cred_t *crp)
{
        const int secs = HZ * 5;
        const int ticks = HZ / 10;
        int loops = secs / ticks;

        mutex_enter(&sv_mutex);
        while (sv_ndevices <= 0 && sv_tset != NULL && loops > 0) {
                if (nst_nlive(sv_tset) <= 0) {
                        nst_destroy(sv_tset);
                        sv_tset = NULL;
                        break;
                }

                /* threads still active - wait for them to exit */
                mutex_exit(&sv_mutex);
                delay(ticks);
                loops--;
                mutex_enter(&sv_mutex);
        }
        mutex_exit(&sv_mutex);

        if (loops <= 0) {
                cmn_err(CE_WARN,
#ifndef DEBUG
                    /* do not write to console when non-DEBUG */
                    "!"
#endif
                    "sv:svclose: threads still active "
                    "after %d sec - leaking thread set", secs);
        }

        return (0);
}


static int
svioctl(dev_t dev, int cmd, intptr_t arg, int mode, cred_t *crp, int *rvalp)
{
        char itmp1[12], itmp2[12]; /* temp char array for editing ints */
        spcs_s_info_t kstatus;  /* Kernel version of spcs status */
        spcs_s_info_t ustatus;  /* Address of user version of spcs status */
        sv_list32_t svl32;      /* 32 bit Initial structure for SVIOC_LIST */
        sv_version_t svv;       /* Version structure */
        sv_conf_t svc;          /* User config structure */
        sv_list_t svl;          /* Initial structure for SVIOC_LIST */
        void *usvn;             /* Address of user sv_name_t */
        void *svn = NULL;       /* Array for SVIOC_LIST */
        uint64_t phash;         /* pathname hash */
        int rc = 0;             /* Return code -- errno */
        int size;               /* Number of items in array */
        int bytes;              /* Byte size of array */
        int ilp32;              /* Convert data structures for ilp32 userland */

        *rvalp = 0;

        /*
         * If sv_mod_status is 0 or SV_PREVENT_UNLOAD, then it will continue.
         * else it means it previously was SV_PREVENT_UNLOAD, and now it’s
         * SV_ALLOW_UNLOAD, expecting the driver to eventually unload.
         *
         * SV_ALLOW_UNLOAD is final state, so no need to grab sv_mutex.
         */
        if (sv_mod_status == SV_ALLOW_UNLOAD) {
                return (EBUSY);
        }

        if ((cmd != SVIOC_LIST) && ((rc = drv_priv(crp)) != 0))
                return (rc);

        kstatus = spcs_s_kcreate();
        if (!kstatus) {
                DTRACE_PROBE1(sv_ioctl_err_kcreate, dev_t, dev);
                return (ENOMEM);
        }

        ilp32 = (ddi_model_convert_from((mode & FMODELS)) == DDI_MODEL_ILP32);

        switch (cmd) {

        case SVIOC_ENABLE:

                if (ilp32) {
                        sv_conf32_t svc32;
```

```
                        if (ddi_copyin((void *)arg, &svc32,
                            sizeof (svc32), mode) < 0) {
                                spcs_s_kfree(kstatus);
                                return (EFAULT);
                        }

                        svc.svc_error = (spcs_s_info_t)svc32.svc_error;
                        (void) strcpy(svc.svc_path, svc32.svc_path);
                        svc.svc_flag  = svc32.svc_flag;
                        svc.svc_major = svc32.svc_major;
                        svc.svc_minor = svc32.svc_minor;
                } else {
                        if (ddi_copyin((void *)arg, &svc,
                            sizeof (svc), mode) < 0) {
                                spcs_s_kfree(kstatus);
                                return (EFAULT);
                        }
                }

                /* force to raw access */
                svc.svc_flag = NSC_DEVICE;

                if (sv_tset == NULL) {
                        mutex_enter(&sv_mutex);

                        if (sv_tset == NULL) {
                                sv_tset = nst_init("sv_thr", sv_threads);
                        }

                        mutex_exit(&sv_mutex);

                        if (sv_tset == NULL) {
                                cmn_err(CE_WARN,
                                    "!sv: could not allocate %d threads",
                                    sv_threads);
                        }
                }

                rc = sv_enable(svc.svc_path, svc.svc_flag,
                    makedevice(svc.svc_major, svc.svc_minor), kstatus);

                if (rc == 0) {
                        sv_config_time = nsc_lbolt();

                        mutex_enter(&sv_mutex);
                        sv_thread_tune(sv_threads_dev);
                        mutex_exit(&sv_mutex);
                }

                DTRACE_PROBE3(sv_ioctl_end, dev_t, dev, int, *rvalp, int, rc);

                return (spcs_s_ocopyoutf(&kstatus, svc.svc_error, rc));
                /* NOTREACHED */

        case SVIOC_DISABLE:

                if (ilp32) {
                        sv_conf32_t svc32;

                        if (ddi_copyin((void *)arg, &svc32,
                            sizeof (svc32), mode) < 0) {
                                spcs_s_kfree(kstatus);
                                return (EFAULT);
                        }

                        svc.svc_error = (spcs_s_info_t)svc32.svc_error;
```

```
			svc.svc_major = svc32.svc_major;
			svc.svc_minor = svc32.svc_minor;
			(void) strcpy(svc.svc_path, svc32.svc_path);
			svc.svc_flag  = svc32.svc_flag;
		} else {
			if (ddi_copyin((void *)arg, &svc,
			    sizeof (svc), mode) < 0) {
				spcs_s_kfree(kstatus);
				return (EFAULT);
			}
		}

		if (svc.svc_major == (major_t)-1 &&
		    svc.svc_minor == (minor_t)-1) {
			sv_dev_t *svp;
			int i;

			/*
			 * User level could not find the minor device
			 * node, so do this the slow way by searching
			 * the entire sv config for a matching pathname.
			 */

			phash = nsc_strhash(svc.svc_path);

			mutex_enter(&sv_mutex);

			for (i = 0; i < sv_max_devices; i++) {
				svp = &sv_devs[i];

				if (svp->sv_state == SV_DISABLE ||
				    svp->sv_fd == NULL)
					continue;

				if (nsc_fdpathcmp(svp->sv_fd, phash,
				    svc.svc_path) == 0) {
					svc.svc_major = getmajor(svp->sv_dev);
					svc.svc_minor = getminor(svp->sv_dev);
					break;
				}
			}

			mutex_exit(&sv_mutex);

			if (svc.svc_major == (major_t)-1 &&
			    svc.svc_minor == (minor_t)-1)
				return (spcs_s_ocopyoutf(&kstatus,
				    svc.svc_error, SV_ENODEV));
		}

		rc = sv_disable(makedevice(svc.svc_major, svc.svc_minor),
		    kstatus);

		if (rc == 0) {
			sv_config_time = nsc_lbolt();

			mutex_enter(&sv_mutex);
			sv_thread_tune(-sv_threads_dev);
			mutex_exit(&sv_mutex);
		}

		DTRACE_PROBE3(sv_ioctl_2, dev_t, dev, int, *rvalp, int, rc);

		return (spcs_s_ocopyoutf(&kstatus, svc.svc_error, rc));
		/* NOTREACHED */
```

```
	case SVIOC_LIST:

		if (ilp32) {
			if (ddi_copyin((void *)arg, &svl32,
			    sizeof (svl32), mode) < 0) {
				spcs_s_kfree(kstatus);
				return (EFAULT);
			}

			ustatus = (spcs_s_info_t)svl32.svl_error;
			size = svl32.svl_count;
			usvn = (void *)(unsigned long)svl32.svl_names;
		} else {
			if (ddi_copyin((void *)arg, &svl,
			    sizeof (svl), mode) < 0) {
				spcs_s_kfree(kstatus);
				return (EFAULT);
			}

			ustatus = svl.svl_error;
			size = svl.svl_count;
			usvn = svl.svl_names;
		}

		/* Do some boundary checking */
		if ((size < 0) || (size > sv_max_devices)) {
			/* Array size is out of range */
			return (spcs_s_ocopyoutf(&kstatus, ustatus,
			    SV_EARRBOUNDS, "0",
			    spcs_s_inttostring(sv_max_devices, itmp1,
			    sizeof (itmp1), 0),
			    spcs_s_inttostring(size, itmp2,
			    sizeof (itmp2), 0)));
		}

		if (ilp32)
			bytes = size * sizeof (sv_name32_t);
		else
			bytes = size * sizeof (sv_name_t);

		/* Allocate memory for the array of structures */
		if (bytes != 0) {
			svn = kmem_zalloc(bytes, KM_SLEEP);
			if (!svn) {
				return (spcs_s_ocopyoutf(&kstatus,
				    ustatus, ENOMEM));
			}
		}

		rc = sv_list(svn, size, rvalp, ilp32);
		if (rc) {
			if (svn != NULL)
				kmem_free(svn, bytes);
			return (spcs_s_ocopyoutf(&kstatus, ustatus, rc));
		}

		if (ilp32) {
			svl32.svl_timestamp = (uint32_t)sv_config_time;
			svl32.svl_maxdevs = (int32_t)sv_max_devices;

			/* Return the list structure */
			if (ddi_copyout(&svl32, (void *)arg,
			    sizeof (svl32), mode) < 0) {
				spcs_s_kfree(kstatus);
				if (svn != NULL)
					kmem_free(svn, bytes);
```

```
                                return (EFAULT);
                        }
                } else {
                        svl.svl_timestamp = sv_config_time;
                        svl.svl_maxdevs = sv_max_devices;

                        /* Return the list structure */
                        if (ddi_copyout(&svl, (void *)arg,
                            sizeof (svl), mode) < 0) {
                                spcs_s_kfree(kstatus);
                                if (svn != NULL)
                                        kmem_free(svn, bytes);
                                return (EFAULT);
                        }
                }

                /* Return the array */
                if (svn != NULL) {
                        if (ddi_copyout(svn, usvn, bytes, mode) < 0) {
                                kmem_free(svn, bytes);
                                spcs_s_kfree(kstatus);
                                return (EFAULT);
                        }
                        kmem_free(svn, bytes);
                }

                DTRACE_PROBE3(sv_ioctl_3, dev_t, dev, int, *rvalp, int, 0);

                return (spcs_s_ocopyoutf(&kstatus, ustatus, 0));
                /* NOTREACHED */

        case SVIOC_VERSION:

                if (ilp32) {
                        sv_version32_t svv32;

                        if (ddi_copyin((void *)arg, &svv32,
                            sizeof (svv32), mode) < 0) {
                                spcs_s_kfree(kstatus);
                                return (EFAULT);
                        }

                        svv32.svv_major_rev = sv_major_rev;
                        svv32.svv_minor_rev = sv_minor_rev;
                        svv32.svv_micro_rev = sv_micro_rev;
                        svv32.svv_baseline_rev = sv_baseline_rev;

                        if (ddi_copyout(&svv32, (void *)arg,
                            sizeof (svv32), mode) < 0) {
                                spcs_s_kfree(kstatus);
                                return (EFAULT);
                        }

                        ustatus = (spcs_s_info_t)svv32.svv_error;
                } else {
                        if (ddi_copyin((void *)arg, &svv,
                            sizeof (svv), mode) < 0) {
                                spcs_s_kfree(kstatus);
                                return (EFAULT);
                        }

                        svv.svv_major_rev = sv_major_rev;
                        svv.svv_minor_rev = sv_minor_rev;
                        svv.svv_micro_rev = sv_micro_rev;
                        svv.svv_baseline_rev = sv_baseline_rev;
```

```
                        if (ddi_copyout(&svv, (void *)arg,
                            sizeof (svv), mode) < 0) {
                                spcs_s_kfree(kstatus);
                                return (EFAULT);
                        }

                        ustatus = svv.svv_error;
                }

                DTRACE_PROBE3(sv_ioctl_4, dev_t, dev, int, *rvalp, int, 0);

                return (spcs_s_ocopyoutf(&kstatus, ustatus, 0));
                /* NOTREACHED */

        case SVIOC_UNLOAD:
                rc = sv_prepare_unload();

                if (ddi_copyout(&rc, (void *)arg, sizeof (rc), mode) < 0) {
                        rc = EFAULT;
                }

                spcs_s_kfree(kstatus);
                return (rc);

        default:
                spcs_s_kfree(kstatus);

                DTRACE_PROBE3(sv_ioctl_4, dev_t, dev, int, *rvalp, int, EINVAL);

                return (EINVAL);
                /* NOTREACHED */
        }

        /* NOTREACHED */
}


/* ARGSUSED */
static int
svprint(dev_t dev, char *str)
{
        int instance = ddi_get_instance(sv_dip);
        cmn_err(CE_WARN, "!%s%d: %s", ddi_get_name(sv_dip), instance, str);
        return (0);
}


static void
_sv_lyr_strategy(struct buf *bp)
{
        caddr_t buf_addr;               /* pointer to linear buffer in bp */
        nsc_buf_t *bufh = NULL;
        nsc_buf_t *hndl = NULL;
        sv_dev_t *svp;
        nsc_vec_t *v;
        sv_maj_t *maj;
        nsc_size_t fba_req, fba_len;    /* FBA lengths */
        nsc_off_t fba_off;              /* FBA offset */
        size_t tocopy, nbytes;          /* byte lengths */
        int rw, rc;                     /* flags and return codes */
        int (*fn)();

        rc = 0;

        if (sv_debug > 5)
                cmn_err(CE_CONT, "!_sv_lyr_strategy(%p)\n", (void *)bp);
        svp = sv_find_enabled(bp->b_edev, &maj);
        if (svp == NULL) {
                if (maj && (fn = maj->sm_strategy) != 0) {
                        if (!(maj->sm_flag & D_MP)) {
                                UNSAFE_ENTER();
                                rc = (*fn)(bp);
                                UNSAFE_EXIT();
                        } else {
                                rc = (*fn)(bp);
                        }
                        return;
                } else {
                        bioerror(bp, ENODEV);
                        biodone(bp);
                        return;
                }
        }

        ASSERT(RW_READ_HELD(&svp->sv_lock));

        if (svp->sv_flag == 0) {
                /*
                 * guard access mode
                 * - prevent user level access to the device
                 */
                DTRACE_PROBE1(sv_lyr_strategy_err_guard, struct buf *, bp);
                bioerror(bp, EPERM);
                goto out;
        }

        if ((rc = sv_reserve(svp->sv_fd, NSC_MULTI|NSC_PCATCH)) != 0) {
                DTRACE_PROBE1(sv_lyr_strategy_err_rsrv, struct buf *, bp);

                if (rc == EINTR)
                        cmn_err(CE_WARN, "!nsc_reserve() returned EINTR");
                bioerror(bp, rc);
                goto out;
        }

        if (bp->b_lblkno >= (diskaddr_t)svp->sv_nblocks) {
                DTRACE_PROBE1(sv_lyr_strategy_eof, struct buf *, bp);

                if (bp->b_flags & B_READ) {
                        /* return EOF, not an error */
                        bp->b_resid = bp->b_bcount;
                        bioerror(bp, 0);
                } else
                        bioerror(bp, EINVAL);

                goto done;
        }

        /*
         * Preallocate a handle once per call to strategy.
         * If this fails, then the nsc_alloc_buf() will allocate
         * a temporary handle per allocation/free pair.
         */

        DTRACE_PROBE1(sv_dbg_alloch_start, sv_dev_t *, svp);

        bufh = nsc_alloc_handle(svp->sv_fd, NULL, NULL, NULL);

        DTRACE_PROBE1(sv_dbg_alloch_end, sv_dev_t *, svp);

        if (bufh && (bufh->sb_flag & NSC_HACTIVE) != 0) {
```

```
                DTRACE_PROBE1(sv_lyr_strategy_err_hactive, struct buf *, bp);

                cmn_err(CE_WARN,
                    "!sv: allocated active handle (bufh %p, flags %x)",
                    (void *)bufh, bufh->sb_flag);

                bioerror(bp, ENXIO);
                goto done;
        }

        fba_req = FBA_LEN(bp->b_bcount);
        if (fba_req + bp->b_lblkno > (diskaddr_t)svp->sv_nblocks)
                fba_req = (nsc_size_t)(svp->sv_nblocks - bp->b_lblkno);

        rw = (bp->b_flags & B_READ) ? NSC_READ : NSC_WRITE;

        bp_mapin(bp);

        bp->b_resid = bp->b_bcount;
        buf_addr = bp->b_un.b_addr;
        fba_off = 0;

        /*
         * fba_req  - requested size of transfer in FBAs after
         *              truncation to device extent, and allowing for
         *              possible non-FBA bounded final chunk.
         * fba_off  - offset of start of chunk from start of bp in FBAs.
         * fba_len  - size of this chunk in FBAs.
         */

loop:
        fba_len = min(fba_req, svp->sv_maxfbas);
        hndl = bufh;

        DTRACE_PROBE4(sv_dbg_allocb_start,
            sv_dev_t *, svp,
            uint64_t, (uint64_t)(bp->b_lblkno + fba_off),
            uint64_t, (uint64_t)fba_len,
            int, rw);

        rc = nsc_alloc_buf(svp->sv_fd, (nsc_off_t)(bp->b_lblkno + fba_off),
            fba_len, rw, &hndl);

        DTRACE_PROBE1(sv_dbg_allocb_end, sv_dev_t *, svp);

        if (rc > 0) {
                DTRACE_PROBE1(sv_lyr_strategy_err_alloc, struct buf *, bp);
                bioerror(bp, rc);
                if (hndl != bufh)
                        (void) nsc_free_buf(hndl);
                hndl = NULL;
                goto done;
        }

        tocopy = min(FBA_SIZE(fba_len), bp->b_resid);
        v = hndl->sb_vec;

        if (rw == NSC_WRITE && FBA_OFF(tocopy) != 0) {
                /*
                 * Not overwriting all of the last FBA, so read in the
                 * old contents now before we overwrite it with the new
                 * data.
                 */

                DTRACE_PROBE2(sv_dbg_read_start, sv_dev_t *, svp,
                    uint64_t, (uint64_t)(hndl->sb_pos + hndl->sb_len - 1));
```

```
		rc = nsc_read(hndl, (hndl->sb_pos + hndl->sb_len - 1), 1, 0);
		if (rc > 0) {
			bioerror(bp, rc);
			goto done;
		}

		DTRACE_PROBE1(sv_dbg_read_end, sv_dev_t *, svp);
	}

	DTRACE_PROBE1(sv_dbg_bcopy_start, sv_dev_t *, svp);

	while (tocopy > 0) {
		nbytes = min(tocopy, (nsc_size_t)v->sv_len);

		if (bp->b_flags & B_READ)
			(void) bcopy(v->sv_addr, buf_addr, nbytes);
		else
			(void) bcopy(buf_addr, v->sv_addr, nbytes);

		bp->b_resid -= nbytes;
		buf_addr += nbytes;
		tocopy -= nbytes;
		v++;
	}

	DTRACE_PROBE1(sv_dbg_bcopy_end, sv_dev_t *, svp);

	if ((bp->b_flags & B_READ) == 0) {
		DTRACE_PROBE3(sv_dbg_write_start, sv_dev_t *, svp,
		    uint64_t, (uint64_t)hndl->sb_pos,
		    uint64_t, (uint64_t)hndl->sb_len);

		rc = nsc_write(hndl, hndl->sb_pos, hndl->sb_len, 0);

		DTRACE_PROBE1(sv_dbg_write_end, sv_dev_t *, svp);

		if (rc > 0) {
			bioerror(bp, rc);
			goto done;
		}
	}

	/*
	 * Adjust FBA offset and requested (ie. remaining) length,
	 * loop if more data to transfer.
	 */

	fba_off += fba_len;
	fba_req -= fba_len;

	if (fba_req > 0) {
		DTRACE_PROBE1(sv_dbg_freeb_start, sv_dev_t *, svp);

		rc = nsc_free_buf(hndl);

		DTRACE_PROBE1(sv_dbg_freeb_end, sv_dev_t *, svp);

		if (rc > 0) {
			DTRACE_PROBE1(sv_lyr_strategy_err_free,
			    struct buf *, bp);
			bioerror(bp, rc);
		}

		hndl = NULL;
```

```
		if (rc <= 0)
			goto loop;
	}

done:
	if (hndl != NULL) {
		DTRACE_PROBE1(sv_dbg_freeb_start, sv_dev_t *, svp);

		rc = nsc_free_buf(hndl);

		DTRACE_PROBE1(sv_dbg_freeb_end, sv_dev_t *, svp);

		if (rc > 0) {
			DTRACE_PROBE1(sv_lyr_strategy_err_free,
			    struct buf *, bp);
			bioerror(bp, rc);
		}

		hndl = NULL;
	}

	if (bufh)
		(void) nsc_free_handle(bufh);

	DTRACE_PROBE1(sv_dbg_rlse_start, sv_dev_t *, svp);

	nsc_release(svp->sv_fd);

	DTRACE_PROBE1(sv_dbg_rlse_end, sv_dev_t *, svp);

out:
	if (sv_debug > 5) {
		cmn_err(CE_CONT,
		    "!_sv_lyr_strategy: bp %p, bufh %p, bp->b_error %d\n",
		    (void *)bp, (void *)bufh, bp->b_error);
	}

	DTRACE_PROBE2(sv_lyr_strategy_end, struct buf *, bp, int, bp->b_error);

	rw_exit(&svp->sv_lock);
	biodone(bp);
}


static void
sv_async_strategy(blind_t arg)
{
	struct buf *bp = (struct buf *)arg;
	_sv_lyr_strategy(bp);
}


static int
sv_lyr_strategy(struct buf *bp)
{
	nsthread_t *tp;
	int nlive;

	/*
	 * If B_ASYNC was part of the DDI we could use it as a hint to
	 * not create a thread for synchronous i/o.
	 */
	if (sv_dev_to_sv(bp->b_edev, NULL) == NULL) {
		/* not sv enabled - just pass through */
		DTRACE_PROBE1(sv_lyr_strategy_notsv, struct buf *, bp);
		_sv_lyr_strategy(bp);
```

```
                return (0);
        }

        if (sv_debug > 4) {
                cmn_err(CE_CONT, "!sv_lyr_strategy: nthread %d nlive %d\n",
                    nst_nthread(sv_tset), nst_nlive(sv_tset));
        }

        /*
         * If there are only guard devices enabled there
         * won't be a threadset, so don't try and use it.
         */
        tp = NULL;
        if (sv_tset != NULL) {
                tp = nst_create(sv_tset, sv_async_strategy, (blind_t)bp, 0);
        }

        if (tp == NULL) {
                /*
                 * out of threads, so fall back to synchronous io.
                 */
                if (sv_debug > 0) {
                        cmn_err(CE_CONT,
                            "!sv_lyr_strategy: thread alloc failed\n");
                }

                DTRACE_PROBE1(sv_lyr_strategy_no_thread,
                    struct buf *, bp);

                _sv_lyr_strategy(bp);
                sv_no_threads++;
        } else {
                nlive = nst_nlive(sv_tset);
                if (nlive > sv_max_nlive) {
                        if (sv_debug > 0) {
                                cmn_err(CE_CONT,
                                    "!sv_lyr_strategy: "
                                    "new max nlive %d (nthread %d)\n",
                                    nlive, nst_nthread(sv_tset));
                        }

                        sv_max_nlive = nlive;
                }
        }

        return (0);
}


#ifndef offsetof
#define offsetof(s, m)  ((size_t)(&((s *)0)->m))
#endif

/*
 * re-write the size of the current partition
 */
static int
sv_fix_dkiocgvtoc(const intptr_t arg, const int mode, sv_dev_t *svp)
{
        size_t offset;
        int ilp32;
        int pnum;
        int rc;

        ilp32 = (ddi_model_convert_from((mode & FMODELS)) == DDI_MODEL_ILP32);
```

```
        rc = nskern_partition(svp->sv_dev, &pnum);
        if (rc != 0) {
                return (rc);
        }

        if (pnum < 0 || pnum >= V_NUMPAR) {
                cmn_err(CE_WARN,
                    "!sv_gvtoc: unable to determine partition number "
                    "for dev %lx", svp->sv_dev);
                return (EINVAL);
        }

        if (ilp32) {
                int32_t p_size;

#ifdef _SunOS_5_6
                offset = offsetof(struct vtoc, v_part);
                offset += sizeof (struct partition) * pnum;
                offset += offsetof(struct partition, p_size);
#else
                offset = offsetof(struct vtoc32, v_part);
                offset += sizeof (struct partition32) * pnum;
                offset += offsetof(struct partition32, p_size);
#endif

                p_size = (int32_t)svp->sv_nblocks;
                if (p_size == 0) {
                        if (sv_reserve(svp->sv_fd,
                            NSC_MULTI|NSC_PCATCH) == 0) {
                                p_size = (int32_t)svp->sv_nblocks;
                                nsc_release(svp->sv_fd);
                        } else {
                                rc = EINTR;
                        }
                }

                if ((rc == 0) && ddi_copyout(&p_size, (void *)(arg + offset),
                    sizeof (p_size), mode) != 0) {
                        rc = EFAULT;
                }
        } else {
                long p_size;

                offset = offsetof(struct vtoc, v_part);
                offset += sizeof (struct partition) * pnum;
                offset += offsetof(struct partition, p_size);

                p_size = (long)svp->sv_nblocks;
                if (p_size == 0) {
                        if (sv_reserve(svp->sv_fd,
                            NSC_MULTI|NSC_PCATCH) == 0) {
                                p_size = (long)svp->sv_nblocks;
                                nsc_release(svp->sv_fd);
                        } else {
                                rc = EINTR;
                        }
                }

                if ((rc == 0) && ddi_copyout(&p_size, (void *)(arg + offset),
                    sizeof (p_size), mode) != 0) {
                        rc = EFAULT;
                }
        }

        return (rc);
}
______unchanged_portion_omitted_
```

```
**********************************************************
   50433 Thu Feb 25 15:39:39 2016
new/usr/src/uts/common/inet/ipf/netinet/ip_fil.h
2976 remove useless offsetof() macros
**********************************************************
______unchanged_portion_omitted_

#define fr_caddr        fr_dun.fru_caddr
#define fr_data         fr_dun.fru_data
#define fr_dfunc        fr_dun.fru_func
#define fr_ipf          fr_dun.fru_ipf
#define fr_ip           fr_ipf->fri_ip
#define fr_mip          fr_ipf->fri_mip
#define fr_icmpm        fr_ipf->fri_icmpm
#define fr_icmp         fr_ipf->fri_icmp
#define fr_tuc          fr_ipf->fri_tuc
#define fr_satype       fr_ipf->fri_satype
#define fr_datype       fr_ipf->fri_datype
#define fr_sifpidx      fr_ipf->fri_sifpidx
#define fr_difpidx      fr_ipf->fri_difpidx
#define fr_proto        fr_ip.fi_p
#define fr_mproto       fr_mip.fi_p
#define fr_ttl          fr_ip.fi_ttl
#define fr_mttl         fr_mip.fi_ttl
#define fr_tos          fr_ip.fi_tos
#define fr_mtos         fr_mip.fi_tos
#define fr_tcpfm        fr_tuc.ftu_tcpfm
#define fr_tcpf         fr_tuc.ftu_tcpf
#define fr_scmp         fr_tuc.ftu_scmp
#define fr_dcmp         fr_tuc.ftu_dcmp
#define fr_dport        fr_tuc.ftu_dport
#define fr_sport        fr_tuc.ftu_sport
#define fr_stop         fr_tuc.ftu_stop
#define fr_dtop         fr_tuc.ftu_dtop
#define fr_dst          fr_ip.fi_dst.in4
#define fr_daddr        fr_ip.fi_dst.in4.s_addr
#define fr_src          fr_ip.fi_src.in4
#define fr_saddr        fr_ip.fi_src.in4.s_addr
#define fr_dmsk         fr_mip.fi_dst.in4
#define fr_dmask        fr_mip.fi_dst.in4.s_addr
#define fr_smsk         fr_mip.fi_src.in4
#define fr_smask        fr_mip.fi_src.in4.s_addr
#define fr_dstnum       fr_ip.fi_dstnum
#define fr_srcnum       fr_ip.fi_srcnum
#define fr_dsttype      fr_ip.fi_dsttype
#define fr_srctype      fr_ip.fi_srctype
#define fr_dstptr       fr_mip.fi_dstptr
#define fr_srcptr       fr_mip.fi_srcptr
#define fr_dstfunc      fr_mip.fi_dstfunc
#define fr_srcfunc      fr_mip.fi_srcfunc
#define fr_optbits      fr_ip.fi_optmsk
#define fr_optmask      fr_mip.fi_optmsk
#define fr_secbits      fr_ip.fi_secmsk
#define fr_secmask      fr_mip.fi_secmsk
#define fr_authbits     fr_ip.fi_auth
#define fr_authmask     fr_mip.fi_auth
#define fr_flx          fr_ip.fi_flx
#define fr_mflx         fr_mip.fi_flx
#define fr_ifname       fr_ifnames[0]
#define fr_oifname      fr_ifnames[2]
#define fr_ifa          fr_ifas[0]
#define fr_oifa         fr_ifas[2]
#define fr_tif          fr_tifs[0]
#define fr_rif          fr_tifs[1]

#define FR_NOLOGTAG     0
```

```
#ifndef offsetof
#define offsetof(t,m)   (size_t)((&((t *)0)->m))
#endif
#define FR_CMPSIZ       (sizeof(struct frentry) - \
                         offsetof(struct frentry, fr_func))

/*
 * fr_type
 */
#define FR_T_NONE       0
#define FR_T_IPF        1       /* IPF structures */
#define FR_T_BPFOPC     2       /* BPF opcode */
#define FR_T_CALLFUNC   3       /* callout to function in fr_func only */
#define FR_T_COMPIPF    4       /* compiled C code */
#define FR_T_BUILTIN    0x80000000      /* rule is in kernel space */

/*
 * fr_flags
 */
#define FR_CALL         0x00000 /* call rule */
#define FR_BLOCK        0x00001 /* do not allow packet to pass */
#define FR_PASS         0x00002 /* allow packet to pass */
#define FR_AUTH         0x00003 /* use authentication */
#define FR_PREAUTH      0x00004 /* require preauthentication */
#define FR_ACCOUNT      0x00005 /* Accounting rule */
#define FR_SKIP         0x00006 /* skip rule */
#define FR_DIVERT       0x00007 /* divert rule */
#define FR_CMDMASK      0x0000f
#define FR_LOG          0x00010 /* Log */
#define FR_LOGB         0x00011 /* Log-fail */
#define FR_LOGP         0x00012 /* Log-pass */
#define FR_LOGMASK      (FR_LOG|FR_CMDMASK)
#define FR_CALLNOW      0x00020 /* call another function (fr_func) if matches */
#define FR_NOTSRCIP     0x00040
#define FR_NOTDSTIP     0x00080
#define FR_QUICK        0x00100 /* match & stop processing list */
#define FR_KEEPFRAG     0x00200 /* keep fragment information */
#define FR_KEEPSTATE    0x00400 /* keep ‘connection’ state information */
#define FR_FASTROUTE    0x00800 /* bypass normal routing */
#define FR_RETRST       0x01000 /* Return TCP RST packet - reset connection */
#define FR_RETICMP      0x02000 /* Return ICMP unreachable packet */
#define FR_FAKEICMP     0x03000 /* Return ICMP unreachable with fake source */
#define FR_OUTQUE       0x04000 /* outgoing packets */
#define FR_INQUE        0x08000 /* ingoing packets */
#define FR_LOGBODY      0x10000 /* Log the body */
#define FR_LOGFIRST     0x20000 /* Log the first byte if state held */
#define FR_LOGORBLOCK   0x40000 /* block the packet if it can’t be logged */
#define FR_DUP          0x80000 /* duplicate packet */
#define FR_FRSTRICT     0x100000        /* strict frag. cache */
#define FR_STSTRICT     0x200000        /* strict keep state */
#define FR_NEWISN       0x400000        /* new ISN for outgoing TCP */
#define FR_NOICMPERR    0x800000        /* do not match ICMP errors in state */
#define FR_STATESYNC    0x1000000       /* synchronize state to slave */
#define FR_NOMATCH      0x8000000       /* no match occured */
                /*      0x10000000      FF_LOGPASS */
                /*      0x20000000      FF_LOGBLOCK */
                /*      0x40000000      FF_LOGNOMATCH */
                /*      0x80000000      FF_BLOCKNONIP */
#define FR_COPIED       0x40000000      /* copied from user space */
#define FR_INACTIVE     0x80000000      /* only used when flush’ing rules */

#define FR_RETMASK      (FR_RETICMP|FR_RETRST|FR_FAKEICMP)
#define FR_ISBLOCK(x)   (((x) & FR_CMDMASK) == FR_BLOCK)
#define FR_ISPASS(x)    (((x) & FR_CMDMASK) == FR_PASS)
#define FR_ISAUTH(x)    (((x) & FR_CMDMASK) == FR_AUTH)
```

```
#define FR_ISPREAUTH(x) (((x) & FR_CMDMASK) == FR_PREAUTH)
#define FR_ISACCOUNT(x) (((x) & FR_CMDMASK) == FR_ACCOUNT)
#define FR_ISSKIP(x)    (((x) & FR_CMDMASK) == FR_SKIP)
#define FR_ISNOMATCH(x) ((x) & FR_NOMATCH)
#define FR_INOUT        (FR_INQUE|FR_OUTQUE)

/*
 * recognized flags for SIOCGETFF and SIOCSETFF, and get put in fr_flags
 */
#define FF_LOGPASS      0x10000000
#define FF_LOGBLOCK     0x20000000
#define FF_LOGNOMATCH   0x40000000
#define FF_LOGGING      (FF_LOGPASS|FF_LOGBLOCK|FF_LOGNOMATCH)
#define FF_BLOCKNONIP   0x80000000      /* Solaris2 Only */


/*
 * Structure that passes information on what/how to flush to the kernel.
 */
typedef struct  ipfflush        {
        int     ipflu_how;
        int     ipflu_arg;
} ipfflush_t;
```
______***unchanged_portion_omitted_***

```
**********************************************************
   33099 Thu Feb 25 15:39:39 2016
new/usr/src/uts/common/io/drm/drmP.h
2976 remove useless offsetof() macros
**********************************************************
/*
 * drmP.h -- Private header for Direct Rendering Manager -*- linux-c -*-
 * Created: Mon Jan  4 10:05:05 1999 by faith@precisioninsight.com
 */
/*
 * Copyright 1999 Precision Insight, Inc., Cedar Park, Texas.
 * Copyright 2000 VA Linux Systems, Inc., Sunnyvale, California.
 * Copyright (c) 2009, Intel Corporation.
 * All rights reserved.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and associated documentation files (the "Software"),
 * to deal in the Software without restriction, including without limitation
 * the rights to use, copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit persons to whom the
 * Software is furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice (including the next
 * paragraph) shall be included in all copies or substantial portions of the
 * Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL
 * VA LINUX SYSTEMS AND/OR ITS SUPPLIERS BE LIABLE FOR ANY CLAIM, DAMAGES OR
 * OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
 * ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * Authors:
 *    Rickard E. (Rik) Faith <faith@valinux.com>
 *    Gareth Hughes <gareth@valinux.com>
 *
 */

/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _DRMP_H
#define _DRMP_H

#include <sys/sysmacros.h>
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/modctl.h>
#include <sys/stat.h>
#include <sys/file.h>
#include <sys/cmn_err.h>
#include <sys/varargs.h>
#include <sys/pci.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/sunldi.h>
#include <sys/pmem.h>
#include <sys/agpgart.h>
#include <sys/time.h>
#include <sys/sysmacros.h>
#endif /* ! codereview */
#include "drm_atomic.h"
```

```
#include "drm.h"
#include "queue.h"
#include "drm_linux_list.h"

#ifndef __inline__
#define __inline__      inline
#endif

#if !defined(__FUNCTION__)
#if defined(C99)
#define __FUNCTION__ __func__
#else
#define __FUNCTION__    " "
#endif
#endif

/* DRM space units */
#define DRM_PAGE_SHIFT                  PAGESHIFT
#define DRM_PAGE_SIZE                   (1 << DRM_PAGE_SHIFT)
#define DRM_PAGE_OFFSET                 (DRM_PAGE_SIZE - 1)
#define DRM_PAGE_MASK                   ~(DRM_PAGE_SIZE - 1)
#define DRM_MB2PAGES(x)                 ((x) << 8)
#define DRM_PAGES2BYTES(x)              ((x) << DRM_PAGE_SHIFT)
#define DRM_BYTES2PAGES(x)              ((x) >> DRM_PAGE_SHIFT)
#define DRM_PAGES2KB(x)                 ((x) << 2)
#define DRM_ALIGNED(offset)             (((offset) & DRM_PAGE_OFFSET) == 0)

#define PAGE_SHIFT                      DRM_PAGE_SHIFT
#define PAGE_SIZE                       DRM_PAGE_SIZE

#define DRM_MAX_INSTANCES       8
#define DRM_DEVNODE             "drm"
#define DRM_UNOPENED            0
#define DRM_OPENED              1

#define DRM_HASH_SIZE           16 /* Size of key hash table */
#define DRM_KERNEL_CONTEXT      0  /* Change drm_resctx if changed */
#define DRM_RESERVED_CONTEXTS   1  /* Change drm_resctx if changed */

#define DRM_MEM_DMA        0
#define DRM_MEM_SAREA      1
#define DRM_MEM_DRIVER     2
#define DRM_MEM_MAGIC      3
#define DRM_MEM_IOCTLS     4
#define DRM_MEM_MAPS       5
#define DRM_MEM_BUFS       6
#define DRM_MEM_SEGS       7
#define DRM_MEM_PAGES      8
#define DRM_MEM_FILES     9
#define DRM_MEM_QUEUES    10
#define DRM_MEM_CMDS      11
#define DRM_MEM_MAPPINGS  12
#define DRM_MEM_BUFLISTS  13
#define DRM_MEM_DRMLISTS  14
#define DRM_MEM_TOTALDRM  15
#define DRM_MEM_BOUNDDRM  16
#define DRM_MEM_CTXBITMAP 17
#define DRM_MEM_STUB      18
#define DRM_MEM_SGLISTS   19
#define DRM_MEM_AGPLISTS  20
#define DRM_MEM_CTXLIST   21
#define DRM_MEM_MM              22
#define DRM_MEM_HASHTAB         23
#define DRM_MEM_OBJECTS         24

#define DRM_MAX_CTXBITMAP (PAGE_SIZE * 8)
```

```c
#define DRM_MAP_HASH_OFFSET 0x10000000
#define DRM_MAP_HASH_ORDER 12
#define DRM_OBJECT_HASH_ORDER 12
#define DRM_FILE_PAGE_OFFSET_START ((0xFFFFFFFFUL >> PAGE_SHIFT) + 1)
#define DRM_FILE_PAGE_OFFSET_SIZE ((0xFFFFFFFFUL >> PAGE_SHIFT) * 16)
#define DRM_MM_INIT_MAX_PAGES 256


/* Internal types and structures */
#define DRM_ARRAY_SIZE(x) (sizeof (x) / sizeof (x[0]))
#define DRM_MIN(a, b) ((a) < (b) ? (a) : (b))
#define DRM_MAX(a, b) ((a) > (b) ? (a) : (b))

#define DRM_IF_VERSION(maj, min) (maj << 16 | min)

#define __OS_HAS_AGP    1

#define DRM_DEV_MOD     (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP)
#define DRM_DEV_UID     0
#define DRM_DEV_GID     0

#define DRM_CURRENTPID          ddi_get_pid()
#define DRM_SPINLOCK(l)         mutex_enter(l)
#define DRM_SPINUNLOCK(u)       mutex_exit(u)
#define DRM_SPINLOCK_ASSERT(l)
#define DRM_LOCK()      mutex_enter(&dev->dev_lock)
#define DRM_UNLOCK()    mutex_exit(&dev->dev_lock)
#define DRM_LOCK_OWNED()        ASSERT(mutex_owned(&dev->dev_lock))
#define spin_lock_irqsave(l, flag)              mutex_enter(l)
#define spin_unlock_irqrestore(u, flag) mutex_exit(u)
#define spin_lock(l)    mutex_enter(l)
#define spin_unlock(u)  mutex_exit(u)


#define DRM_UDELAY(sec)  delay(drv_usectohz(sec *1000))
#define DRM_MEMORYBARRIER()

typedef struct drm_file         drm_file_t;
typedef struct drm_device       drm_device_t;
typedef struct drm_driver_info drm_driver_t;

#define DRM_DEVICE      drm_device_t *dev = dev1
#define DRM_IOCTL_ARGS  \
        drm_device_t *dev1, intptr_t data, drm_file_t *fpriv, int mode

#define DRM_COPYFROM_WITH_RETURN(dest, src, size)       \
        if (ddi_copyin((src), (dest), (size), 0)) {     \
                DRM_ERROR("%s: copy from user failed", __func__);       \
                return (EFAULT);        \
        }

#define DRM_COPYTO_WITH_RETURN(dest, src, size) \
        if (ddi_copyout((src), (dest), (size), 0)) {    \
                DRM_ERROR("%s: copy to user failed", __func__); \
                return (EFAULT);        \
        }

#define DRM_COPY_FROM_USER(dest, src, size) \
        ddi_copyin((src), (dest), (size), 0) /* flag for src */

#define DRM_COPY_TO_USER(dest, src, size) \
        ddi_copyout((src), (dest), (size), 0) /* flags for dest */

#define DRM_COPY_FROM_USER_UNCHECKED(arg1, arg2, arg3)  \
        ddi_copyin((arg2), (arg1), (arg3), 0)
```

```c
#define DRM_COPY_TO_USER_UNCHECKED(arg1, arg2, arg3)        \
        ddi_copyout((arg2), arg1, arg3, 0)

#define DRM_READ8(map, offset) \
        *(volatile uint8_t *)((uintptr_t)((map)->dev_addr) + (offset))
#define DRM_READ16(map, offset) \
        *(volatile uint16_t *)((uintptr_t)((map)->dev_addr) + (offset))
#define DRM_READ32(map, offset) \
        *(volatile uint32_t *)((uintptr_t)((map)->dev_addr) + (offset))
#define DRM_WRITE8(map, offset, val) \
        *(volatile uint8_t *)((uintptr_t)((map)->dev_addr) + (offset)) = (val)
#define DRM_WRITE16(map, offset, val) \
        *(volatile uint16_t *)((uintptr_t)((map)->dev_addr) + (offset)) = (val)
#define DRM_WRITE32(map, offset, val) \
        *(volatile uint32_t *)((uintptr_t)((map)->dev_addr) + (offset)) = (val)

typedef struct drm_wait_queue {
        kcondvar_t      cv;
        kmutex_t        lock;
}wait_queue_head_t;

#define DRM_INIT_WAITQUEUE(q, pri)      \
{ \
        mutex_init(&(q)->lock, NULL, MUTEX_DRIVER, pri); \
        cv_init(&(q)->cv, NULL, CV_DRIVER, NULL);       \
}

#define DRM_FINI_WAITQUEUE(q)   \
{ \
        mutex_destroy(&(q)->lock);      \
        cv_destroy(&(q)->cv);   \
}

#define DRM_WAKEUP(q)   \
{ \
        mutex_enter(&(q)->lock); \
        cv_broadcast(&(q)->cv); \
        mutex_exit(&(q)->lock); \
}

#define jiffies ddi_get_lbolt()

#define DRM_WAIT_ON(ret, q, timeout, condition)                         \
        mutex_enter(&(q)->lock);                                        \
        while (!(condition)) {                                          \
                ret = cv_reltimedwait_sig(&(q)->cv, &(q)->lock, timeout,\
                    TR_CLOCK_TICK);                                     \
                if (ret == -1) {                                        \
                        ret = EBUSY;                                    \
                        break;                                          \
                } else if (ret == 0) {                                  \
                        ret = EINTR;                                    \
                        break;                                          \
                } else {                                                \
                        ret = 0;                                        \
                }                                                       \
        }                                                               \
        mutex_exit(&(q)->lock);

#define DRM_WAIT(ret, q, condition)  \
mutex_enter(&(q)->lock);        \
if (!(condition)) {     \
        ret = cv_timedwait_sig(&(q)->cv, &(q)->lock, jiffies + 30 * DRM_HZ); \
        if (ret == -1) {                                \
                /* gfx maybe hang */    \
                if (!(condition))       \
                        ret = -2;       \
        } else {        \
                ret = 0;        \
        }       \
} \
mutex_exit(&(q)->lock);


#define DRM_GETSAREA()                                                  \
{                                                                       \
        drm_local_map_t *map;                                           \
        DRM_SPINLOCK_ASSERT(&dev->dev_lock);                            \
        TAILQ_FOREACH(map, &dev->maplist, link) {                       \
                if (map->type == _DRM_SHM &&                            \
                        map->flags & _DRM_CONTAINS_LOCK) {              \
                        dev_priv->sarea = map;                          \
                        break;                                          \
                }                                                       \
        }                                                               \
}

#define LOCK_TEST_WITH_RETURN(dev, fpriv)                                       \
        if (!_DRM_LOCK_IS_HELD(dev->lock.hw_lock->lock) ||                      \
            dev->lock.filp != fpriv) {                                          \
                DRM_DEBUG("%s called without lock held", __func__);     \
                return (EINVAL);        \
        }

#define DRM_IRQ_ARGS    caddr_t arg
#define IRQ_HANDLED             DDI_INTR_CLAIMED
#define IRQ_NONE                DDI_INTR_UNCLAIMED

enum {
        DRM_IS_NOT_AGP,
        DRM_IS_AGP,
        DRM_MIGHT_BE_AGP
};

/* Capabilities taken from src/sys/dev/pci/pcireg.h. */
#ifndef PCIY_AGP
#define PCIY_AGP                0x02
#endif

#ifndef PCIY_EXPRESS
#define PCIY_EXPRESS            0x10
#endif

#define PAGE_ALIGN(addr)        (((addr) + DRM_PAGE_SIZE - 1) & DRM_PAGE_MASK)
#define DRM_SUSER(p)            (crgetsgid(p) == 0 || crgetsuid(p) == 0)

#define DRM_GEM_OBJIDR_HASHNODE 1024
#define idr_list_for_each(entry, head) \
        for (int key = 0; key < DRM_GEM_OBJIDR_HASHNODE; key++) \
                list_for_each(entry, &(head)->next[key])

/*
 * wait for 400 milliseconds
 */
#define DRM_HZ                  drv_usectohz(400000)

typedef unsigned long dma_addr_t;
typedef uint64_t        u64;
typedef uint32_t        u32;
typedef uint16_t        u16;
typedef uint8_t         u8;
typedef uint_t          irqreturn_t;
```

```
#define DRM_SUPPORT     1
#define DRM_UNSUPPORT   0

#define __OS_HAS_AGP    1

typedef struct drm_pci_id_list
{
        int vendor;
        int device;
        long driver_private;
        char *name;
} drm_pci_id_list_t;

#define DRM_AUTH        0x1
#define DRM_MASTER      0x2
#define DRM_ROOT_ONLY   0x4
typedef int drm_ioctl_t(DRM_IOCTL_ARGS);
typedef struct drm_ioctl_desc {
        int     (*func)(DRM_IOCTL_ARGS);
        int     flags;
} drm_ioctl_desc_t;

typedef struct drm_magic_entry {
        drm_magic_t             magic;
        struct drm_file         *priv;
        struct drm_magic_entry  *next;
} drm_magic_entry_t;

typedef struct drm_magic_head {
        struct drm_magic_entry *head;
        struct drm_magic_entry *tail;
} drm_magic_head_t;

typedef struct drm_buf {
        int             idx;            /* Index into master buflist */
        int             total;          /* Buffer size */
        int             order;          /* log-base-2(total) */
        int             used;           /* Amount of buffer in use (for DMA) */
        unsigned long   offset;         /* Byte offset (used internally) */
        void            *address;       /* Address of buffer */
        unsigned long   bus_address;    /* Bus address of buffer */
        struct drm_buf  *next;          /* Kernel-only: used for free list */
        volatile int    pending;        /* On hardware DMA queue */
        drm_file_t              *filp;
                                /* Uniq. identifier of holding process */
        int             context;        /* Kernel queue for this buffer */
        enum {
                DRM_LIST_NONE    = 0,
                DRM_LIST_FREE    = 1,
                DRM_LIST_WAIT    = 2,
                DRM_LIST_PEND    = 3,
                DRM_LIST_PRIO    = 4,
                DRM_LIST_RECLAIM = 5
        }               list;           /* Which list we're on */

        int             dev_priv_size;  /* Size of buffer private stoarge */
        void            *dev_private;   /* Per-buffer private storage */
} drm_buf_t;

typedef struct drm_freelist {
        int               initialized;  /* Freelist in use             */
        uint32_t          count;        /* Number of free buffers      */
        drm_buf_t         *next;        /* End pointer                 */

        int               low_mark;     /* Low water mark              */
        int               high_mark;    /* High water mark              */
} drm_freelist_t;

typedef struct drm_buf_entry {
        int               buf_size;
        int               buf_count;
        drm_buf_t         *buflist;
        int               seg_count;
        int               page_order;

        uint32_t          *seglist;
        unsigned long     *seglist_bus;

        drm_freelist_t    freelist;
} drm_buf_entry_t;

typedef TAILQ_HEAD(drm_file_list, drm_file) drm_file_list_t;

/* BEGIN CSTYLED */
typedef struct drm_local_map {
        unsigned long   offset;  /*  Physical address (0 for SAREA)     */
        unsigned long   size;    /* Physical size (bytes)              */
        drm_map_type_t  type;    /* Type of memory mapped              */
        drm_map_flags_t flags;   /* Flags                              */
        void            *handle; /* User-space: "Handle" to pass to mmap */
                                 /* Kernel-space: kernel-virtual address */
        int             mtrr;    /* Boolean: MTRR used                 */
                                 /* Private data                       */
        int             rid;     /* PCI resource ID for bus_space      */
        int             kernel_owned; /* Boolean: 1= initmapped, 0= addmapped */
        caddr_t         dev_addr;         /* base device address        */
        ddi_acc_handle_t  dev_handle;     /* The data access handle     */
        ddi_umem_cookie_t drm_umem_cookie; /* For SAREA alloc and free  */
        TAILQ_ENTRY(drm_local_map) link;
} drm_local_map_t;
/* END CSTYLED */

/*
 * This structure defines the drm_mm memory object, which will be used by the
 * DRM for its buffer objects.
 */
struct drm_gem_object {
        /* Reference count of this object */
        atomic_t refcount;

        /* Handle count of this object. Each handle also holds a reference */
        atomic_t handlecount;

        /* Related drm device */
        struct drm_device *dev;

        int flink;
        /*
         * Size of the object, in bytes.  Immutable over the object’s
         * lifetime.
         */
        size_t size;

        /*
         * Global name for this object, starts at 1. 0 means unnamed.
         * Access is covered by the object_name_lock in the related drm_device
         */
        int name;

        /*
         * Memory domains. These monitor which caches contain read/write data
```

```
         * related to the object. When transitioning from one set of domains
         * to another, the driver is called to ensure that caches are suitably
         * flushed and invalidated
         */
        uint32_t read_domains;
        uint32_t write_domain;

        /*
         * While validating an exec operation, the
         * new read/write domain values are computed here.
         * They will be transferred to the above values
         * at the point that any cache flushing occurs
         */
        uint32_t pending_read_domains;
        uint32_t pending_write_domain;

        void *driver_private;

        drm_local_map_t *map;
        ddi_dma_handle_t dma_hdl;
        ddi_acc_handle_t acc_hdl;
        caddr_t kaddr;
        size_t real_size;       /* real size of memory */
        pfn_t *pfnarray;
};

struct idr_list {
        struct idr_list *next, *prev;
        struct drm_gem_object *obj;
        uint32_t        handle;
        caddr_t contain_ptr;
};

struct drm_file {
        TAILQ_ENTRY(drm_file) link;
        int               authenticated;
        int               master;
        int               minor;
        pid_t             pid;
        uid_t             uid;
        int               refs;
        drm_magic_t       magic;
        unsigned long     ioctl_count;
        void             *driver_priv;
        /* Mapping of mm object handles to object pointers. */
        struct idr_list object_idr;
        /* Lock for synchronization of access to object_idr. */
        kmutex_t table_lock;

        dev_t dev;
        cred_t *credp;
};

typedef struct drm_lock_data {
        drm_hw_lock_t   *hw_lock;       /* Hardware lock                */
        drm_file_t      *filp;
        /* Uniq. identifier of holding process */
        kcondvar_t      lock_cv;        /* lock queue - SOLARIS Specific */
        kmutex_t        lock_mutex;     /* lock - SOLARIS Specific */
        unsigned long   lock_time;      /* Time of last lock in clock ticks */
} drm_lock_data_t;

/*
 * This structure, in drm_device_t, is always initialized while the device
 * is open.  dev->dma_lock protects the incrementing of dev->buf_use, which
 * when set marks that no further bufs may be allocated until device teardown
 * occurs (when the last open of the device has closed).  The high/low
 * watermarks of bufs are only touched by the X Server, and thus not
 * concurrently accessed, so no locking is needed.
 */
typedef struct drm_device_dma {
        drm_buf_entry_t bufs[DRM_MAX_ORDER+1];
        int             buf_count;
        drm_buf_t       **buflist;      /* Vector of pointers info bufs    */
        int             seg_count;
        int             page_count;
        unsigned long   *pagelist;
        unsigned long   byte_count;
        enum {
                _DRM_DMA_USE_AGP = 0x01,
                _DRM_DMA_USE_SG  = 0x02
        } flags;
} drm_device_dma_t;

typedef struct drm_agp_mem {
        void            *handle;
        unsigned long   bound; /* address */
        int             pages;
        caddr_t         phys_addr;
        struct drm_agp_mem *prev;
        struct drm_agp_mem *next;
} drm_agp_mem_t;

typedef struct drm_agp_head {
        agp_info_t      agp_info;
        const char      *chipset;
        drm_agp_mem_t   *memory;
        unsigned long   mode;
        int             enabled;
        int             acquired;
        unsigned long   base;
        int             mtrr;
        int             cant_use_aperture;
        unsigned long   page_mask;
        ldi_ident_t     agpgart_li;
        ldi_handle_t    agpgart_lh;
} drm_agp_head_t;


typedef struct drm_dma_handle {
        ddi_dma_handle_t        dma_hdl;
        ddi_acc_handle_t        acc_hdl;
        ddi_dma_cookie_t        cookie;
        uint_t          cookie_num;
        uintptr_t               vaddr;   /* virtual addr */
        uintptr_t               paddr;   /* physical addr */
        size_t          real_sz; /* real size of memory */
} drm_dma_handle_t;

typedef struct drm_sg_mem {
        unsigned long   handle;
        void            *virtual;
        int             pages;
        dma_addr_t      *busaddr;
        ddi_umem_cookie_t       *umem_cookie;
        drm_dma_handle_t        *dmah_sg;
        drm_dma_handle_t        *dmah_gart; /* Handle to PCI memory */
} drm_sg_mem_t;

/*
 * Generic memory manager structs
 */
```

```
struct drm_mm_node {
        struct list_head fl_entry;
        struct list_head ml_entry;
        int free;
        unsigned long start;
        unsigned long size;
        struct drm_mm *mm;
        void *private;
};

struct drm_mm {
        struct list_head fl_entry;
        struct list_head ml_entry;
};

typedef TAILQ_HEAD(drm_map_list, drm_local_map) drm_map_list_t;

typedef TAILQ_HEAD(drm_vbl_sig_list, drm_vbl_sig) drm_vbl_sig_list_t;
typedef struct drm_vbl_sig {
        TAILQ_ENTRY(drm_vbl_sig) link;
        unsigned int    sequence;
        int             signo;
        int             pid;
} drm_vbl_sig_t;


/* used for clone device */
typedef TAILQ_HEAD(drm_cminor_list, drm_cminor) drm_cminor_list_t;
typedef struct drm_cminor {
        TAILQ_ENTRY(drm_cminor) link;
        drm_file_t              *fpriv;
        int                     minor;
} drm_cminor_t;

/* location of GART table */
#define DRM_ATI_GART_MAIN       1
#define DRM_ATI_GART_FB         2

typedef struct ati_pcigart_info {
        int gart_table_location;
        int is_pcie;
        void *addr;
        dma_addr_t bus_addr;
        drm_local_map_t mapping;
} drm_ati_pcigart_info;

/* DRM device structure */
struct drm_device;
struct drm_driver_info {
        int (*load)(struct drm_device *, unsigned long);
        int (*firstopen)(struct drm_device *);
        int (*open)(struct drm_device *, drm_file_t *);
        void (*preclose)(struct drm_device *, drm_file_t *);
        void (*postclose)(struct drm_device *, drm_file_t *);
        void (*lastclose)(struct drm_device *);
        int (*unload)(struct drm_device *);
        void (*reclaim_buffers_locked)(struct drm_device *, drm_file_t *);
        int (*presetup)(struct drm_device *);
        int (*postsetup)(struct drm_device *);
        int (*open_helper)(struct drm_device *, drm_file_t *);
        void (*free_filp_priv)(struct drm_device *, drm_file_t *);
        void (*release)(struct drm_device *, void *);
        int (*dma_ioctl)(DRM_IOCTL_ARGS);
        void (*dma_ready)(struct drm_device *);
        int (*dma_quiescent)(struct drm_device *);
        int (*dma_flush_block_and_flush)(struct drm_device *,
                        int, drm_lock_flags_t);
        int (*dma_flush_unblock)(struct drm_device *, int,
                                drm_lock_flags_t);
        int (*context_ctor)(struct drm_device *, int);
        int (*context_dtor)(struct drm_device *, int);
        int (*kernel_context_switch)(struct drm_device *, int, int);
        int (*kernel_context_switch_unlock)(struct drm_device *);
        int (*device_is_agp) (struct drm_device *);
        int (*irq_preinstall)(struct drm_device *);
        void (*irq_postinstall)(struct drm_device *);
        void (*irq_uninstall)(struct drm_device *dev);
        uint_t (*irq_handler)(DRM_IRQ_ARGS);
        int (*vblank_wait)(struct drm_device *, unsigned int *);
        int (*vblank_wait2)(struct drm_device *, unsigned int *);
        /* added for intel minimized vblank */
        u32 (*get_vblank_counter)(struct drm_device *dev, int crtc);
        int (*enable_vblank)(struct drm_device *dev, int crtc);
        void (*disable_vblank)(struct drm_device *dev, int crtc);

        /*
         * Driver-specific constructor for drm_gem_objects, to set up
         * obj->driver_private.
         *
         * Returns 0 on success.
         */
        int (*gem_init_object) (struct drm_gem_object *obj);
        void (*gem_free_object) (struct drm_gem_object *obj);


        drm_ioctl_desc_t *driver_ioctls;
        int     max_driver_ioctl;

        int     buf_priv_size;
        int     driver_major;
        int     driver_minor;
        int     driver_patchlevel;
        const char *driver_name;        /* Simple driver name              */
        const char *driver_desc;        /* Longer driver name              */
        const char *driver_date;        /* Date of last major changes.     */

        unsigned use_agp :1;
        unsigned require_agp :1;
        unsigned use_sg :1;
        unsigned use_dma :1;
        unsigned use_pci_dma :1;
        unsigned use_dma_queue :1;
        unsigned use_irq :1;
        unsigned use_vbl_irq :1;
        unsigned use_vbl_irq2 :1;
        unsigned use_mtrr :1;
        unsigned use_gem;
};

/*
 * hardware-specific code needs to initialize mutexes which
 * can be used in interrupt context, so they need to know
 * the interrupt priority. Interrupt cookie in drm_device
 * structure is the intr_block field.
 */
#define DRM_INTR_PRI(dev) \
        DDI_INTR_PRI((dev)->intr_block)

struct drm_device {
        drm_driver_t    *driver;
        drm_cminor_list_t       minordevs;
```

```
        dev_info_t *dip;
        void    *drm_handle;
        int drm_supported;
        const char *desc; /* current driver description */
        kmutex_t *irq_mutex;
        kcondvar_t *irq_cv;

        ddi_iblock_cookie_t intr_block;
        uint32_t        pci_device;     /* PCI device id */
        uint32_t        pci_vendor;
        char            *unique;        /* Unique identifier: e.g., busid  */
        int             unique_len;     /* Length of unique field          */
        int             if_version;     /* Highest interface version set */
        int             flags;  /* Flags to open(2)                */

        /* Locks */
        kmutex_t        vbl_lock;       /* protects vblank operations */
        kmutex_t        dma_lock;       /* protects dev->dma */
        kmutex_t        irq_lock;       /* protects irq condition checks */
        kmutex_t        dev_lock;       /* protects everything else */
        drm_lock_data_t   lock;         /* Information on hardware lock    */
        kmutex_t        struct_mutex;   /* < For others */

        /* Usage Counters */
        int               open_count;   /* Outstanding files open          */
        int               buf_use;      /* Buffers in use -- cannot alloc  */

        /* Performance counters */
        unsigned long     counters;
        drm_stat_type_t   types[15];
        uint32_t          counts[15];

        /* Authentication */
        drm_file_list_t   files;
        drm_magic_head_t  magiclist[DRM_HASH_SIZE];

        /* Linked list of mappable regions. Protected by dev_lock */
        drm_map_list_t    maplist;

        drm_local_map_t   **context_sareas;
        int               max_context;

        /* DMA queues (contexts) */
        drm_device_dma_t  *dma;         /* Optional pointer for DMA support */

        /* Context support */
        int               irq;          /* Interrupt used by board         */
        int               irq_enabled;  /* True if the irq handler is enabled */
        int               pci_domain;
        int               pci_bus;
        int               pci_slot;
        int               pci_func;
        atomic_t          context_flag; /* Context swapping flag           */
        int               last_context; /* Last current context            */

        /* Only used for Radeon */
        atomic_t        vbl_received;
        atomic_t        vbl_received2;

        drm_vbl_sig_list_t vbl_sig_list;
        drm_vbl_sig_list_t vbl_sig_list2;
        /*
         * At load time, disabling the vblank interrupt won't be allowed since
         * old clients may not call the modeset ioctl and therefore misbehave.
         * Once the modeset ioctl *has* been called though, we can safely
         * disable them when unused.
         */
        int vblank_disable_allowed;

        wait_queue_head_t       vbl_queue;      /* vbl wait channel */
        /* vbl wait channel array */
        wait_queue_head_t       *vbl_queues;

        /* number of VBLANK interrupts */
        /* (driver must alloc the right number of counters) */
        atomic_t          *_vblank_count;
        /* signal list to send on VBLANK */
        struct drm_vbl_sig_list *vbl_sigs;

        /* number of signals pending on all crtcs */
        atomic_t          vbl_signal_pending;
        /* number of users of vblank interrupts per crtc */
        atomic_t          *vblank_refcount;
        /* protected by dev->vbl_lock, used for wraparound handling */
        u32               *last_vblank;
        /* so we don’t call enable more than */
        atomic_t          *vblank_enabled;
        /* Display driver is setting mode */
        int             *vblank_inmodeset;
        /* Don’t wait while crtc is likely disabled */
        int             *vblank_suspend;
        /* size of vblank counter register */
        u32             max_vblank_count;
        int             num_crtcs;
        kmutex_t        tasklet_lock;
        void (*locked_tasklet_func)(struct drm_device *dev);

        pid_t             buf_pgid;
        drm_agp_head_t    *agp;
        drm_sg_mem_t      *sg;  /* Scatter gather memory */
        uint32_t          *ctx_bitmap;
        void              *dev_private;
        unsigned int      agp_buffer_token;
        drm_local_map_t   *agp_buffer_map;

        kstat_t           *asoft_ksp; /* kstat support */

        /* name Drawable information */
        kmutex_t        drw_lock;
        unsigned int drw_bitfield_length;
        u32 *drw_bitfield;
        unsigned int drw_info_length;
        drm_drawable_info_t **drw_info;

        /* \name GEM information */
        /* @{ */
        kmutex_t object_name_lock;
        struct idr_list object_name_idr;
        atomic_t object_count;
        atomic_t object_memory;
        atomic_t pin_count;
        atomic_t pin_memory;
        atomic_t gtt_count;
        atomic_t gtt_memory;
        uint32_t gtt_total;
        uint32_t invalidate_domains;    /* domains pending invalidation */
        uint32_t flush_domains; /* domains pending flush */
        /* @} */

        /*
         * Saving S3 context
         */
```

```
        void              *s3_private;
};

/* Memory management support (drm_memory.c) */
void    drm_mem_init(void);
void    drm_mem_uninit(void);
void    *drm_alloc(size_t, int);
void    *drm_calloc(size_t, size_t, int);
void    *drm_realloc(void *, size_t, size_t, int);
void    drm_free(void *, size_t, int);
int     drm_ioremap(drm_device_t *, drm_local_map_t *);
void    drm_ioremapfree(drm_local_map_t *);

void drm_core_ioremap(struct drm_local_map *, struct drm_device *);
void drm_core_ioremapfree(struct drm_local_map *, struct drm_device *);

void drm_pci_free(drm_device_t *, drm_dma_handle_t *);
void *drm_pci_alloc(drm_device_t *, size_t, size_t, dma_addr_t, int);

struct drm_local_map *drm_core_findmap(struct drm_device *, unsigned long);

int     drm_context_switch(drm_device_t *, int, int);
int     drm_context_switch_complete(drm_device_t *, int);
int     drm_ctxbitmap_init(drm_device_t *);
void    drm_ctxbitmap_cleanup(drm_device_t *);
void    drm_ctxbitmap_free(drm_device_t *, int);
int     drm_ctxbitmap_next(drm_device_t *);

/* Locking IOCTL support (drm_lock.c) */
int     drm_lock_take(drm_lock_data_t *, unsigned int);
int     drm_lock_transfer(drm_device_t *,
                        drm_lock_data_t *, unsigned int);
int     drm_lock_free(drm_device_t *,
                    volatile unsigned int *, unsigned int);

/* Buffer management support (drm_bufs.c) */
unsigned long drm_get_resource_start(drm_device_t *, unsigned int);
unsigned long drm_get_resource_len(drm_device_t *, unsigned int);
int     drm_initmap(drm_device_t *, unsigned long, unsigned long,
    unsigned int, int, int);
void    drm_rmmap(drm_device_t *, drm_local_map_t *);
int     drm_addmap(drm_device_t *, unsigned long, unsigned long,
    drm_map_type_t, drm_map_flags_t, drm_local_map_t **);
int     drm_order(unsigned long);

/* DMA support (drm_dma.c) */
int     drm_dma_setup(drm_device_t *);
void    drm_dma_takedown(drm_device_t *);
void    drm_free_buffer(drm_device_t *, drm_buf_t *);
void    drm_reclaim_buffers(drm_device_t *, drm_file_t *);
#define drm_core_reclaim_buffers        drm_reclaim_buffers

/* IRQ support (drm_irq.c) */
int     drm_irq_install(drm_device_t *);
int     drm_irq_uninstall(drm_device_t *);
uint_t  drm_irq_handler(DRM_IRQ_ARGS);
void    drm_driver_irq_preinstall(drm_device_t *);
void    drm_driver_irq_postinstall(drm_device_t *);
void    drm_driver_irq_uninstall(drm_device_t *);
int     drm_vblank_wait(drm_device_t *, unsigned int *);
void    drm_vbl_send_signals(drm_device_t *);
void    drm_handle_vblank(struct drm_device *dev, int crtc);
u32     drm_vblank_count(struct drm_device *dev, int crtc);
int     drm_vblank_get(struct drm_device *dev, int crtc);
void    drm_vblank_put(struct drm_device *dev, int crtc);
int     drm_vblank_init(struct drm_device *dev, int num_crtcs);
```

```
void    drm_vblank_cleanup(struct drm_device *dev);
int    drm_modeset_ctl(DRM_IOCTL_ARGS);

/* AGP/GART support (drm_agpsupport.c) */
int     drm_device_is_agp(drm_device_t *);
int     drm_device_is_pcie(drm_device_t *);
drm_agp_head_t *drm_agp_init(drm_device_t *);
void    drm_agp_fini(drm_device_t *);
int     drm_agp_do_release(drm_device_t *);
void    *drm_agp_allocate_memory(size_t pages,
            uint32_t type, drm_device_t *dev);
int     drm_agp_free_memory(agp_allocate_t *handle, drm_device_t *dev);
int     drm_agp_bind_memory(unsigned int, uint32_t, drm_device_t *);
int     drm_agp_unbind_memory(unsigned long, drm_device_t *);
int     drm_agp_bind_pages(drm_device_t *dev,
                    pfn_t *pages,
                    unsigned long num_pages,
                    uint32_t gtt_offset);
int     drm_agp_unbind_pages(drm_device_t *dev,
                    unsigned long num_pages,
                    uint32_t gtt_offset,
                    uint32_t type);
void drm_agp_chipset_flush(struct drm_device *dev);
void drm_agp_rebind(struct drm_device *dev);

/* kstat support (drm_kstats.c) */
int     drm_init_kstats(drm_device_t *);
void    drm_fini_kstats(drm_device_t *);

/* Scatter Gather Support (drm_scatter.c) */
void    drm_sg_cleanup(drm_device_t *, drm_sg_mem_t *);

/* ATI PCIGART support (ati_pcigart.c) */
int     drm_ati_pcigart_init(drm_device_t *, drm_ati_pcigart_info *);
int     drm_ati_pcigart_cleanup(drm_device_t *, drm_ati_pcigart_info *);

/* Locking IOCTL support (drm_drv.c) */
int     drm_lock(DRM_IOCTL_ARGS);
int     drm_unlock(DRM_IOCTL_ARGS);
int     drm_version(DRM_IOCTL_ARGS);
int     drm_setversion(DRM_IOCTL_ARGS);
/* Cache management (drm_cache.c) */
void drm_clflush_pages(caddr_t *pages, unsigned long num_pages);

/* Misc. IOCTL support (drm_ioctl.c) */
int     drm_irq_by_busid(DRM_IOCTL_ARGS);
int     drm_getunique(DRM_IOCTL_ARGS);
int     drm_setunique(DRM_IOCTL_ARGS);
int     drm_getmap(DRM_IOCTL_ARGS);
int     drm_getclient(DRM_IOCTL_ARGS);
int     drm_getstats(DRM_IOCTL_ARGS);
int     drm_noop(DRM_IOCTL_ARGS);

/* Context IOCTL support (drm_context.c) */
int     drm_resctx(DRM_IOCTL_ARGS);
int     drm_addctx(DRM_IOCTL_ARGS);
int     drm_modctx(DRM_IOCTL_ARGS);
int     drm_getctx(DRM_IOCTL_ARGS);
int     drm_switchctx(DRM_IOCTL_ARGS);
int     drm_newctx(DRM_IOCTL_ARGS);
int     drm_rmctx(DRM_IOCTL_ARGS);
int     drm_setsareactx(DRM_IOCTL_ARGS);
int     drm_getsareactx(DRM_IOCTL_ARGS);

/* Drawable IOCTL support (drm_drawable.c) */
int     drm_adddraw(DRM_IOCTL_ARGS);
```

```
int     drm_rmdraw(DRM_IOCTL_ARGS);
int     drm_update_draw(DRM_IOCTL_ARGS);

/* Authentication IOCTL support (drm_auth.c) */
int     drm_getmagic(DRM_IOCTL_ARGS);
int     drm_authmagic(DRM_IOCTL_ARGS);
int     drm_remove_magic(drm_device_t *, drm_magic_t);
drm_file_t      *drm_find_file(drm_device_t *, drm_magic_t);
/* Buffer management support (drm_bufs.c) */
int     drm_addmap_ioctl(DRM_IOCTL_ARGS);
int     drm_rmmap_ioctl(DRM_IOCTL_ARGS);
int     drm_addbufs_ioctl(DRM_IOCTL_ARGS);
int     drm_infobufs(DRM_IOCTL_ARGS);
int     drm_markbufs(DRM_IOCTL_ARGS);
int     drm_freebufs(DRM_IOCTL_ARGS);
int     drm_mapbufs(DRM_IOCTL_ARGS);

/* DMA support (drm_dma.c) */
int     drm_dma(DRM_IOCTL_ARGS);

/* IRQ support (drm_irq.c) */
int     drm_control(DRM_IOCTL_ARGS);
int     drm_wait_vblank(DRM_IOCTL_ARGS);

/* AGP/GART support (drm_agpsupport.c) */
int     drm_agp_acquire(DRM_IOCTL_ARGS);
int     drm_agp_release(DRM_IOCTL_ARGS);
int     drm_agp_enable(DRM_IOCTL_ARGS);
int     drm_agp_info(DRM_IOCTL_ARGS);
int     drm_agp_alloc(DRM_IOCTL_ARGS);
int     drm_agp_free(DRM_IOCTL_ARGS);
int     drm_agp_unbind(DRM_IOCTL_ARGS);
int     drm_agp_bind(DRM_IOCTL_ARGS);

/* Scatter Gather Support (drm_scatter.c) */
int     drm_sg_alloc(DRM_IOCTL_ARGS);
int     drm_sg_free(DRM_IOCTL_ARGS);

/*      drm_mm.c        */
struct drm_mm_node *drm_mm_get_block(struct drm_mm_node *parent,
                                    unsigned long size, unsigned alignment);
struct drm_mm_node *drm_mm_search_free(const struct drm_mm *mm,
                                    unsigned long size,
                                    unsigned alignment, int best_match);

extern void drm_mm_clean_ml(const struct drm_mm *mm);
extern int drm_debug_flag;

/* We add function to support DRM_DEBUG,DRM_ERROR,DRM_INFO */
extern void drm_debug(const char *fmt, ...);
extern void drm_error(const char *fmt, ...);
extern void drm_info(const char *fmt, ...);

#ifdef DEBUG
#define DRM_DEBUG               if (drm_debug_flag >= 2) drm_debug
#define DRM_INFO                if (drm_debug_flag >= 1) drm_info
#else
#define DRM_DEBUG(...)
#define DRM_INFO(...)
#endif

#define DRM_ERROR               drm_error


#define MAX_INSTNUMS 16
```

```
extern int drm_dev_to_instance(dev_t);
extern int drm_dev_to_minor(dev_t);
extern void *drm_supp_register(dev_info_t *, drm_device_t *);
extern int drm_supp_unregister(void *);

extern int drm_open(drm_device_t *, drm_cminor_t *, int, int, cred_t *);
extern int drm_close(drm_device_t *, int, int, int, cred_t *);
extern int drm_attach(drm_device_t *);
extern int drm_detach(drm_device_t *);
extern int drm_probe(drm_device_t *, drm_pci_id_list_t *);

extern int drm_pci_init(drm_device_t *);
extern void drm_pci_end(drm_device_t *);
extern int pci_get_info(drm_device_t *, int *, int *, int *);
extern int pci_get_irq(drm_device_t *);
extern int pci_get_vendor(drm_device_t *);
extern int pci_get_device(drm_device_t *);

extern struct drm_drawable_info *drm_get_drawable_info(drm_device_t *,
                                                        drm_drawable_t);
/* File Operations helpers (drm_fops.c) */
extern drm_file_t *drm_find_file_by_proc(drm_device_t *, cred_t *);
extern drm_cminor_t *drm_find_file_by_minor(drm_device_t *, int);
extern int drm_open_helper(drm_device_t *, drm_cminor_t *, int, int,
    cred_t *);

/* Graphics Execution Manager library functions (drm_gem.c) */
int drm_gem_init(struct drm_device *dev);
void drm_gem_object_free(struct drm_gem_object *obj);
struct drm_gem_object *drm_gem_object_alloc(struct drm_device *dev,
                                            size_t size);
void drm_gem_object_handle_free(struct drm_gem_object *obj);

void drm_gem_object_reference(struct drm_gem_object *obj);
void drm_gem_object_unreference(struct drm_gem_object *obj);

int drm_gem_handle_create(struct drm_file *file_priv,
                            struct drm_gem_object *obj,
                            int *handlep);
void drm_gem_object_handle_reference(struct drm_gem_object *obj);

void drm_gem_object_handle_unreference(struct drm_gem_object *obj);

struct drm_gem_object *drm_gem_object_lookup(struct drm_file *filp,
                                            int handle);
int drm_gem_close_ioctl(DRM_IOCTL_ARGS);
int drm_gem_flink_ioctl(DRM_IOCTL_ARGS);
int drm_gem_open_ioctl(DRM_IOCTL_ARGS);
void drm_gem_open(struct drm_file *file_private);
void drm_gem_release(struct drm_device *dev, struct drm_file *file_private);


#endif  /* _DRMP_H */
```

```
**********************************************************
  127992 Thu Feb 25 15:39:40 2016
new/usr/src/uts/common/io/sfe/sfe_util.c
2976 remove useless offsetof() macros
**********************************************************
/*
 * sfe_util.c: general ethernet mac driver framework version 2.6
 *
 * Copyright (c) 2002-2008 Masayuki Murayama.  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 author nor the names of its contributors may be
 *    used to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "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
 * COPYRIGHT OWNER OR CONTRIBUTORS 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.
 */

/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * System Header files.
 */
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/debug.h>
#include <sys/kmem.h>
#include <sys/vtrace.h>
#include <sys/ethernet.h>
#include <sys/modctl.h>
#include <sys/errno.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/stream.h>         /* required for MBLK* */
#include <sys/strsun.h>         /* required for mionack() */
#include <sys/byteorder.h>
#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/pci.h>
#include <inet/common.h>
#include <inet/led.h>
#include <inet/mi.h>
#include <inet/nd.h>
```

```
#include <sys/crc32.h>

#include <sys/note.h>

#include "sfe_mii.h"
#include "sfe_util.h"



extern char ident[];

/* Debugging support */
#ifdef GEM_DEBUG_LEVEL
static int gem_debug = GEM_DEBUG_LEVEL;
#define DPRINTF(n, args)        if (gem_debug > (n)) cmn_err args
#else
#define DPRINTF(n, args)
#undef ASSERT
#define ASSERT(x)
#endif

#define IOC_LINESIZE    0x40    /* Is it right for amd64? */

/*
 * Useful macros and typedefs
 */
#define ROUNDUP(x, a)   (((x) + (a) - 1) & ~((a) - 1))

#define GET_NET16(p)    ((((uint8_t *)(p))[0] << 8)| ((uint8_t *)(p))[1])
#define GET_ETHERTYPE(p)        GET_NET16(((uint8_t *)(p)) + ETHERADDRL*2)

#define GET_IPTYPEv4(p) (((uint8_t *)(p))[sizeof (struct ether_header) + 9])
#define GET_IPTYPEv6(p) (((uint8_t *)(p))[sizeof (struct ether_header) + 6])


#ifndef INT32_MAX
#define INT32_MAX       0x7fffffff
#endif

#define VTAG_OFF        (ETHERADDRL*2)
#ifndef VTAG_SIZE
#define VTAG_SIZE       4
#endif
#ifndef VTAG_TPID
#define VTAG_TPID       0x8100U
#endif

#define GET_TXBUF(dp, sn)       \
        &(dp)->tx_buf[SLOT((dp)->tx_slots_base + (sn), (dp)->gc.gc_tx_buf_size)]

#ifndef offsetof
#define offsetof(t, m)  ((long)&(((t *) 0)->m))
#endif
#define TXFLAG_VTAG(flag)       \
        (((flag) & GEM_TXFLAG_VTAG) >> GEM_TXFLAG_VTAG_SHIFT)

#define MAXPKTBUF(dp)   \
        ((dp)->mtu + sizeof (struct ether_header) + VTAG_SIZE + ETHERFCSL)

#define WATCH_INTERVAL_FAST     drv_usectohz(100*1000)  /* 100mS */
#define BOOLEAN(x)      ((x) != 0)

/*
 * Macros to distinct chip generation.
 */
```

```
/*
 * Private functions
 */
static void gem_mii_start(struct gem_dev *);
static void gem_mii_stop(struct gem_dev *);

/* local buffer management */
static void gem_nd_setup(struct gem_dev *dp);
static void gem_nd_cleanup(struct gem_dev *dp);
static int gem_alloc_memory(struct gem_dev *);
static void gem_free_memory(struct gem_dev *);
static void gem_init_rx_ring(struct gem_dev *);
static void gem_init_tx_ring(struct gem_dev *);
__INLINE__ static void gem_append_rxbuf(struct gem_dev *, struct rxbuf *);

static void gem_tx_timeout(struct gem_dev *);
static void gem_mii_link_watcher(struct gem_dev *dp);
static int gem_mac_init(struct gem_dev *dp);
static int gem_mac_start(struct gem_dev *dp);
static int gem_mac_stop(struct gem_dev *dp, uint_t flags);
static void gem_mac_ioctl(struct gem_dev *dp, queue_t *wq, mblk_t *mp);

static  struct ether_addr       gem_etherbroadcastaddr = {
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff
};
```
______***unchanged_portion_omitted_***

```
**********************************************************
   33744 Thu Feb 25 15:39:41 2016
new/usr/src/uts/common/io/vscan/vscan_svc.c
2976 remove useless offsetof() macros
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/stat.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/time.h>
#include <sys/varargs.h>
#include <sys/conf.h>
#include <sys/modctl.h>
#include <sys/cmn_err.h>
#include <sys/vnode.h>
#include <fs/fs_subr.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/disp.h>
#include <sys/sdt.h>
#include <sys/cred.h>
#include <sys/list.h>
#include <sys/vscan.h>
#include <sys/sysmacros.h>
#endif /* ! codereview */

#define VS_REQ_MAGIC            0x52515354 /* 'RQST' */

#define VS_REQS_DEFAULT         20000   /* pending scan requests - reql */
#define VS_NODES_DEFAULT        128     /* concurrent file scans */
#define VS_WORKERS_DEFAULT      32      /* worker threads */
#define VS_SCANWAIT_DEFAULT     15*60   /* seconds to wait for scan result */
#define VS_REQL_HANDLER_TIMEOUT 30
#define VS_EXT_RECURSE_DEPTH    8

/* access derived from scan result (VS_STATUS_XXX) and file attributes */
#define VS_ACCESS_UNDEFINED     0
#define VS_ACCESS_ALLOW         1       /* return 0 */
#define VS_ACCESS_DENY          2       /* return EACCES */

#define tolower(C)      (((C) >= 'A' && (C) <= 'Z') ? (C) - 'A' + 'a' : (C))
```

```
#define offsetof(s, m)  (size_t)(&(((s *)0)->m))

/* global variables - tunable via /etc/system */
uint32_t vs_reqs_max = VS_REQS_DEFAULT; /* max scan requests */
uint32_t vs_nodes_max = VS_NODES_DEFAULT; /* max in-progress scan requests */
uint32_t vs_workers = VS_WORKERS_DEFAULT; /* max workers send reqs to vscand */
uint32_t vs_scan_wait = VS_SCANWAIT_DEFAULT; /* secs to wait for scan result */


/*
 * vscan_svc_state
 *
 *   +-----------------+
 *   | VS_SVC_UNCONFIG |
 *   +-----------------+
 *      |           ^
 *      | svc_init  | svc_fini
 *      v           |
 *   +-----------------+
 *   | VS_SVC_IDLE     |<----|
 *   +-----------------+         |
 *      |                   |
 *      | svc_enable        |
 *      |<----------------|  |
 *      v                |  |
 *   +-----------------+  |  |
 *   | VS_SVC_ENABLED  |--|  |
 *   +-----------------+     |
 *      |                   |
 *      | svc_disable       | handler thread exit,
 *      v                   | all requests complete
 *   +-----------------+         |
 *   | VS_SVC_DISABLED |-----|
 *   +-----------------+
 *
 * svc_enable may occur when we are already in the ENABLED
 * state if vscand has exited without clean shutdown and
 * then reconnected within the delayed disable time period
 * (vs_reconnect_timeout) - see vscan_drv
 */

typedef enum {
        VS_SVC_UNCONFIG,
        VS_SVC_IDLE,
        VS_SVC_ENABLED, /* service enabled and registered */
        VS_SVC_DISABLED /* service disabled and nunregistered */
} vscan_svc_state_t;
______unchanged_portion_omitted_
```

```
**********************************************************
   17123 Thu Feb 25 15:39:41 2016
new/usr/src/uts/common/sys/ecppvar.h
2976 remove useless offsetof() macros
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _SYS_ECPPVAR_H
#define _SYS_ECPPVAR_H

#include <sys/note.h>
#include <sys/sysmacros.h>
#endif /* ! codereview */

#ifdef  __cplusplus
extern "C" {
#endif

struct ecppunit;

/*
 * Hardware-abstraction structure
 */
struct ecpp_hw {
        int     (*map_regs)(struct ecppunit *);         /* map registers */
        void    (*unmap_regs)(struct ecppunit *);       /* unmap registers */
        int     (*config_chip)(struct ecppunit *);      /* configure SuperIO */
        void    (*config_mode)(struct ecppunit *);      /* config new mode */
        void    (*mask_intr)(struct ecppunit *);        /* mask interrupts */
        void    (*unmask_intr)(struct ecppunit *);      /* unmask interrupts */
        int     (*dma_start)(struct ecppunit *);        /* start DMA transfer */
        int     (*dma_stop)(struct ecppunit *, size_t *); /* stop DMA xfer */
        size_t  (*dma_getcnt)(struct ecppunit *);       /* get DMA counter */
        ddi_dma_attr_t  *attr;                          /* DMA attributes */
};

#define ECPP_MAP_REGS(pp)               (pp)->hw->map_regs(pp)
#define ECPP_UNMAP_REGS(pp)             (pp)->hw->unmap_regs(pp)
#define ECPP_CONFIG_CHIP(pp)            (pp)->hw->config_chip(pp)
#define ECPP_CONFIG_MODE(pp)            (pp)->hw->config_mode(pp)
#define ECPP_MASK_INTR(pp)              (pp)->hw->mask_intr(pp)
#define ECPP_UNMASK_INTR(pp)            (pp)->hw->unmask_intr(pp)
```

```
#define ECPP_DMA_START(pp)              (pp)->hw->dma_start(pp)
#define ECPP_DMA_STOP(pp, cnt)          (pp)->hw->dma_stop(pp, cnt)
#define ECPP_DMA_GETCNT(pp)             (pp)->hw->dma_getcnt(pp)

/* NSC 87332/97317 and EBus DMAC */
struct ecpp_ebus {
        struct config_reg       *c_reg;         /* configuration registers */
        ddi_acc_handle_t        c_handle;       /* handle for conf regs */
        struct cheerio_dma_reg  *dmac;          /* ebus dmac registers */
        ddi_acc_handle_t        d_handle;       /* handle for dmac registers */
        struct config2_reg      *c2_reg;        /* 97317 2nd level conf regs */
        ddi_acc_handle_t        c2_handle;      /* handle for c2_reg */
};

/* Southbridge SuperIO and 8237 DMAC */
struct ecpp_m1553 {
        struct isaspace         *isa_space;     /* all of isa space */
        ddi_acc_handle_t        d_handle;       /* handle for isa space */
        uint8_t                 chn;            /* 8237 dma channel */
        int                     isadma_entered; /* Southbridge DMA workaround */
};

#if defined(__x86)
struct ecpp_x86 {
        uint8_t                 chn;
};
#endif

/*
 * Hardware binding structure
 */
struct ecpp_hw_bind {
        char            *name;          /* binding name */
        struct ecpp_hw  *hw;            /* hw description */
        char            *info;          /* info string */
};

/* ecpp e_busy states */
typedef enum {
        ECPP_IDLE = 1,  /* No ongoing transfers */
        ECPP_BUSY = 2,  /* Ongoing transfers on the cable */
        ECPP_DATA = 3,  /* Not used */
        ECPP_ERR = 4,   /* Bad status in Centronics mode */
        ECPP_FLUSH = 5  /* Currently flushing the q */
} ecpp_busy_t;

/*
 * ecpp soft state structure
 */
struct ecppunit {
        kmutex_t        umutex;         /* lock for this structure */
        int             instance;       /* instance number */
        dev_info_t      *dip;           /* device information */
        ddi_iblock_cookie_t ecpp_trap_cookie;   /* interrupt cookie */
        ecpp_busy_t     e_busy;         /* ecpp busy flag */
        kcondvar_t      pport_cv;       /* cv to signal idle state */
        /*
         * common SuperIO registers
         */
        struct info_reg         *i_reg;         /* info registers */
        struct fifo_reg         *f_reg;         /* fifo register */
        ddi_acc_handle_t        i_handle;
        ddi_acc_handle_t        f_handle;
        /*
         * DMA support
         */
        ddi_dma_handle_t        dma_handle;     /* DMA handle */
        ddi_dma_cookie_t        dma_cookie;     /* current cookie */
        uint_t                  dma_cookie_count;       /* # of cookies */
        uint_t                  dma_nwin;       /* # of DMA windows */
        uint_t                  dma_curwin;     /* current window number */
        uint_t                  dma_dir;        /* transfer direction */
        /*
         * hardware-dependent stuff
         */
        struct ecpp_hw  *hw;            /* operations/attributes */
        union {                         /* hw-dependent data */
                struct ecpp_ebus        ebus;
                struct ecpp_m1553       m1553;
#if defined(__x86)
                struct ecpp_x86         x86;
#endif
        } uh;
        /*
         * DDI/STREAMS stuff
         */
        boolean_t       oflag;          /* instance open flag */
        queue_t         *readq;         /* pointer to readq */
        queue_t         *writeq;        /* pointer to writeq */
        mblk_t          *msg;           /* current message block */
        boolean_t       suspended;      /* driver suspended status */
        /*
         * Modes of operation
         */
        int             current_mode;   /* 1284 mode */
        uchar_t         current_phase;  /* 1284 phase */
        uchar_t         backchannel;    /* backchannel mode supported */
        uchar_t         io_mode;        /* transfer mode: PIO/DMA */
        /*
         * Ioctls support
         */
        struct ecpp_transfer_parms xfer_parms;  /* transfer parameters */
        struct ecpp_regs regs;          /* control/status registers */
        uint8_t         saved_dsr;      /* store the dsr returned from TESTIO */
        boolean_t       timeout_error;  /* store the timeout for GETERR */
        uchar_t         port;           /* xfer type: dma/pio/tfifo */
        struct prn_timeouts prn_timeouts; /* prnio timeouts */
        /*
         * ecpp.conf parameters
         */
        uchar_t         init_seq;       /* centronics init seq */
        uint32_t        wsrv_retry;     /* delay (ms) before next wsrv */
        uint32_t        wait_for_busy;  /* wait for BUSY to deassert */
        uint32_t        data_setup_time; /* pio centronics handshake */
        uint32_t        strobe_pulse_width; /* pio centronics handshake */
        uint8_t         fast_centronics; /* DMA/PIO centronics */
        uint8_t         fast_compat;    /* DMA/PIO 1284 compatible mode */
        uint32_t        ecp_rev_speed;  /* rev xfer speed in ECP, bytes/sec */
        uint32_t        rev_watchdog;   /* rev xfer watchdog period, ms */
        /*
         * Timeouts
         */
        timeout_id_t    timeout_id;     /* io transfers timer */
        timeout_id_t    fifo_timer_id;  /* drain SuperIO FIFO */
        timeout_id_t    wsrv_timer_id;  /* wsrv timeout */
        /*
         * Softintr data
         */
        ddi_softintr_t  softintr_id;
        int             softintr_flags; /* flags indicating softintr task */
        uint8_t         softintr_pending;
        /*
```

```
         * Misc stuff
         */
        caddr_t         ioblock;        /* transfer buffer block */
        size_t          xfercnt;        /* # of bytes to transfer */
        size_t          resid;          /* # of bytes not transferred */
        caddr_t         next_byte;      /* next byte for PIO transfer */
        caddr_t         last_byte;      /* last byte for PIO transfer */
        uint32_t        ecpp_drain_counter;     /* allows fifo to drain */
        uchar_t         dma_cancelled;  /* flushed while dma'ing */
        uint8_t         tfifo_intr;     /* TFIFO switch interrupt workaround */
        size_t          nread;          /* requested read */
        size_t          last_dmacnt;    /* DMA counter value for rev watchdog */
        uint32_t        rev_timeout_cnt; /* number of watchdog invocations */
        /*
         * Spurious interrupt detection
         */
        hrtime_t        lastspur;       /* last time spurious intrs started */
        long            nspur;          /* spurious intrs counter */
        /*
         * Statistics
         */
        kstat_t         *ksp;           /* kstat pointer */
        kstat_t         *intrstats;     /* kstat interrupt counter */
        /*
         * number of bytes, transferred in and out in each mode
         */
        uint32_t        ctxpio_obytes;
        uint32_t        obytes[ECPP_EPP_MODE+1];
        uint32_t        ibytes[ECPP_EPP_MODE+1];
        /*
         * other stats
         */
        uint32_t        to_mode[ECPP_EPP_MODE+1]; /* # transitions to mode */
        uint32_t        xfer_tout;      /* # transfer timeouts */
        uint32_t        ctx_cf;         /* # periph check failures */
        uint32_t        joblen;         /* of bytes xfer'd since open */
        uint32_t        isr_reattempt_high;     /* max times isr has looped */
        /*
         * interrupt stats
         */
        uint_t          intr_hard;
        uint_t          intr_spurious;
        uint_t          intr_soft;
        /*
         * identify second register set for ecp mode on Sx86
         */
        int             noecpregs;
};

_NOTE(MUTEX_PROTECTS_DATA(ecppunit::umutex, ecppunit))
_NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::dip))
_NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::instance))
_NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::i_reg))
_NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::f_reg))
_NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::i_handle))
_NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::f_handle))
_NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::ecpp_trap_cookie))
_NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::readq))
_NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::writeq))

/*
 * current_phase values
 */
#define ECPP_PHASE_INIT         0x00    /* initialization */
#define ECPP_PHASE_NEGO         0x01    /* negotiation */
#define ECPP_PHASE_TERM         0x02    /* termination */
```

```c
#define ECPP_PHASE_PO           0x03    /* power-on */

#define ECPP_PHASE_C_FWD_DMA    0x10    /* cntrx/compat fwd dma xfer */
#define ECPP_PHASE_C_FWD_PIO    0x11    /* cntrx/compat fwd PIO xfer */
#define ECPP_PHASE_C_IDLE       0x12    /* cntrx/compat idle */

#define ECPP_PHASE_NIBT_REVDATA 0x20    /* nibble/byte reverse data */
#define ECPP_PHASE_NIBT_AVAIL   0x21    /* nibble/byte reverse data available */
#define ECPP_PHASE_NIBT_NAVAIL  0x22    /* nibble/byte reverse data not avail */
#define ECPP_PHASE_NIBT_REVIDLE 0x22    /* nibble/byte reverse idle */
#define ECPP_PHASE_NIBT_REVINTR 0x23    /* nibble/byte reverse interrupt */

#define ECPP_PHASE_ECP_SETUP    0x30    /* ecp setup */
#define ECPP_PHASE_ECP_FWD_XFER 0x31    /* ecp forward transfer */
#define ECPP_PHASE_ECP_FWD_IDLE 0x32    /* ecp forward idle */
#define ECPP_PHASE_ECP_FWD_REV  0x33    /* ecp forward to reverse */
#define ECPP_PHASE_ECP_REV_XFER 0x34    /* ecp reverse transfer */
#define ECPP_PHASE_ECP_REV_IDLE 0x35    /* ecp reverse idle */
#define ECPP_PHASE_ECP_REV_FWD  0x36    /* ecp reverse to forward */

#define ECPP_PHASE_EPP_INIT_IDLE 0x40   /* epp init phase */
#define ECPP_PHASE_EPP_IDLE     0x41    /* epp all-round phase */

#define FAILURE_PHASE           0x80
#define UNDEFINED_PHASE         0x81

/* ecpp return values */
#define SUCCESS         1
#define FAILURE         2

#define TRUE            1
#define FALSE           0

/* message type */
#define ECPP_BACKCHANNEL        0x45

/* transfer modes */
#define ECPP_DMA                0x1
#define ECPP_PIO                0x2

/* tuneable timing defaults */
#define CENTRONICS_RETRY        750     /* 750 milliseconds */
#define WAIT_FOR_BUSY           1000    /* 1000 microseconds */
#define SUSPEND_TOUT            10      /* # seconds before suspend fails */

/* Centronics hanshaking defaults */
#define DATA_SETUP_TIME         2       /* 2 uSec Data Setup Time (2x min) */
#define STROBE_PULSE_WIDTH      2       /* 2 uSec Strobe Pulse (2x min) */

/* 1284 Extensibility Request values */
#define ECPP_XREQ_NIBBLE        0x00    /* Nibble Mode Rev Channel Transfer */
#define ECPP_XREQ_BYTE          0x01    /* Byte Mode Rev Channel Transfer */
#define ECPP_XREQ_ID            0x04    /* Request Device ID */
#define ECPP_XREQ_ECP           0x10    /* Request ECP Mode */
#define ECPP_XREQ_ECPRLE        0x30    /* Request ECP Mode with RLE */
#define ECPP_XREQ_EPP           0x40    /* Request EPP Mode */
#define ECPP_XREQ_XLINK         0x80    /* Request Extensibility Link */

/* softintr flags */
#define ECPP_SOFTINTR_PIONEXT   0x1     /* write next byte in PIO mode */

/* Stream  defaults */
#define IO_BLOCK_SZ     1024 * 128      /* transfer buffer size */
#define ECPPHIWAT       32 * 1024  * 6
#define ECPPLOWAT       32 * 1024  * 4
```

```c
/* Loop timers */
#define ECPP_REG_WRITE_MAX_LOOP 100     /* cpu is faster than superio */
#define ECPP_ISR_MAX_DELAY      30      /* DMAC slow PENDING status */

/* misc constants */
#define ECPP_FIFO_SZ            16      /* FIFO size */
#define FIFO_DRAIN_PERIOD       250000  /* max FIFO drain period in usec */
#define NIBBLE_REV_BLKSZ        1024    /* send up to # bytes at a time */
#define FWD_TIMEOUT_DEFAULT     90      /* forward xfer timeout in seconds */
#define REV_TIMEOUT_DEFAULT     0       /* reverse xfer timeout in seconds */

/* ECP mode constants */
#define ECP_REV_BLKSZ           1024    /* send up to # bytes at a time */
#define ECP_REV_BLKSZ_MAX       (4 * 1024)      /* maximum of # bytes */
#define ECP_REV_SPEED           (1 * 1024 * 1024)       /* bytes/sec */
#define ECP_REV_MINTOUT         5       /* min ECP rev xfer timeout in ms */
#define REV_WATCHDOG            100     /* poll DMA counter every # ms */

/* spurious interrupt detection */
#define SPUR_CRITICAL           100     /* number of interrupts... */
#define SPUR_PERIOD             1000000000 /* in # ns */

/*
 * Copyin/copyout states
 */
#define ECPP_STRUCTIN           0
#define ECPP_STRUCTOUT          1
#define ECPP_ADDRIN             2
#define ECPP_ADDROUT            3

/*
 * As other ioctls require the same structure, put inner struct's into union
 */
struct ecpp_copystate {
        int     state;          /* see above */
        void    *uaddr;         /* user address of the following structure */
        union {
                struct ecpp_device_id           devid;
                struct prn_1284_device_id       prn_devid;
                struct prn_interface_info       prn_if;
        } un;
};

/*
 * The structure is dynamically created for each M_IOCTL and is bound to mblk
 */
_NOTE(SCHEME_PROTECTS_DATA("unique per call", ecpp_copystate))

/* kstat structure */
struct ecppkstat {
        /*
         * number of bytes, transferred in and out in each mode
         */
        struct kstat_named      ek_ctx_obytes;
        struct kstat_named      ek_ctxpio_obytes;
        struct kstat_named      ek_nib_ibytes;
        struct kstat_named      ek_ecp_obytes;
        struct kstat_named      ek_ecp_ibytes;
        struct kstat_named      ek_epp_obytes;
        struct kstat_named      ek_epp_ibytes;
        struct kstat_named      ek_diag_obytes;
        /*
         * number of transitions to particular mode
         */
        struct kstat_named      ek_to_ctx;
        struct kstat_named      ek_to_nib;
        struct kstat_named      ek_to_ecp;
        struct kstat_named      ek_to_epp;
        struct kstat_named      ek_to_diag;
        /*
         * other stats
         */
        struct kstat_named      ek_xfer_tout;   /* # transfer timeouts */
        struct kstat_named      ek_ctx_cf;      /* # periph check failures */
        struct kstat_named      ek_joblen;      /* # bytes xfer'd since open */
        struct kstat_named      ek_isr_reattempt_high;  /* max # times */
                                                        /* isr has looped */
        struct kstat_named      ek_mode;        /* 1284 mode */
        struct kstat_named      ek_phase;       /* 1284 ECP phase */
        struct kstat_named      ek_backchan;    /* backchannel mode supported */
        struct kstat_named      ek_iomode;      /* transfer mode: pio/dma */
        struct kstat_named      ek_state;       /* ecpp busy flag */
};

/* Macros for superio programming */
#define PP_PUTB(x, y, z)        ddi_put8(x, y, z)
#define PP_GETB(x, y)           ddi_get8(x, y)

#define DSR_READ(pp)            PP_GETB((pp)->i_handle, &(pp)->i_reg->dsr)
#define DCR_READ(pp)            PP_GETB((pp)->i_handle, &(pp)->i_reg->dcr)
#define ECR_READ(pp)            \
        (pp->noecpregs) ? 0xff : PP_GETB((pp)->f_handle, &(pp)->f_reg->ecr)
#define DATAR_READ(pp)          PP_GETB((pp)->i_handle, &(pp)->i_reg->ir.datar)
#define DFIFO_READ(pp)          \
        (pp->noecpregs) ? 0xff : PP_GETB((pp)->f_handle, &(pp)->f_reg->fr.dfifo)
#define TFIFO_READ(pp)          \
        (pp->noecpregs) ? 0xff : PP_GETB((pp)->f_handle, &(pp)->f_reg->fr.tfifo)

#define DCR_WRITE(pp, val)      PP_PUTB((pp)->i_handle, &(pp)->i_reg->dcr, val)
#define ECR_WRITE(pp, val)      \
        if (!pp->noecpregs) PP_PUTB((pp)->f_handle, &(pp)->f_reg->ecr, val)
#define DATAR_WRITE(pp, val)    \
                        PP_PUTB((pp)->i_handle, &(pp)->i_reg->ir.datar, val)
#define DFIFO_WRITE(pp, val)    \
        if (!pp->noecpregs) PP_PUTB((pp)->f_handle, &(pp)->f_reg->fr.dfifo, val)
#define TFIFO_WRITE(pp, val)    \
        if (!pp->noecpregs) PP_PUTB((pp)->f_handle, &(pp)->f_reg->fr.tfifo, val)

/*
 * Macros to manipulate register bits
 */
#define OR_SET_BYTE_R(handle, addr, val) \
{               \
        uint8_t tmpval;                                 \
        tmpval = ddi_get8(handle, (uint8_t *)addr);     \
        tmpval |= val;                                  \
        ddi_put8(handle, (uint8_t *)addr, tmpval);      \
}

#define OR_SET_LONG_R(handle, addr, val) \
{               \
        uint32_t tmpval;                                \
        tmpval = ddi_get32(handle, (uint32_t *)addr);   \
        tmpval |= val;                                  \
        ddi_put32(handle, (uint32_t *)addr, tmpval);    \
}

#define AND_SET_BYTE_R(handle, addr, val) \
{               \
        uint8_t tmpval;                                 \
        tmpval = ddi_get8(handle, (uint8_t *)addr);     \
        tmpval &= val;                                  \
        ddi_put8(handle, (uint8_t *)addr, tmpval);      \
}

#define AND_SET_LONG_R(handle, addr, val) \
{               \
        uint32_t tmpval;                                \
        tmpval = ddi_get32(handle, (uint32_t *)addr);   \
        tmpval &= val;                                  \
        ddi_put32(handle, (uint32_t *)addr, tmpval);    \
}

#define NOR_SET_LONG_R(handle, addr, val, mask) \
{               \
        uint32_t tmpval;                                \
        tmpval = ddi_get32(handle, (uint32_t *)addr);   \
        tmpval &= ~(mask);                              \
        tmpval |= val;                                  \
        ddi_put32(handle, (uint32_t *)addr, tmpval);    \
}

/*
 * Macros for Cheerio/RIO DMAC programming
 */
#define SET_DMAC_CSR(pp, val)   ddi_put32(pp->uh.ebus.d_handle, \
                                ((uint32_t *)&pp->uh.ebus.dmac->csr), \
                                ((uint32_t)val))
#define GET_DMAC_CSR(pp)        ddi_get32(pp->uh.ebus.d_handle, \
                                (uint32_t *)&(pp->uh.ebus.dmac->csr))

#define SET_DMAC_ACR(pp, val)   ddi_put32(pp->uh.ebus.d_handle, \
                                ((uint32_t *)&pp->uh.ebus.dmac->acr), \
                                ((uint32_t)val))

#define GET_DMAC_ACR(pp)        ddi_get32(pp->uh.ebus.d_handle, \
                                (uint32_t *)&pp->uh.ebus.dmac->acr)

#define SET_DMAC_BCR(pp, val)   ddi_put32(pp->uh.ebus.d_handle, \
                                ((uint32_t *)&pp->uh.ebus.dmac->bcr), \
                                ((uint32_t)val))

#define GET_DMAC_BCR(pp)        ddi_get32(pp->uh.ebus.d_handle, \
                                ((uint32_t *)&pp->uh.ebus.dmac->bcr))

#define DMAC_RESET_TIMEOUT      10000   /* in usec */

/*
 * Macros to distinguish between PIO and DMA Compatibility mode
 */
#define COMPAT_PIO(pp) (((pp)->io_mode == ECPP_PIO) &&          \
                    ((pp)->current_mode == ECPP_CENTRONICS ||   \
                    (pp)->current_mode == ECPP_COMPAT_MODE))

#define COMPAT_DMA(pp) (((pp)->io_mode == ECPP_DMA) &&          \
                    ((pp)->current_mode == ECPP_CENTRONICS ||   \
                    (pp)->current_mode == ECPP_COMPAT_MODE))

/*
 * Other useful macros
 */
#define NELEM(a)        (sizeof (a) / sizeof (*(a)))
#define offsetof(s, m)  ((size_t)(&(((s *)0)->m)))

#ifdef  __cplusplus
}
```

______*unchanged_portion_omitted_*

```
**********************************************************
   10091 Thu Feb 25 15:39:42 2016
new/usr/src/uts/common/sys/ib/clients/of/sol_ofs/sol_cma.h
2976 remove useless offsetof() macros
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
 */

#ifndef _SYS_IB_CLIENTS_OF_SOL_OFS_SOL_CMA_H
#define _SYS_IB_CLIENTS_OF_SOL_OFS_SOL_CMA_H

#ifdef __cplusplus
extern "C" {
#endif

#include <sys/sysmacros.h>
#endif /* ! codereview */

#include <sys/ib/clients/of/sol_ofs/sol_ofs_common.h>
#include <sys/ib/clients/of/rdma/rdma_cm.h>
#include <sys/ib/clients/of/sol_ofs/sol_ib_cma.h> /* Transport Specific */

#if !defined(offsetof)
#define offsetof(s, m)          (size_t)(&(((s *)0)->m))
#endif

#define IS_UDP_CMID(idp)        ((idp)->ps == RDMA_PS_UDP || \
        (idp)->ps == RDMA_PS_IPOIB)
#define IS_VALID_SOCKADDR(sockaddrp) \
        ((sockaddrp)->sa_family == AF_INET || \
        (sockaddrp)->sa_family == AF_INET6)

/*
 * Global structure which contains information about all
 * CMIDs, which have called rdma_listen().
 */
typedef struct sol_cma_glbl_listen_s {
        avl_node_t      cma_listen_node;

        uint64_t        cma_listen_chan_sid;
        void            *cma_listen_clnt_hdl;
        void            *cma_listen_svc_hdl;
        genlist_t       cma_listen_chan_list;
} sol_cma_glbl_listen_t;
______unchanged_portion_omitted_
```

****
12412 Thu Feb 25 15:39:42 2016
new/usr/src/uts/common/sys/sysmacros.h
3373 gcc >= 4.5 concerns about offsetof()
Portions contributed by: Igor Pashev <pashev.igor@gmail.com>
****

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */


/*
 * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 *
 * Copyright 2013 Nexenta Systems, Inc.  All rights reserved.
 */

#ifndef _SYS_SYSMACROS_H
#define _SYS_SYSMACROS_H

#include <sys/param.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * Some macros for units conversion
 */
/*
 * Disk blocks (sectors) and bytes.
 */
#define dtob(DD)        ((DD) << DEV_BSHIFT)
#define btod(BB)        (((BB) + DEV_BSIZE - 1) >> DEV_BSHIFT)
#define btodt(BB)       ((BB) >> DEV_BSHIFT)
#define lbtod(BB)       (((offset_t)(BB) + DEV_BSIZE - 1) >> DEV_BSHIFT)

/* common macros */
#ifndef MIN
#define MIN(a, b)       ((a) < (b) ? (a) : (b))
#endif
#ifndef MAX
#define MAX(a, b)       ((a) < (b) ? (b) : (a))
#endif
#ifndef ABS
#define ABS(a)          ((a) < 0 ? -(a) : (a))
```

```
#endif
#ifndef SIGNOF
#define SIGNOF(a)       ((a) < 0 ? -1 : (a) > 0)
#endif

#ifdef _KERNEL

/*
 * Convert a single byte to/from binary-coded decimal (BCD).
 */
extern unsigned char byte_to_bcd[256];
extern unsigned char bcd_to_byte[256];

#define BYTE_TO_BCD(x)  byte_to_bcd[(x) & 0xff]
#define BCD_TO_BYTE(x)  bcd_to_byte[(x) & 0xff]

#endif  /* _KERNEL */

/*
 * WARNING: The device number macros defined here should not be used by device
 * drivers or user software. Device drivers should use the device functions
 * defined in the DDI/DKI interface (see also ddi.h). Application software
 * should make use of the library routines available in makedev(3). A set of
 * new device macros are provided to operate on the expanded device number
 * format supported in SVR4. Macro versions of the DDI device functions are
 * provided for use by kernel proper routines only. Macro routines bmajor(),
 * major(), minor(), emajor(), eminor(), and makedev() will be removed or
 * their definitions changed at the next major release following SVR4.
 */

#define O_BITSMAJOR     7       /* # of SVR3 major device bits */
#define O_BITSMINOR     8       /* # of SVR3 minor device bits */
#define O_MAXMAJ        0x7f    /* SVR3 max major value */
#define O_MAXMIN        0xff    /* SVR3 max minor value */


#define L_BITSMAJOR32   14      /* # of SVR4 major device bits */
#define L_BITSMINOR32   18      /* # of SVR4 minor device bits */
#define L_MAXMAJ32      0x3fff  /* SVR4 max major value */
#define L_MAXMIN32      0x3ffff /* MAX minor for 3b2 software drivers. */
                                /* For 3b2 hardware devices the minor is */
                                /* restricted to 256 (0-255) */

#ifdef _LP64
#define L_BITSMAJOR     32      /* # of major device bits in 64-bit Solaris */
#define L_BITSMINOR     32      /* # of minor device bits in 64-bit Solaris */
#define L_MAXMAJ        0xfffffffful    /* max major value */
#define L_MAXMIN        0xfffffffful    /* max minor value */
#else
#define L_BITSMAJOR     L_BITSMAJOR32
#define L_BITSMINOR     L_BITSMINOR32
#define L_MAXMAJ        L_MAXMAJ32
#define L_MAXMIN        L_MAXMIN32
#endif

#ifdef _KERNEL

/* major part of a device internal to the kernel */

#define major(x)        (major_t)((((unsigned)(x)) >> O_BITSMINOR) & O_MAXMAJ)
#define bmajor(x)       (major_t)((((unsigned)(x)) >> O_BITSMINOR) & O_MAXMAJ)

/* get internal major part of expanded device number */

#define getmajor(x)     (major_t)((((dev_t)(x)) >> L_BITSMINOR) & L_MAXMAJ)
```

```c
/* minor part of a device internal to the kernel */

#define minor(x)        (minor_t)((x) & O_MAXMIN)

/* get internal minor part of expanded device number */

#define getminor(x)     (minor_t)((x) & L_MAXMIN)

#else   /* _KERNEL */

/* major part of a device external from the kernel (same as emajor below) */

#define major(x)        (major_t)((((unsigned)(x)) >> O_BITSMINOR) & O_MAXMAJ)

/* minor part of a device external from the kernel  (same as eminor below) */

#define minor(x)        (minor_t)((x) & O_MAXMIN)

#endif  /* _KERNEL */

/* create old device number */

#define makedev(x, y) (unsigned short)(((x) << O_BITSMINOR) | ((y) & O_MAXMIN))

/* make an new device number */

#define makedevice(x, y) (dev_t)(((dev_t)(x) << L_BITSMINOR) | ((y) & L_MAXMIN))


/*
 * emajor() allows kernel/driver code to print external major numbers
 * eminor() allows kernel/driver code to print external minor numbers
 */

#define emajor(x) \
        (major_t)(((unsigned int)(x) >> O_BITSMINOR) > O_MAXMAJ) ? \
            NODEV : (((unsigned int)(x) >> O_BITSMINOR) & O_MAXMAJ)

#define eminor(x) \
        (minor_t)((x) & O_MAXMIN)

/*
 * get external major and minor device
 * components from expanded device number
 */
#define getemajor(x)    (major_t)((((dev_t)(x) >> L_BITSMINOR) > L_MAXMAJ) ? \
                            NODEV : (((dev_t)(x) >> L_BITSMINOR) & L_MAXMAJ))
#define geteminor(x)    (minor_t)((x) & L_MAXMIN)

/*
 * These are versions of the kernel routines for compressing and
 * expanding long device numbers that don't return errors.
 */
#if (L_BITSMAJOR32 == L_BITSMAJOR) && (L_BITSMINOR32 == L_BITSMINOR)

#define DEVCMPL(x)      (x)
#define DEVEXPL(x)      (x)

#else

#define DEVCMPL(x)      \
        (dev32_t)((((x) >> L_BITSMINOR) > L_MAXMAJ32 || \
            ((x) & L_MAXMIN) > L_MAXMIN32) ? NODEV32 : \
            ((((x) >> L_BITSMINOR) << L_BITSMINOR32) | ((x) & L_MAXMIN32)))

#define DEVEXPL(x)      \
```

```c
        (((x) == NODEV32) ? NODEV : \
        makedevice(((x) >> L_BITSMINOR32) & L_MAXMAJ32, (x) & L_MAXMIN32))

#endif /* L_BITSMAJOR32 ... */

/* convert to old (SVR3.2) dev format */

#define cmpdev(x) \
        (o_dev_t)((((x) >> L_BITSMINOR) > O_MAXMAJ || \
            ((x) & L_MAXMIN) > O_MAXMIN) ? NODEV : \
            ((((x) >> L_BITSMINOR) << O_BITSMINOR) | ((x) & O_MAXMIN)))

/* convert to new (SVR4) dev format */

#define expdev(x) \
        (dev_t)(((dev_t)(((x) >> O_BITSMINOR) & O_MAXMAJ) << L_BITSMINOR) | \
            ((x) & O_MAXMIN))

/*
 * Macro for checking power of 2 address alignment.
 */
#define IS_P2ALIGNED(v, a) ((((uintptr_t)(v)) & ((uintptr_t)(a) - 1)) == 0)

/*
 * Macros for counting and rounding.
 */
#define howmany(x, y)   (((x)+((y)-1))/(y))
#define roundup(x, y)   ((((x)+((y)-1))/(y))*(y))

/*
 * Macro to determine if value is a power of 2
 */
#define ISP2(x)         (((x) & ((x) - 1)) == 0)

/*
 * Macros for various sorts of alignment and rounding.  The "align" must
 * be a power of 2.  Often times it is a block, sector, or page.
 */

/*
 * return x rounded down to an align boundary
 * eg, P2ALIGN(1200, 1024) == 1024 (1*align)
 * eg, P2ALIGN(1024, 1024) == 1024 (1*align)
 * eg, P2ALIGN(0x1234, 0x100) == 0x1200 (0x12*align)
 * eg, P2ALIGN(0x5600, 0x100) == 0x5600 (0x56*align)
 */
#define P2ALIGN(x, align)               ((x) & -(align))

/*
 * return x % (mod) align
 * eg, P2PHASE(0x1234, 0x100) == 0x34 (x-0x12*align)
 * eg, P2PHASE(0x5600, 0x100) == 0x00 (x-0x56*align)
 */
#define P2PHASE(x, align)               ((x) & ((align) - 1))

/*
 * return how much space is left in this block (but if it's perfectly
 * aligned, return 0).
 * eg, P2NPHASE(0x1234, 0x100) == 0xcc (0x13*align-x)
 * eg, P2NPHASE(0x5600, 0x100) == 0x00 (0x56*align-x)
 */
#define P2NPHASE(x, align)              (-(x) & ((align) - 1))

/*
 * return x rounded up to an align boundary
 * eg, P2ROUNDUP(0x1234, 0x100) == 0x1300 (0x13*align)
 * eg, P2ROUNDUP(0x5600, 0x100) == 0x5600 (0x56*align)
 */
#define P2ROUNDUP(x, align)             (-(-(x) & -(align)))

/*
 * return the ending address of the block that x is in
 * eg, P2END(0x1234, 0x100) == 0x12ff (0x13*align - 1)
 * eg, P2END(0x5600, 0x100) == 0x56ff (0x57*align - 1)
 */
#define P2END(x, align)                 (-(~(x) & -(align)))

/*
 * return x rounded up to the next phase (offset) within align.
 * phase should be < align.
 * eg, P2PHASEUP(0x1234, 0x100, 0x10) == 0x1310 (0x13*align + phase)
 * eg, P2PHASEUP(0x5600, 0x100, 0x10) == 0x5610 (0x56*align + phase)
 */
#define P2PHASEUP(x, align, phase)      ((phase) - (((phase) - (x)) & -(align)))

/*
 * return TRUE if adding len to off would cause it to cross an align
 * boundary.
 * eg, P2BOUNDARY(0x1234, 0xe0, 0x100) == TRUE (0x1234 + 0xe0 == 0x1314)
 * eg, P2BOUNDARY(0x1234, 0x50, 0x100) == FALSE (0x1234 + 0x50 == 0x1284)
 */
#define P2BOUNDARY(off, len, align) \
        (((off) ^ ((off) + (len) - 1)) > (align) - 1)

/*
 * Return TRUE if they have the same highest bit set.
 * eg, P2SAMEHIGHBIT(0x1234, 0x1001) == TRUE (the high bit is 0x1000)
 * eg, P2SAMEHIGHBIT(0x1234, 0x3010) == FALSE (high bit of 0x3010 is 0x2000)
 */
#define P2SAMEHIGHBIT(x, y)             (((x) ^ (y)) < ((x) & (y)))

/*
 * Typed version of the P2* macros.  These macros should be used to ensure
 * that the result is correctly calculated based on the data type of (x),
 * which is passed in as the last argument, regardless of the data
 * type of the alignment.  For example, if (x) is of type uint64_t,
 * and we want to round it up to a page boundary using "PAGESIZE" as
 * the alignment, we can do either
 *      P2ROUNDUP(x, (uint64_t)PAGESIZE)
 * or
 *      P2ROUNDUP_TYPED(x, PAGESIZE, uint64_t)
 */
#define P2ALIGN_TYPED(x, align, type)   \
        ((type)(x) & -(type)(align))
#define P2PHASE_TYPED(x, align, type)   \
        ((type)(x) & ((type)(align) - 1))
#define P2NPHASE_TYPED(x, align, type)  \
        (-(type)(x) & ((type)(align) - 1))
#define P2ROUNDUP_TYPED(x, align, type) \
        (-(-(type)(x) & -(type)(align)))
#define P2END_TYPED(x, align, type)     \
        (-(~(type)(x) & -(type)(align)))
#define P2PHASEUP_TYPED(x, align, phase, type)  \
        ((type)(phase) - (((type)(phase) - (type)(x)) & -(type)(align)))
#define P2CROSS_TYPED(x, y, align, type)        \
        (((type)(x) ^ (type)(y)) > (type)(align) - 1)
#define P2SAMEHIGHBIT_TYPED(x, y, type) \
        (((type)(x) ^ (type)(y)) < ((type)(x) & (type)(y)))

/*
 * Macros to atomically increment/decrement a variable.  mutex and var
 * must be pointers.
```

```
 */
#define INCR_COUNT(var, mutex) mutex_enter(mutex), (*(var))++, mutex_exit(mutex)
#define DECR_COUNT(var, mutex) mutex_enter(mutex), (*(var))--, mutex_exit(mutex)

/*
 * Macros to declare bitfields - the order in the parameter list is
 * Low to High - that is, declare bit 0 first.  We only support 8-bit bitfields
 * because if a field crosses a byte boundary it's not likely to be meaningful
 * without reassembly in its nonnative endianness.
 */
#if defined(_BIT_FIELDS_LTOH)
#define DECL_BITFIELD2(_a, _b)                          \
        uint8_t _a, _b
#define DECL_BITFIELD3(_a, _b, _c)                      \
        uint8_t _a, _b, _c
#define DECL_BITFIELD4(_a, _b, _c, _d)                  \
        uint8_t _a, _b, _c, _d
#define DECL_BITFIELD5(_a, _b, _c, _d, _e)              \
        uint8_t _a, _b, _c, _d, _e
#define DECL_BITFIELD6(_a, _b, _c, _d, _e, _f)          \
        uint8_t _a, _b, _c, _d, _e, _f
#define DECL_BITFIELD7(_a, _b, _c, _d, _e, _f, _g)      \
        uint8_t _a, _b, _c, _d, _e, _f, _g
#define DECL_BITFIELD8(_a, _b, _c, _d, _e, _f, _g, _h)  \
        uint8_t _a, _b, _c, _d, _e, _f, _g, _h
#elif defined(_BIT_FIELDS_HTOL)
#define DECL_BITFIELD2(_a, _b)                          \
        uint8_t _b, _a
#define DECL_BITFIELD3(_a, _b, _c)                      \
        uint8_t _c, _b, _a
#define DECL_BITFIELD4(_a, _b, _c, _d)                  \
        uint8_t _d, _c, _b, _a
#define DECL_BITFIELD5(_a, _b, _c, _d, _e)              \
        uint8_t _e, _d, _c, _b, _a
#define DECL_BITFIELD6(_a, _b, _c, _d, _e, _f)          \
        uint8_t _f, _e, _d, _c, _b, _a
#define DECL_BITFIELD7(_a, _b, _c, _d, _e, _f, _g)      \
        uint8_t _g, _f, _e, _d, _c, _b, _a
#define DECL_BITFIELD8(_a, _b, _c, _d, _e, _f, _g, _h)  \
        uint8_t _h, _g, _f, _e, _d, _c, _b, _a
#else
#error  One of _BIT_FIELDS_LTOH or _BIT_FIELDS_HTOL must be defined
#endif  /* _BIT_FIELDS_LTOH */

/* avoid any possibility of clashing with <stddef.h> version */
#if (defined(_KERNEL) || defined(_FAKE_KERNEL)) && !defined(_KMEMUSER)

#if !defined(offsetof)
#if __GNUC__ > 4 || (__GNUC__ == 4 && __GNUC_MINOR__ >= 5)
#define offsetof(s, m) __builtin_offsetof(s, m)
#else
#endif /* ! codereview */
#define offsetof(s, m)  ((size_t)(&(((s *)0)->m)))
#endif
#endif /* ! codereview */
#endif /* !offsetof */

#define container_of(m, s, name)                        \
        (void *)((uintptr_t)(m) - (uintptr_t)offsetof(s, name))

#define ARRAY_SIZE(x)   (sizeof (x) / sizeof (x[0]))
#endif /* _KERNEL, !_KMEMUSER */

#ifdef  __cplusplus
}
#endif
```

```
#endif  /* _SYS_SYSMACROS_H */
```

```
**********************************************************
   10191 Thu Feb 25 15:39:43 2016
new/usr/src/uts/common/sys/usb/clients/audio/usb_ac/usb_ac.h
2976 remove useless offsetof() macros
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _SYS_USB_AC_H
#define _SYS_USB_AC_H



#ifdef __cplusplus
extern "C" {
#endif

#include <sys/sunldi.h>
#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/usb/usba/usbai_private.h>


int usb_ac_open(dev_info_t *);
void usb_ac_close(dev_info_t *);


/* structure for each unit described by descriptors */
typedef struct usb_ac_unit_list {
        uint_t          acu_type;
        void            *acu_descriptor;
        size_t          acu_descr_length;
} usb_ac_unit_list_t;

#define USB_AC_ID_NONE                  0

#define USB_AC_FIND_ONE                 0
#define USB_AC_FIND_ALL                 1
#define USB_AC_MAX_DEPTH                8

/*
 * plumbing data; info per plumbed module
 */
typedef struct usb_ac_plumbed {
```

```
        struct usb_ac_state *acp_uacp;  /* usb_ac state pointer */
        dev_info_t      *acp_dip;       /* devinfo pointer */
        uint_t          acp_ifno;       /* interface number */
        int             acp_driver;     /* Plumbed driver, see value below */

        ldi_handle_t    acp_lh;         /* ldi handle of plumbed driver */
        dev_t           acp_devt;       /* devt of plumbed driver */
        ddi_taskq_t     *acp_tqp;       /* taskq for I/O to plumbed driver */
        int             acp_flags;
#define ACP_ENABLED     1

        void            *acp_data;      /* ptr to streams or hid data */
} usb_ac_plumbed_t;


/*
 * request structure to usb_as: info per MCTL request;
 * only one active at a time.
 */
typedef struct usb_ac_to_as_req {
        usb_audio_formats_t acr_curr_format; /* format data from mixer */
} usb_ac_to_as_req_t;


/* registration and plumbing info per streaming interface */
typedef struct usb_ac_streams_info {
                                        /* ptr to entry in plumbed list */
        usb_ac_plumbed_t *acs_plumbed;
                                        /* valid registration data rcvd */
        uint_t          acs_rcvd_reg_data;
                                        /* pointer to registration data */
        usb_as_registration_t acs_streams_reg;


        /* Multiple command management */
        int             acs_setup_teardown_count;

        uint8_t         acs_default_gain;
} usb_ac_streams_info_t;


/* power state */
typedef struct usb_ac_power {
        void            *acpm_state;    /* points back to usb_ac_state */
        int             acpm_pm_busy;   /* device busy accounting */
        uint8_t         acpm_wakeup_enabled;

        /* this is the bit mask of the power states that device has */
        uint8_t         acpm_pwr_states;

        /* wakeup and power transistion capabilites of an interface */
        uint8_t         acpm_capabilities;

        /* current power level the device is in */
        uint8_t         acpm_current_power;
} usb_ac_power_t;

_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_power_t::acpm_state))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_power_t::acpm_wakeup_enabled))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_power_t::acpm_pwr_states))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_power_t::acpm_capabilities))

typedef struct usb_audio_format {
        int             sr;     /* sample rate */
        uint_t          ch;     /* channels */
        uint_t          prec;   /* precision */
        uint_t          enc;    /* encoding */
} usb_audio_format_t;


typedef struct usb_audio_eng {
        void  *statep;
        usb_ac_streams_info_t *streams;
        audio_engine_t  *af_engp;

        int             af_eflags;      /* ENGINE_* flags */
        usb_audio_format_t      fmt;
        uint64_t        af_defgain;

        unsigned        intrate;        /* interrupt rate */
        unsigned        sampsz;         /* sample size */
        unsigned        framesz;        /* frame size */
        unsigned        fragsz;         /* fragment size */
        unsigned        nfrags;         /* number of fragments in buffer */
        unsigned        fragfr;         /* number of frames per fragment */
        unsigned        frsmshift;      /* right shift: frames in sample cnt */
        unsigned        smszshift;      /* left shift: sample cnt * sampsz */


        caddr_t         bufp;           /* I/O buf; framework to/from drv */
        unsigned        bufsz;          /* buffer size */
        caddr_t         bufpos;         /* buffer position */
        caddr_t         bufendp;        /* end of buffer */


        uint64_t        frames;
        uint64_t        io_count;       /* i/o requests from the driver */
        uint64_t        bufio_count;    /* i/o requests to the framework */

        boolean_t       started;
        boolean_t       busy;

        kcondvar_t      usb_audio_cv;

        kmutex_t        lock;
} usb_audio_eng_t;


/* limits */
#define USB_AC_MAX_PLUMBED              3       /* play, record, hid */
#define USB_AC_MAX_AS_PLUMBED           2       /* play, record */
typedef struct usb_ac_state  usb_ac_state_t;
typedef struct usb_audio_ctrl {
        audio_ctrl_t            *af_ctrlp;      /* framework handle */
        usb_ac_state_t          *statep;

        kmutex_t        ctrl_mutex;
        uint64_t                cval;           /* current control value */
} usb_audio_ctrl_t;

enum {
        CTL_VOLUME_MONO = 0,
        CTL_VOLUME_STERO,
        CTL_REC_MONO,
        CTL_REC_STERO,
        CTL_REC_SRC,
        CTL_MONITOR_GAIN,
        CTL_MIC_BOOST,
        CTL_NUM
};

#define USB_AC_ENG_MAX   2
```

```

/* usb_ac soft state */
struct usb_ac_state {

        dev_info_t              *usb_ac_dip;
        uint_t                  usb_ac_instance;
        usb_log_handle_t        usb_ac_log_handle;

        uint_t                  usb_ac_dev_state;
        uint_t                  usb_ac_ifno;
        kmutex_t                usb_ac_mutex;

        usb_client_dev_data_t   *usb_ac_dev_data; /* registration data */
        audio_dev_t             *usb_ac_audio_dev;




        usb_audio_eng_t  engines[USB_AC_ENG_MAX];



        int             flags;
        usb_audio_ctrl_t        *controls[CTL_NUM];

        /* descriptors */
        usb_if_descr_t          usb_ac_if_descr;

        /* unit number array, indexed by unit ID */
        uint_t                  usb_ac_max_unit;
        usb_ac_unit_list_t      *usb_ac_units;

        /* adjacency matrix for reflecting connections */
        uchar_t                 **usb_ac_connections;
        size_t                  usb_ac_connections_len;
        uchar_t                 *usb_ac_connections_a;
        size_t                  usb_ac_connections_a_len;
        uchar_t                 *usb_ac_unit_type;
        uchar_t                 *usb_ac_traverse_path;
        uchar_t                 usb_ac_traverse_path_index;

        /* port types, eg LINE IN, Micr, Speakers */
        uint64_t                        usb_ac_input_ports;
        uint64_t                        usb_ac_output_ports;

        /* pipe handle */
        usb_pipe_handle_t       usb_ac_default_ph;

        /* serial access */
        usb_serialization_t     usb_ac_ser_acc;

        /* power management */
        usb_ac_power_t          *usb_ac_pm; /* power capabilities */

        /* mixer registration data */
        uint_t                  usb_ac_registered_with_mixer;

        /* plumbing management */
        uint_t                  usb_ac_plumbing_state;
        ushort_t                usb_ac_busy_count;
        usb_ac_plumbed_t        usb_ac_plumbed[USB_AC_MAX_PLUMBED];

        /* Current plumbed module index to usb_ac_plumbed structure */
        int                     usb_ac_current_plumbed_index;

        /* per streams interface info */
        usb_ac_streams_info_t   usb_ac_streams[USB_AC_MAX_AS_PLUMBED];


        ddi_taskq_t             *tqp;

        char                    dstr[64];
};

/* warlock directives, stable data */
_NOTE(MUTEX_PROTECTS_DATA(usb_ac_state_t::usb_ac_mutex, usb_ac_state_t))
_NOTE(MUTEX_PROTECTS_DATA(usb_ac_state_t::usb_ac_mutex, usb_ac_power_t))
_NOTE(MUTEX_PROTECTS_DATA(usb_ac_state_t::usb_ac_mutex, usb_ac_plumbed_t))
_NOTE(MUTEX_PROTECTS_DATA(usb_audio_eng_t::lock, usb_audio_eng_t))
_NOTE(MUTEX_PROTECTS_DATA(usb_audio_eng_t::lock, usb_audio_format_t))
_NOTE(MUTEX_PROTECTS_DATA(usb_audio_ctrl_t::ctrl_mutex, usb_audio_ctrl_t))


_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_dip))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_ser_acc))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_pm))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_instance))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_default_ph))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_log_handle))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_if_descr))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_dev_data))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_ifno))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::flags))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_input_ports))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::engines))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_audio_dev))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::controls))

_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::af_eflags))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::streams))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::statep))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::fmt))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::fragfr))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::frsmshift))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::started))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::af_engp))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::io_count))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::intrate))

_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_ctrl_t::statep))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_ctrl_t::af_ctrlp))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_ctrl_t::cval))

_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_plumbed_t::acp_tqp))
_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_plumbed_t::acp_uacp))

_NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_format_t::ch))

/* usb_ac driver only care about two states:  plumbed or unplumbed */
#define USB_AC_STATE_UNPLUMBED          0
#define USB_AC_STATE_PLUMBED            1
#define USB_AC_STATE_PLUMBED_RESTORING  2

/* Default pipe states */
#define USB_AC_DEF_CLOSED               0
#define USB_AC_DEF_OPENED               1

#define USB_AC_BUFFER_SIZE              256     /* descriptor buffer size */


/*
 * delay before restoring state
```

```
 */
#define USB_AC_RESTORE_DELAY            drv_usectohz(1000000)

/* value for acp_driver */
#define USB_AS_PLUMBED  1
#define USB_AH_PLUMBED  2
#define UNKNOWN_PLUMBED 3

/* other useful macros */
#define offsetof(s, m)  ((size_t)(&(((s *)0)->m)))






#define AF_REGISTERED   0x1
#define AD_SETUP        0x10


int usb_audio_attach(usb_ac_state_t *);
/*
 * framework gain range
 */
#define AUDIO_CTRL_STEREO_VAL(l, r)     (((l) & 0xff) | (((r) & 0xff) << 8))
#define AUDIO_CTRL_STEREO_LEFT(v)       ((uint8_t)((v) & 0xff))
#define AUDIO_CTRL_STEREO_RIGHT(v)      ((uint8_t)(((v) >> 8) & 0xff))


#define AF_MAX_GAIN     100
#define AF_MIN_GAIN     0



int usb_ac_get_audio(void *, void *, int);

void usb_ac_send_audio(void *, void *, int);

void usb_ac_stop_play(usb_ac_state_t *, usb_audio_eng_t *);


#ifdef __cplusplus
}
```

______***unchanged_portion_omitted_***

```
**********************************************************
   11708 Thu Feb 25 15:39:43 2016
new/usr/src/uts/common/sys/usb/clients/video/usbvc/usbvc_var.h
2976 remove useless offsetof() macros
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _SYS_USB_USBVC_VAR_H
#define _SYS_USB_USBVC_VAR_H


#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/list.h>
#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <sys/usb/usba/usbai_private.h>
#include <sys/videodev2.h>
#include <sys/usb/clients/video/usbvc/usbvc.h>

typedef struct usbvc_state usbvc_state_t;

/*
 * Power Management support
 */
typedef struct usbvc_power  {

        void            *usbvc_state;   /* points back to usbvc_state */
        uint8_t         usbvc_pwr_states; /* bit mask of device pwr states */
        int             usbvc_pm_busy;

        /* Wakeup and power transistion capabilites of an interface */
        uint8_t         usbvc_pm_capabilities;

        /* flag to indicate if driver is about to raise power level */
        boolean_t       usbvc_raise_power;

        uint8_t         usbvc_current_power;
        uint8_t         usbvc_wakeup_enabled;
} usbvc_power_t;
```

```
/* Raw data buf from the USB cam */
typedef struct usbvc_buf
{
        uchar_t *data;
        uint_t len;     /* the length of the allocated memory of data */
        uint_t filled;  /* number of bytes filled */
        uint_t len_read; /* bytes read */
        uchar_t status; /* empty, filling done, read done */

        /* cookie used for memory mapping */
        ddi_umem_cookie_t       umem_cookie;
        struct                  v4l2_buffer v4l2_buf;
        list_node_t             buf_node;       /* list */
} usbvc_buf_t;

/* Group data buf related lists and other elements */
typedef struct usbvc_buf_grp
{
    list_t              uv_buf_free;
        list_t          uv_buf_done;
        usbvc_buf_t     *buf_filling;
        uint_t          buf_cnt;
        usbvc_buf_t     *buf_head;
} usbvc_buf_grp_t;

/*
 * UVC Spec: one format descriptor may be followed by sererval frame
 * descriptors, one still image descriptor and one color matching descriptor.
 * It is called a format group. There might be several format groups follow
 * one input/output header.
 */
typedef struct usbvc_format_group {
        usbvc_format_descr_t    *format;
        usbvc_frames_t          *frames;
        uint8_t                 frame_cnt;

        /* bytes per pix, used to calculate bytesperline */
        uint8_t                 v4l2_bpp;

        uint8_t                 v4l2_color;
        uint32_t                v4l2_pixelformat;       /* fcc, pixelformat */
        usbvc_still_image_frame_t       *still;
        usbvc_color_matching_descr_t    *color;
        usbvc_frames_t                  *cur_frame;
} usbvc_format_group_t;

/* A stream interface may have several format groups */
typedef struct usbvc_stream_if {

        /* The actual format groups we parsed for the stream interface */
        uint8_t                 fmtgrp_cnt;

        usb_if_data_t           *if_descr;
        usbvc_input_header_t    *input_header;
        usbvc_output_header_t   *output_header;
        usbvc_format_group_t    *format_group;
        usbvc_format_group_t    *cur_format_group;
        usbvc_vs_probe_commit_t ctrl_pc;
        usb_ep_descr_t          *curr_ep;       /* current isoc ep descr */
        usb_pipe_handle_t       datain_ph;      /* current isoc pipe handle */
        uint_t                  curr_alt;       /* current alternate  */

        /* The max payload that the isoc data EPs can support */
        uint32_t        max_isoc_payload;

        uchar_t         start_polling;  /* indicate if isoc polling started */
        /*
         * To flag if VIDIOC_STREAMON is executed, only used by STREAM mode
         * for suspend/resume. If it's non-zero, we'll have to resume the
         * device's isoc polling operation after resume.
         */
        uint8_t         stream_on;

        uchar_t         fid;            /* the MJPEG FID bit */
        usbvc_buf_grp_t buf_read;       /* buf used for read I/O */
        uint8_t                 buf_read_num; /* desired buf num for read I/O */
        usbvc_buf_grp_t buf_map;        /* buf used for mmap I/O */
        list_node_t     stream_if_node;
} usbvc_stream_if_t;

/* video interface collection */
typedef struct usbvc_vic {

        /* bFirstInterface, the video control infterface num of this VIC */
        uint8_t vctrl_if_num;

        /*
         * bInterfaceCount -1, the total number of stream interfaces
         * belong to this VIC
         */
        uint8_t vstrm_if_cnt;
} usbvc_vic_t;

/* Macros */
#define USBVC_OPEN              0x00000001

/* For serialization. */
#define USBVC_SER_NOSIG B_FALSE
#define USBVC_SER_SIG           B_TRUE

/*
 * Masks for debug printing
 */
#define PRINT_MASK_ATTA         0x00000001
#define PRINT_MASK_OPEN         0x00000002
#define PRINT_MASK_CLOSE        0x00000004
#define PRINT_MASK_READ         0x00000008
#define PRINT_MASK_IOCTL        0x00000010
#define PRINT_MASK_PM   0x00000020
#define PRINT_MASK_CB   0x00000040
#define PRINT_MASK_HOTPLUG      0x00000080
#define PRINT_MASK_DEVCTRL      0x00000100
#define PRINT_MASK_DEVMAP       0x00000200
#define PRINT_MASK_ALL          0xFFFFFFFF

#define offsetof(s, m)  ((size_t)(&(((s *)0)->m)))

#define USBVC_MAX_PKTS 40

#define USBVC_DEFAULT_READ_BUF_NUM 3
#define USBVC_MAX_READ_BUF_NUM 40
#define USBVC_MAX_MAP_BUF_NUM 40

/* According to UVC specs, the frame interval is in 100ns unit */
#define USBVC_FRAME_INTERVAL_DENOMINATOR        10000000

/* Only D3...D0 are writable, Table 4-6, UVC Spec */
#define USBVC_POWER_MODE_MASK   0xf0;

enum usbvc_buf_status {
        USBVC_BUF_INIT          = 0,  /* Allocated, to be queued */
```

```
        USBVC_BUF_MAPPED    = 1,  /* For map I/O only. Memory is mapped. */
        USBVC_BUF_EMPTY             = 2, /* not initialized, to be filled */

        /*
         * buf is filled with a full frame without any errors,
         * it will be moved to full list.
         */
        USBVC_BUF_DONE              = 4,

        /*
         * buf is filled to full but no EOF bit is found at the end
         * of video data
         */
        USBVC_BUF_ERR               = 8
};
______unchanged_portion_omitted_
```

```
**********************************************************
    5932 Thu Feb 25 15:39:43 2016
new/usr/src/uts/common/xen/io/xnb.h
3373 gcc >= 4.5 concerns about offsetof()
Portions contributed by: Igor Pashev <pashev.igor@gmail.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 *
 * xnb.h - definitions for Xen dom0 network driver
 */

#ifndef _SYS_XNB_H
#define _SYS_XNB_H

#include <sys/types.h>
#include <sys/kstat.h>
#include <sys/stream.h>
#include <sys/ethernet.h>
#include <sys/hypervisor.h>
#include <sys/sysmacros.h>
#endif /* ! codereview */
#include <xen/public/io/netif.h>

#ifdef __cplusplus
extern "C" {
#endif

#define NET_TX_RING_SIZE  __CONST_RING_SIZE(netif_tx, PAGESIZE)
#define NET_RX_RING_SIZE  __CONST_RING_SIZE(netif_rx, PAGESIZE)
#define NET_TX_RING_SIZE  __RING_SIZE((netif_tx_sring_t *)0, PAGESIZE)
#define NET_RX_RING_SIZE  __RING_SIZE((netif_rx_sring_t *)0, PAGESIZE)

#define XNBMAXPKT       1500            /* MTU size */

/* DEBUG flags */
#define XNBDDI          0x01
#define XNBTRACE        0x02
#define XNBSEND         0x04
#define XNBRECV         0x08
#define XNBINTR         0x10
#define XNBRING         0x20
#define XNBCKSUM        0x40
```

```
#define XNB_STATE_INIT  0x01
#define XNB_STATE_READY 0x02

typedef struct xnb xnb_t;

/*
 * The xnb module provides core inter-domain network protocol functionality.
 * It is connected to the rest of Solaris in two ways:
 * - as a GLDv3 driver (with xnbu),
 * - as a GLDv3 consumer (with xnbo).
 *
 * The different modes of operation are termed "flavours" and each
 * instance of an xnb based driver operates in one and only one mode.
 * The common xnb driver exports a set of functions to these drivers
 * (declarations at the foot of this file) and calls back into the
 * drivers via the xnb_flavour_t structure.
 */
typedef struct xnb_flavour {
        void            (*xf_from_peer)(xnb_t *, mblk_t *);
        boolean_t       (*xf_peer_connected)(xnb_t *);
        void            (*xf_peer_disconnected)(xnb_t *);
        boolean_t       (*xf_hotplug_connected)(xnb_t *);
        boolean_t       (*xf_start_connect)(xnb_t *);
        mblk_t          *(*xf_cksum_from_peer)(xnb_t *, mblk_t *, uint16_t);
        uint16_t        (*xf_cksum_to_peer)(xnb_t *, mblk_t *);
        boolean_t       (*xf_mcast_add)(xnb_t *, ether_addr_t *);
        boolean_t       (*xf_mcast_del)(xnb_t *, ether_addr_t *);
} xnb_flavour_t;
______unchanged_portion_omitted_
```

4750 Thu Feb 25 15:39:44 2016
new/usr/src/uts/common/xen/io/xnf.h
3373 gcc >= 4.5 concerns about offsetof()
Portions contributed by: Igor Pashev <pashev.igor@gmail.com>

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _SYS_XNF_H
#define _SYS_XNF_H

#ifdef __cplusplus
extern "C" {
#endif

#define NET_TX_RING_SIZE  __CONST_RING_SIZE(netif_tx, PAGESIZE)
#define NET_RX_RING_SIZE  __CONST_RING_SIZE(netif_rx, PAGESIZE)
#define NET_TX_RING_SIZE  __RING_SIZE((netif_tx_sring_t *)0, PAGESIZE)
#define NET_RX_RING_SIZE  __RING_SIZE((netif_rx_sring_t *)0, PAGESIZE)

#define XNF_MAXPKT      1500            /* MTU size */
#define XNF_FRAMESIZE   1514            /* frame size including MAC header */

/* DEBUG flags */
#define XNF_DEBUG_DDI           0x01
#define XNF_DEBUG_TRACE         0x02

/*
 * Information about each receive buffer and any transmit look-aside
 * buffers.
 */
typedef struct xnf_buf {
        frtn_t                  free_rtn;
        struct xnf              *xnfp;
        ddi_dma_handle_t        dma_handle;
        caddr_t                 buf;            /* DMA-able data buffer */
        paddr_t                 buf_phys;
        mfn_t                   buf_mfn;
        size_t                  len;
        struct xnf_buf          *next;  /* For linking into free list */
        ddi_acc_handle_t        acc_handle;
        grant_ref_t             grant_ref;      /* grant table reference */
```

```
        uint16_t                id;             /* buffer id */
        unsigned int            gen;
} xnf_buf_t;
```
______***unchanged_portion_omitted_***

```
**********************************************************
   15078 Thu Feb 25 15:39:44 2016
new/usr/src/uts/common/xen/public/io/ring.h
3373 gcc >= 4.5 concerns about offsetof()
Portions contributed by: Igor Pashev <pashev.igor@gmail.com>
**********************************************************
/******************************************************************************
 * ring.h
 *
 * Shared producer-consumer ring macros.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to
 * deal in the Software without restriction, including without limitation the
 * rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
 * sell copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
 * DEALINGS IN THE SOFTWARE.
 *
 * Tim Deegan and Andrew Warfield November 2004.
 */

#ifndef __XEN_PUBLIC_IO_RING_H__
#define __XEN_PUBLIC_IO_RING_H__

#include "../xen-compat.h"

#if __XEN_INTERFACE_VERSION__ < 0x00030208
#define xen_mb()  mb()
#define xen_rmb() rmb()
#define xen_wmb() wmb()
#endif

typedef unsigned int RING_IDX;

/* Round a 32-bit unsigned constant down to the nearest power of two. */
#define __RD2(_x)  (((_x) & 0x00000002) ? 0x2                  : ((_x) & 0x1))
#define __RD4(_x)  (((_x) & 0x0000000c) ? __RD2((_x)>>2)<<2    : __RD2(_x))
#define __RD8(_x)  (((_x) & 0x000000f0) ? __RD4((_x)>>4)<<4    : __RD4(_x))
#define __RD16(_x) (((_x) & 0x0000ff00) ? __RD8((_x)>>8)<<8    : __RD8(_x))
#define __RD32(_x) (((_x) & 0xffff0000) ? __RD16((_x)>>16)<<16 : __RD16(_x))

/*
 * Calculate size of a shared ring, given the total available space for the
 * ring and indexes (_sz), and the name tag of the request/response structure.
 * A ring contains as many entries as will fit, rounded down to the nearest
 * power of two (so we can mask with (size-1) to loop around).
 */
#define __CONST_RING_SIZE(_s, _sz) \
    (__RD32(((_sz) - offsetof(struct _s##_sring, ring)) / \
        sizeof(((struct _s##_sring *)0)->ring[0])))
/*
 * The same for passing in an actual pointer instead of a name tag.
 */
#endif /* ! codereview */
#define __RING_SIZE(_s, _sz) \
```

```
    (__RD32(((_sz) - (long)(_s)->ring + (long)(_s)) / sizeof((_s)->ring[0])))

/*
 * Macros to make the correct C datatypes for a new kind of ring.
 *
 * To make a new ring datatype, you need to have two message structures,
 * let’s say request_t, and response_t already defined.
 *
 * In a header where you want the ring datatype declared, you then do:
 *
 *     DEFINE_RING_TYPES(mytag, request_t, response_t);
 *
 * These expand out to give you a set of types, as you can see below.
 * The most important of these are:
 *
 *     mytag_sring_t      - The shared ring.
 *     mytag_front_ring_t - The ’front’ half of the ring.
 *     mytag_back_ring_t  - The ’back’ half of the ring.
 *
 * To initialize a ring in your code you need to know the location and size
 * of the shared memory area (PAGE_SIZE, for instance). To initialise
 * the front half:
 *
 *     mytag_front_ring_t front_ring;
 *     SHARED_RING_INIT((mytag_sring_t *)shared_page);
 *     FRONT_RING_INIT(&front_ring, (mytag_sring_t *)shared_page, PAGE_SIZE);
 *
 * Initializing the back follows similarly (note that only the front
 * initializes the shared ring):
 *
 *     mytag_back_ring_t back_ring;
 *     BACK_RING_INIT(&back_ring, (mytag_sring_t *)shared_page, PAGE_SIZE);
 */

#define DEFINE_RING_TYPES(__name, __req_t, __rsp_t)                     \
                                                                        \
/* Shared ring entry */                                                 \
union __name##_sring_entry {                                            \
    __req_t req;                                                        \
    __rsp_t rsp;                                                        \
};                                                                      \
                                                                        \
/* Shared ring page */                                                  \
struct __name##_sring {                                                 \
    RING_IDX req_prod, req_event;                                       \
    RING_IDX rsp_prod, rsp_event;                                       \
    uint8_t  pad[48];                                                   \
    union __name##_sring_entry ring[1]; /* variable-length */           \
};                                                                      \
                                                                        \
/* "Front" end’s private variables */                                   \
struct __name##_front_ring {                                            \
    RING_IDX req_prod_pvt;                                              \
    RING_IDX rsp_cons;                                                  \
    unsigned int nr_ents;                                               \
    struct __name##_sring *sring;                                       \
};                                                                      \
                                                                        \
/* "Back" end’s private variables */                                    \
struct __name##_back_ring {                                             \
    RING_IDX rsp_prod_pvt;                                              \
    RING_IDX req_cons;                                                  \
    unsigned int nr_ents;                                               \
    struct __name##_sring *sring;                                       \
};                                                                      \
                                                                        \
/* Syntactic sugar */                                                   \
typedef struct __name##_sring __name##_sring_t;                         \
typedef struct __name##_front_ring __name##_front_ring_t;               \
typedef struct __name##_back_ring __name##_back_ring_t

/*
 * Macros for manipulating rings.
 *
 * FRONT_RING_whatever works on the "front end" of a ring: here
 * requests are pushed on to the ring and responses taken off it.
 *
 * BACK_RING_whatever works on the "back end" of a ring: here
 * requests are taken off the ring and responses put on.
 *
 * N.B. these macros do NO INTERLOCKS OR FLOW CONTROL.
 * This is OK in 1-for-1 request-response situations where the
 * requestor (front end) never has more than RING_SIZE()-1
 * outstanding requests.
 */

/* Initialising empty rings */
#define SHARED_RING_INIT(_s) do {                                       \
    (_s)->req_prod  = (_s)->rsp_prod  = 0;                              \
    (_s)->req_event = (_s)->rsp_event = 1;                              \
    (void)memset((_s)->pad, 0, sizeof((_s)->pad));                      \
} while(0)

#define FRONT_RING_INIT(_r, _s, __size) do {                            \
    (_r)->req_prod_pvt = 0;                                             \
    (_r)->rsp_cons = 0;                                                 \
    (_r)->nr_ents = __RING_SIZE(_s, __size);                            \
    (_r)->sring = (_s);                                                 \
} while (0)

#define BACK_RING_INIT(_r, _s, __size) do {                             \
    (_r)->rsp_prod_pvt = 0;                                             \
    (_r)->req_cons = 0;                                                 \
    (_r)->nr_ents = __RING_SIZE(_s, __size);                            \
    (_r)->sring = (_s);                                                 \
} while (0)

/* Initialize to existing shared indexes -- for recovery */
#define FRONT_RING_ATTACH(_r, _s, __size) do {                          \
    (_r)->sring = (_s);                                                 \
    (_r)->req_prod_pvt = (_s)->req_prod;                                \
    (_r)->rsp_cons = (_s)->rsp_prod;                                    \
    (_r)->nr_ents = __RING_SIZE(_s, __size);                            \
} while (0)

#define BACK_RING_ATTACH(_r, _s, __size) do {                           \
    (_r)->sring = (_s);                                                 \
    (_r)->rsp_prod_pvt = (_s)->rsp_prod;                                \
    (_r)->req_cons = (_s)->req_prod;                                    \
    (_r)->nr_ents = __RING_SIZE(_s, __size);                            \
} while (0)

/* How big is this ring? */
#define RING_SIZE(_r)                                                   \
    ((_r)->nr_ents)

/* Number of free requests (for use on front side only). */
#define RING_FREE_REQUESTS(_r)                                          \
    (RING_SIZE(_r) - ((_r)->req_prod_pvt - (_r)->rsp_cons))

/* Test if there is an empty slot available on the front ring.
 * (This is only meaningful from the front. )
```

```
 */
#define RING_FULL(_r)                                                   \
    (RING_FREE_REQUESTS(_r) == 0)

/* Test if there are outstanding messages to be processed on a ring. */
#define RING_HAS_UNCONSUMED_RESPONSES(_r)                               \
    ((_r)->sring->rsp_prod - (_r)->rsp_cons)

#ifdef __GNUC__
#define RING_HAS_UNCONSUMED_REQUESTS(_r) ({                             \
    unsigned int req = (_r)->sring->req_prod - (_r)->req_cons;          \
    unsigned int rsp = RING_SIZE(_r) -                                  \
        ((_r)->req_cons - (_r)->rsp_prod_pvt);                          \
    req < rsp ? req : rsp;                                              \
})
#else
/* Same as above, but without the nice GCC ({ ... }) syntax. */
#define RING_HAS_UNCONSUMED_REQUESTS(_r)                                \
    ((((_r)->sring->req_prod - (_r)->req_cons) <                        \
      (RING_SIZE(_r) - ((_r)->req_cons - (_r)->rsp_prod_pvt))) ?        \
     ((_r)->sring->req_prod - (_r)->req_cons) :                         \
     (RING_SIZE(_r) - ((_r)->req_cons - (_r)->rsp_prod_pvt)))
#endif

/* Direct access to individual ring elements, by index. */
#define RING_GET_REQUEST(_r, _idx)                                      \
    (&((_r)->sring->ring[((_idx) & (RING_SIZE(_r) - 1))].req))

#define RING_GET_RESPONSE(_r, _idx)                                     \
    (&((_r)->sring->ring[((_idx) & (RING_SIZE(_r) - 1))].rsp))

/* Loop termination condition: Would the specified index overflow the ring? */
#define RING_REQUEST_CONS_OVERFLOW(_r, _cons)                           \
    (((_cons) - (_r)->rsp_prod_pvt) >= RING_SIZE(_r))

#define RING_PUSH_REQUESTS(_r) do {                                     \
    xen_wmb(); /* back sees requests /before/ updated producer index */ \
    (_r)->sring->req_prod = (_r)->req_prod_pvt;                         \
} while (0)

#define RING_PUSH_RESPONSES(_r) do {                                    \
    xen_wmb(); /* front sees resps /before/ updated producer index */   \
    (_r)->sring->rsp_prod = (_r)->rsp_prod_pvt;                         \
} while (0)

/*
 * Notification hold-off (req_event and rsp_event):
 *
 * When queueing requests or responses on a shared ring, it may not always be
 * necessary to notify the remote end. For example, if requests are in flight
 * in a backend, the front may be able to queue further requests without
 * notifying the back (if the back checks for new requests when it queues
 * responses).
 *
 * When enqueuing requests or responses:
 *
 *  Use RING_PUSH_{REQUESTS,RESPONSES}_AND_CHECK_NOTIFY(). The second argument
 *  is a boolean return value. True indicates that the receiver requires an
 *  asynchronous notification.
 *
 * After dequeuing requests or responses (before sleeping the connection):
 *
 *  Use RING_FINAL_CHECK_FOR_REQUESTS() or RING_FINAL_CHECK_FOR_RESPONSES().
 *  The second argument is a boolean return value. True indicates that there
 *  are pending messages on the ring (i.e., the connection should not be put
 *  to sleep).
 *
 *  These macros will set the req_event/rsp_event field to trigger a
 *  notification on the very next message that is enqueued. If you want to
 *  create batches of work (i.e., only receive a notification after several
 *  messages have been enqueued) then you will need to create a customised
 *  version of the FINAL_CHECK macro in your own code, which sets the event
 *  field appropriately.
 */

#define RING_PUSH_REQUESTS_AND_CHECK_NOTIFY(_r, _notify) do {           \
    RING_IDX __old = (_r)->sring->req_prod;                             \
    RING_IDX __new = (_r)->req_prod_pvt;                                \
    xen_wmb(); /* back sees requests /before/ updated producer index */ \
    (_r)->sring->req_prod = __new;                                      \
    xen_mb(); /* back sees new requests /before/ we check req_event */  \
    (_notify) = ((RING_IDX)(__new - (_r)->sring->req_event) <           \
                 (RING_IDX)(__new - __old));                            \
} while (0)

#define RING_PUSH_RESPONSES_AND_CHECK_NOTIFY(_r, _notify) do {          \
    RING_IDX __old = (_r)->sring->rsp_prod;                             \
    RING_IDX __new = (_r)->rsp_prod_pvt;                                \
    xen_wmb(); /* front sees resps /before/ updated producer index */   \
    (_r)->sring->rsp_prod = __new;                                      \
    xen_mb(); /* front sees new resps /before/ we check rsp_event */    \
    (_notify) = ((RING_IDX)(__new - (_r)->sring->rsp_event) <           \
                 (RING_IDX)(__new - __old));                            \
} while (0)

#define RING_FINAL_CHECK_FOR_REQUESTS(_r, _work_to_do) do {             \
    (_work_to_do) = RING_HAS_UNCONSUMED_REQUESTS(_r);                   \
    if (_work_to_do) break;                                             \
    (_r)->sring->req_event = (_r)->req_cons + 1;                        \
    xen_mb();                                                           \
    (_work_to_do) = RING_HAS_UNCONSUMED_REQUESTS(_r);                   \
} while (0)

#define RING_FINAL_CHECK_FOR_RESPONSES(_r, _work_to_do) do {            \
    (_work_to_do) = RING_HAS_UNCONSUMED_RESPONSES(_r);                  \
    if (_work_to_do) break;                                             \
    (_r)->sring->rsp_event = (_r)->rsp_cons + 1;                        \
    xen_mb();                                                           \
    (_work_to_do) = RING_HAS_UNCONSUMED_RESPONSES(_r);                  \
} while (0)

#endif /* __XEN_PUBLIC_IO_RING_H__ */

/*
 * Local variables:
 * mode: C
 * c-set-style: "BSD"
 * c-basic-offset: 4
 * tab-width: 4
 * indent-tabs-mode: nil
 * End:
 */
```

```
**********************************************************
  194604 Thu Feb 25 15:39:45 2016
new/usr/src/uts/sun/io/scsi/adapters/sf.c
2976 remove useless offsetof() macros
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 * Copyright (c) 2011 Bayard G. Bell. All rights reserved.
 */

/*
 * sf - Solaris Fibre Channel driver
 *
 * This module implements some of the Fibre Channel FC-4 layer, converting
 * from FC frames to SCSI and back.  (Note: no sequence management is done
 * here, though.)
 */

#if defined(lint) && !defined(DEBUG)
#define DEBUG   1
#endif

/*
 * XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 * Need to use the ugly RAID LUN mappings in FCP Annex D
 * to prevent SCSA from barfing.  This *REALLY* needs to
 * be addressed by the standards committee.
 */
#define RAID_LUNS       1

#ifdef DEBUG
static int sfdebug = 0;
#include <sys/debug.h>

#define SF_DEBUG(level, args) \
        if (sfdebug >= (level)) sf_log args
#else
#define SF_DEBUG(level, args)
#endif

static int sf_bus_config_debug = 0;

/* Why do I have to do this? */
#define offsetof(s, m)  (size_t)(&(((s *)0)->m))
```

```
#include <sys/scsi/scsi.h>
#include <sys/fc4/fcal.h>
#include <sys/fc4/fcp.h>
#include <sys/fc4/fcal_linkapp.h>
#include <sys/socal_cq_defs.h>
#include <sys/fc4/fcal_transport.h>
#include <sys/fc4/fcio.h>
#include <sys/scsi/adapters/sfvar.h>
#include <sys/scsi/impl/scsi_reset_notify.h>
#include <sys/stat.h>
#include <sys/varargs.h>
#include <sys/var.h>
#include <sys/thread.h>
#include <sys/proc.h>
#include <sys/kstat.h>
#include <sys/devctl.h>
#include <sys/scsi/targets/ses.h>
#include <sys/callb.h>
#include <sys/sysmacros.h>
#endif /* ! codereview */

static int sf_info(dev_info_t *, ddi_info_cmd_t, void *, void **);
static int sf_attach(dev_info_t *, ddi_attach_cmd_t);
static int sf_detach(dev_info_t *, ddi_detach_cmd_t);
static void sf_softstate_unlink(struct sf *);
static int sf_scsi_bus_config(dev_info_t *parent, uint_t flag,
    ddi_bus_config_op_t op, void *arg, dev_info_t **childp);
static int sf_scsi_bus_unconfig(dev_info_t *parent, uint_t flag,
    ddi_bus_config_op_t op, void *arg);
static int sf_scsi_tgt_init(dev_info_t *, dev_info_t *,
    scsi_hba_tran_t *, struct scsi_device *);
static void sf_scsi_tgt_free(dev_info_t *, dev_info_t *,
    scsi_hba_tran_t *, struct scsi_device *);
static int sf_pkt_alloc_extern(struct sf *, struct sf_pkt *,
    int, int, int);
static void sf_pkt_destroy_extern(struct sf *, struct sf_pkt *);
static struct scsi_pkt *sf_scsi_init_pkt(struct scsi_address *,
    struct scsi_pkt *, struct buf *, int, int, int, int, int (*)(), caddr_t);
static void sf_scsi_destroy_pkt(struct scsi_address *, struct scsi_pkt *);
static void sf_scsi_dmafree(struct scsi_address *, struct scsi_pkt *);
static void sf_scsi_sync_pkt(struct scsi_address *, struct scsi_pkt *);
static int sf_scsi_reset_notify(struct scsi_address *, int,
    void (*)(caddr_t), caddr_t);
static int sf_scsi_get_name(struct scsi_device *, char *, int);
static int sf_scsi_get_bus_addr(struct scsi_device *, char *, int);
static int sf_add_cr_pool(struct sf *);
static int sf_cr_alloc(struct sf *, struct sf_pkt *, int (*)());
static void sf_cr_free(struct sf_cr_pool *, struct sf_pkt *);
static void sf_crpool_free(struct sf *);
static int sf_kmem_cache_constructor(void *, void *, int);
static void sf_kmem_cache_destructor(void *, void *);
static void sf_statec_callback(void *, int);
static int sf_login(struct sf *, uchar_t, uchar_t, uint_t, int);
static int sf_els_transport(struct sf *, struct sf_els_hdr *);
static void sf_els_callback(struct fcal_packet *);
static int sf_do_prli(struct sf *, struct sf_els_hdr *, struct la_els_logi *);
static int sf_do_adisc(struct sf *, struct sf_els_hdr *);
static int sf_do_reportlun(struct sf *, struct sf_els_hdr *,
    struct sf_target *);
static void sf_reportlun_callback(struct fcal_packet *);
static int sf_do_inquiry(struct sf *, struct sf_els_hdr *,
    struct sf_target *);
static void sf_inq_callback(struct fcal_packet *);
static struct fcal_packet *sf_els_alloc(struct sf *, uchar_t, int, int,
    int, caddr_t *, caddr_t *);
static void sf_els_free(struct fcal_packet *);
static struct sf_target *sf_create_target(struct sf *,
    struct sf_els_hdr *, int, int64_t);
#ifdef RAID_LUNS
static struct sf_target *sf_lookup_target(struct sf *, uchar_t *, int);
#else
static struct sf_target *sf_lookup_target(struct sf *, uchar_t *, int64_t);
#endif
static void sf_finish_init(struct sf *, int);
static void sf_offline_target(struct sf *, struct sf_target *);
static void sf_create_devinfo(struct sf *, struct sf_target *, int);
static int sf_create_props(dev_info_t *, struct sf_target *, int);
static int sf_commoncap(struct scsi_address *, char *, int, int, int);
static int sf_getcap(struct scsi_address *, char *, int);
static int sf_setcap(struct scsi_address *, char *, int, int);
static int sf_abort(struct scsi_address *, struct scsi_pkt *);
static int sf_reset(struct scsi_address *, int);
static void sf_abort_all(struct sf *, struct sf_target *, int, int, int);
static int sf_start(struct scsi_address *, struct scsi_pkt *);
static int sf_start_internal(struct sf *, struct sf_pkt *);
static void sf_fill_ids(struct sf *, struct sf_pkt *, struct sf_target *);
static int sf_prepare_pkt(struct sf *, struct sf_pkt *, struct sf_target *);
static int sf_dopoll(struct sf *, struct sf_pkt *);
static void sf_cmd_callback(struct fcal_packet *);
static void sf_throttle(struct sf *);
static void sf_watch(void *);
static void sf_throttle_start(struct sf *);
static void sf_check_targets(struct sf *);
static void sf_check_reset_delay(void *);
static int sf_target_timeout(struct sf *, struct sf_pkt *);
static void sf_force_lip(struct sf *);
static void sf_unsol_els_callback(void *, soc_response_t *, caddr_t);
static struct sf_els_hdr *sf_els_timeout(struct sf *, struct sf_els_hdr *);
/*PRINTFLIKE3*/
static void sf_log(struct sf *, int, const char *, ...);
static int sf_kstat_update(kstat_t *, int);
static int sf_open(dev_t *, int, int, cred_t *);
static int sf_close(dev_t, int, int, cred_t *);
static int sf_ioctl(dev_t, int, intptr_t, int, cred_t *, int *);
static struct sf_target *sf_get_target_from_dip(struct sf *, dev_info_t *);
static int sf_bus_get_eventcookie(dev_info_t *, dev_info_t *, char *,
    ddi_eventcookie_t *);
static int sf_bus_add_eventcall(dev_info_t *, dev_info_t *,
    ddi_eventcookie_t, void (*)(), void *, ddi_callback_id_t *cb_id);
static int sf_bus_remove_eventcall(dev_info_t *devi, ddi_callback_id_t cb_id);
static int sf_bus_post_event(dev_info_t *, dev_info_t *,
    ddi_eventcookie_t, void *);

static void sf_hp_daemon(void *);

/*
 * this is required to be able to supply a control node
 * where ioctls can be executed
 */
struct cb_ops sf_cb_ops = {
        sf_open,                        /* open */
        sf_close,                       /* close */
        nodev,                          /* strategy */
        nodev,                          /* print */
        nodev,                          /* dump */
        nodev,                          /* read */
        nodev,                          /* write */
        sf_ioctl,                       /* ioctl */
        nodev,                          /* devmap */
        nodev,                          /* mmap */
        nodev,                          /* segmap */
        nochpoll,                       /* poll */
```

```
        ddi_prop_op,                    /* cb_prop_op */
        0,                              /* streamtab  */
        D_MP | D_NEW | D_HOTPLUG        /* driver flags */

};

/*
 * autoconfiguration routines.
 */
static struct dev_ops sf_ops = {
        DEVO_REV,               /* devo_rev, */
        0,                      /* refcnt  */
        sf_info,                /* info */
        nulldev,                /* identify */
        nulldev,                /* probe */
        sf_attach,              /* attach */
        sf_detach,              /* detach */
        nodev,                  /* reset */
        &sf_cb_ops,             /* driver operations */
        NULL,                   /* bus operations */
        NULL,                   /* power management */
        ddi_quiesce_not_supported,      /* devo_quiesce */
};

#define SF_NAME "FC-AL FCP Nexus Driver"        /* Name of the module. */
static  char    sf_version[] = "1.72 08/19/2008"; /* version of the module */

static struct modldrv modldrv = {
        &mod_driverops, /* Type of module. This one is a driver */
        SF_NAME,
        &sf_ops,        /* driver ops */
};

static struct modlinkage modlinkage = {
        MODREV_1, (void *)&modldrv, NULL
};

/* XXXXXX The following is here to handle broken targets -- remove it later */
static int sf_reportlun_forever = 0;
/* XXXXXX */
static int sf_lip_on_plogo = 0;
static int sf_els_retries = SF_ELS_RETRIES;
static struct sf *sf_head = NULL;
static int sf_target_scan_cnt = 4;
static int sf_pkt_scan_cnt = 5;
static int sf_pool_scan_cnt = 1800;
static void *sf_state = NULL;
static int sf_watchdog_init = 0;
static int sf_watchdog_time = 0;
static int sf_watchdog_timeout = 1;
static int sf_watchdog_tick;
static int sf_watch_running = 0;
static timeout_id_t sf_watchdog_id;
static timeout_id_t sf_reset_timeout_id;
static int sf_max_targets = SF_MAX_TARGETS;
static kmutex_t sf_global_mutex;
static int sf_core = 0;
int *sf_token = NULL; /* Must not be static or lint complains. */
static kcondvar_t sf_watch_cv;
extern pri_t minclsyspri;
static ddi_eventcookie_t        sf_insert_eid;
static ddi_eventcookie_t        sf_remove_eid;

static ndi_event_definition_t   sf_event_defs[] = {
{ SF_EVENT_TAG_INSERT, FCAL_INSERT_EVENT, EPL_KERNEL, 0 },
{ SF_EVENT_TAG_REMOVE, FCAL_REMOVE_EVENT, EPL_INTERRUPT, 0 }
```

```
};

#define SF_N_NDI_EVENTS \
        (sizeof (sf_event_defs) / sizeof (ndi_event_definition_t))

#ifdef DEBUG
static int sf_lip_flag = 1;             /* bool: to allow LIPs */
static int sf_reset_flag = 1;           /* bool: to allow reset after LIP */
static int sf_abort_flag = 0;           /* bool: to do just one abort */
#endif

extern int64_t ddi_get_lbolt64(void);

/*
 * for converting between target number (switch) and hard address/AL_PA
 */
static uchar_t sf_switch_to_alpa[] = {
        0xef, 0xe8, 0xe4, 0xe2, 0xe1, 0xe0, 0xdc, 0xda, 0xd9, 0xd6,
        0xd5, 0xd4, 0xd3, 0xd2, 0xd1, 0xce, 0xcd, 0xcc, 0xcb, 0xca,
        0xc9, 0xc7, 0xc6, 0xc5, 0xc3, 0xbc, 0xba, 0xb9, 0xb6, 0xb5,
        0xb4, 0xb3, 0xb2, 0xb1, 0xae, 0xad, 0xac, 0xab, 0xaa, 0xa9,
        0xa7, 0xa6, 0xa5, 0xa3, 0x9f, 0x9e, 0x9d, 0x9b, 0x98, 0x97,
        0x90, 0x8f, 0x88, 0x84, 0x82, 0x81, 0x80, 0x7c, 0x7a, 0x79,
        0x76, 0x75, 0x74, 0x73, 0x72, 0x71, 0x6e, 0x6d, 0x6c, 0x6b,
        0x6a, 0x69, 0x67, 0x66, 0x65, 0x63, 0x5c, 0x5a, 0x59, 0x56,
        0x55, 0x54, 0x53, 0x52, 0x51, 0x4e, 0x4d, 0x4c, 0x4b, 0x4a,
        0x49, 0x47, 0x46, 0x45, 0x43, 0x3c, 0x3a, 0x39, 0x36, 0x35,
        0x34, 0x33, 0x32, 0x31, 0x2e, 0x2d, 0x2c, 0x2b, 0x2a, 0x29,
        0x27, 0x26, 0x25, 0x23, 0x1f, 0x1e, 0x1d, 0x1b, 0x18, 0x17,
        0x10, 0x0f, 0x08, 0x04, 0x02, 0x01
};

static uchar_t sf_alpa_to_switch[] = {
        0x00, 0x7d, 0x7c, 0x00, 0x7b, 0x00, 0x00, 0x00, 0x7a, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x79, 0x78, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x77, 0x76, 0x00, 0x00, 0x75, 0x00, 0x74,
        0x73, 0x72, 0x00, 0x00, 0x00, 0x71, 0x00, 0x70, 0x6f, 0x6e,
        0x00, 0x6d, 0x6c, 0x6b, 0x6a, 0x69, 0x68, 0x00, 0x00, 0x67,
        0x66, 0x65, 0x64, 0x63, 0x62, 0x00, 0x00, 0x61, 0x60, 0x00,
        0x5f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x5e, 0x00, 0x5d,
        0x5c, 0x5b, 0x00, 0x5a, 0x59, 0x58, 0x57, 0x56, 0x55, 0x00,
        0x00, 0x54, 0x53, 0x52, 0x51, 0x50, 0x4f, 0x00, 0x00, 0x4e,
        0x4d, 0x00, 0x4c, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x4b,
        0x00, 0x4a, 0x49, 0x48, 0x00, 0x47, 0x46, 0x45, 0x44, 0x43,
        0x42, 0x00, 0x00, 0x41, 0x40, 0x3f, 0x3e, 0x3d, 0x3c, 0x00,
        0x00, 0x3b, 0x3a, 0x00, 0x39, 0x00, 0x00, 0x00, 0x38, 0x37,
        0x36, 0x00, 0x35, 0x00, 0x00, 0x00, 0x34, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x33, 0x32, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x31, 0x30, 0x00, 0x00, 0x2f, 0x00, 0x2e, 0x2d, 0x2c,
        0x00, 0x00, 0x00, 0x2b, 0x00, 0x2a, 0x29, 0x28, 0x00, 0x27,
        0x26, 0x25, 0x24, 0x23, 0x22, 0x00, 0x00, 0x21, 0x20, 0x1f,
        0x1e, 0x1d, 0x1c, 0x00, 0x00, 0x1b, 0x1a, 0x00, 0x19, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x00, 0x17, 0x16, 0x15,
        0x00, 0x14, 0x13, 0x12, 0x11, 0x10, 0x0f, 0x00, 0x00, 0x0e,
        0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x00, 0x00, 0x08, 0x07, 0x00,
        0x06, 0x00, 0x00, 0x00, 0x05, 0x04, 0x03, 0x00, 0x02, 0x00,
        0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
};

/*
 * these macros call the proper transport-layer function given
 * a particular transport
 */
#define soc_transport(a, b, c, d) (*a->fcal_ops->fcal_transport)(b, c, d)
#define soc_transport_poll(a, b, c, d)\
        (*a->fcal_ops->fcal_transport_poll)(b, c, d)
```

```
#define soc_get_lilp_map(a, b, c, d, e)\
        (*a->fcal_ops->fcal_lilp_map)(b, c, d, e)
#define soc_force_lip(a, b, c, d, e)\
        (*a->fcal_ops->fcal_force_lip)(b, c, d, e)
#define soc_abort(a, b, c, d, e)\
        (*a->fcal_ops->fcal_abort_cmd)(b, c, d, e)
#define soc_force_reset(a, b, c, d)\
        (*a->fcal_ops->fcal_force_reset)(b, c, d)
#define soc_add_ulp(a, b, c, d, e, f, g, h)\
        (*a->fcal_ops->fcal_add_ulp)(b, c, d, e, f, g, h)
#define soc_remove_ulp(a, b, c, d, e)\
        (*a->fcal_ops->fcal_remove_ulp)(b, c, d, e)
#define soc_take_core(a, b) (*a->fcal_ops->fcal_take_core)(b)


/* power management property defines (should be in a common include file?) */
#define PM_HARDWARE_STATE_PROP          "pm-hardware-state"
#define PM_NEEDS_SUSPEND_RESUME         "needs-suspend-resume"


/* node properties */
#define NODE_WWN_PROP                   "node-wwn"
#define PORT_WWN_PROP                   "port-wwn"
#define LIP_CNT_PROP                    "lip-count"
#define TARGET_PROP                     "target"
#define LUN_PROP                        "lun"


/*
 * initialize this driver and install this module
 */
int
_init(void)
{
        int     i;

        i = ddi_soft_state_init(&sf_state, sizeof (struct sf),
            SF_INIT_ITEMS);
        if (i != 0)
                return (i);

        if ((i = scsi_hba_init(&modlinkage)) != 0) {
                ddi_soft_state_fini(&sf_state);
                return (i);
        }

        mutex_init(&sf_global_mutex, NULL, MUTEX_DRIVER, NULL);
        sf_watch_running = 0;
        cv_init(&sf_watch_cv, NULL, CV_DRIVER, NULL);

        if ((i = mod_install(&modlinkage)) != 0) {
                mutex_destroy(&sf_global_mutex);
                cv_destroy(&sf_watch_cv);
                scsi_hba_fini(&modlinkage);
                ddi_soft_state_fini(&sf_state);
                return (i);
        }

        return (i);
}


/*
 * remove this driver module from the system
 */
int
```

```
_fini(void)
{
        int     i;

        if ((i = mod_remove(&modlinkage)) == 0) {
                scsi_hba_fini(&modlinkage);
                mutex_destroy(&sf_global_mutex);
                cv_destroy(&sf_watch_cv);
                ddi_soft_state_fini(&sf_state);
        }
        return (i);
}


int
_info(struct modinfo *modinfop)
{
        return (mod_info(&modlinkage, modinfop));
}

/*
 * Given the device number return the devinfo pointer or instance
 */
/*ARGSUSED*/
static int
sf_info(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
{
        int             instance = SF_MINOR2INST(getminor((dev_t)arg));
        struct sf       *sf;

        switch (infocmd) {
        case DDI_INFO_DEVT2DEVINFO:
                sf = ddi_get_soft_state(sf_state, instance);
                if (sf != NULL)
                        *result = sf->sf_dip;
                else {
                        *result = NULL;
                        return (DDI_FAILURE);
                }
                break;

        case DDI_INFO_DEVT2INSTANCE:
                *result = (void *)(uintptr_t)instance;
                break;
        default:
                return (DDI_FAILURE);
        }
        return (DDI_SUCCESS);
}

/*
 * either attach or resume this driver
 */
static int
sf_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        int instance;
        int mutex_initted = FALSE;
        uint_t ccount;
        size_t i, real_size;
        struct fcal_transport *handle;
        char buf[64];
        struct sf *sf, *tsf;
        scsi_hba_tran_t *tran = NULL;
        int     handle_bound = FALSE;
        kthread_t *tp;
```

```

        switch ((int)cmd) {

        case DDI_RESUME:

                /*
                 * we've previously been SF_STATE_OFFLINEd by a DDI_SUSPEND,
                 * so time to undo that and get going again by forcing a
                 * lip
                 */

                instance = ddi_get_instance(dip);

                sf = ddi_get_soft_state(sf_state, instance);
                SF_DEBUG(2, (sf, CE_CONT,
                    "sf_attach: DDI_RESUME for sf%d\n", instance));
                if (sf == NULL) {
                        cmn_err(CE_WARN, "sf%d: bad soft state", instance);
                        return (DDI_FAILURE);
                }

                /*
                 * clear suspended flag so that normal operations can resume
                 */
                mutex_enter(&sf->sf_mutex);
                sf->sf_state &= ~SF_STATE_SUSPENDED;
                mutex_exit(&sf->sf_mutex);

                /*
                 * force a login by setting our state to offline
                 */
                sf->sf_timer = sf_watchdog_time + SF_OFFLINE_TIMEOUT;
                sf->sf_state = SF_STATE_OFFLINE;

                /*
                 * call transport routine to register state change and
                 * ELS callback routines (to register us as a ULP)
                 */
                soc_add_ulp(sf->sf_sochandle, sf->sf_socp,
                    sf->sf_sochandle->fcal_portno, TYPE_SCSI_FCP,
                    sf_statec_callback, sf_unsol_els_callback, NULL, sf);

                /*
                 * call transport routine to force loop initialization
                 */
                (void) soc_force_lip(sf->sf_sochandle, sf->sf_socp,
                    sf->sf_sochandle->fcal_portno, 0, FCAL_NO_LIP);

                /*
                 * increment watchdog init flag, setting watchdog timeout
                 * if we are the first (since somebody has to do it)
                 */
                mutex_enter(&sf_global_mutex);
                if (!sf_watchdog_init++) {
                        mutex_exit(&sf_global_mutex);
                        sf_watchdog_id = timeout(sf_watch,
                            (caddr_t)0, sf_watchdog_tick);
                } else {
                        mutex_exit(&sf_global_mutex);
                }

                return (DDI_SUCCESS);

        case DDI_ATTACH:
```

```
		/*
		 * this instance attaching for the first time
		 */

		instance = ddi_get_instance(dip);

		if (ddi_soft_state_zalloc(sf_state, instance) !=
		    DDI_SUCCESS) {
			cmn_err(CE_WARN, "sf%d: failed to allocate soft state",
			    instance);
			return (DDI_FAILURE);
		}

		sf = ddi_get_soft_state(sf_state, instance);
		SF_DEBUG(4, (sf, CE_CONT,
		    "sf_attach: DDI_ATTACH for sf%d\n", instance));
		if (sf == NULL) {
			/* this shouldn’t happen since we just allocated it */
			cmn_err(CE_WARN, "sf%d: bad soft state", instance);
			return (DDI_FAILURE);
		}

		/*
		 * from this point on, if there’s an error, we must de-allocate
		 * soft state before returning DDI_FAILURE
		 */

		if ((handle = ddi_get_parent_data(dip)) == NULL) {
			cmn_err(CE_WARN,
			    "sf%d: failed to obtain transport handle",
			    instance);
			goto fail;
		}

		/* fill in our soft state structure */
		sf->sf_dip = dip;
		sf->sf_state = SF_STATE_INIT;
		sf->sf_throttle = handle->fcal_cmdmax;
		sf->sf_sochandle = handle;
		sf->sf_socp = handle->fcal_handle;
		sf->sf_check_n_close = 0;

		/* create a command/response buffer pool for this instance */
		if (sf_add_cr_pool(sf) != DDI_SUCCESS) {
			cmn_err(CE_WARN,
			    "sf%d: failed to allocate command/response pool",
			    instance);
			goto fail;
		}

		/* create a a cache for this instance */
		(void) sprintf(buf, "sf%d_cache", instance);
		sf->sf_pkt_cache = kmem_cache_create(buf,
		    sizeof (fcal_packet_t) + sizeof (struct sf_pkt) +
		    scsi_pkt_size(), 8,
		    sf_kmem_cache_constructor, sf_kmem_cache_destructor,
		    NULL, NULL, NULL, 0);
		if (sf->sf_pkt_cache == NULL) {
			cmn_err(CE_WARN, "sf%d: failed to allocate kmem cache",
			    instance);
			goto fail;
		}

		/* set up a handle and allocate memory for DMA */
		if (ddi_dma_alloc_handle(sf->sf_dip, sf->sf_sochandle->
		    fcal_dmaattr, DDI_DMA_DONTWAIT, NULL, &sf->
```

```
		    sf_lilp_dmahandle) != DDI_SUCCESS) {
			cmn_err(CE_WARN,
			    "sf%d: failed to allocate dma handle for lilp map",
			    instance);
			goto fail;
		}
		i = sizeof (struct fcal_lilp_map) + 1;
		if (ddi_dma_mem_alloc(sf->sf_lilp_dmahandle,
		    i, sf->sf_sochandle->
		    fcal_accattr, DDI_DMA_CONSISTENT, DDI_DMA_DONTWAIT, NULL,
		    (caddr_t *)&sf->sf_lilp_map, &real_size,
		    &sf->sf_lilp_acchandle) != DDI_SUCCESS) {
			cmn_err(CE_WARN, "sf%d: failed to allocate lilp map",
			    instance);
			goto fail;
		}
		if (real_size < i) {
			/* no error message ??? */
			goto fail;		/* trouble allocating memory */
		}

		/*
		 * set up the address for the DMA transfers (getting a cookie)
		 */
		if (ddi_dma_addr_bind_handle(sf->sf_lilp_dmahandle, NULL,
		    (caddr_t)sf->sf_lilp_map, real_size,
		    DDI_DMA_READ | DDI_DMA_CONSISTENT, DDI_DMA_DONTWAIT, NULL,
		    &sf->sf_lilp_dmacookie, &ccount) != DDI_DMA_MAPPED) {
			cmn_err(CE_WARN,
			    "sf%d: failed to bind dma handle for lilp map",
			    instance);
			goto fail;
		}
		handle_bound = TRUE;
		/* ensure only one cookie was allocated */
		if (ccount != 1) {
			goto fail;
		}

		/* ensure LILP map and DMA cookie addresses are even?? */
		sf->sf_lilp_map = (struct fcal_lilp_map *)(((uintptr_t)sf->
		    sf_lilp_map + 1) & ~1);
		sf->sf_lilp_dmacookie.dmac_address = (sf->
		    sf_lilp_dmacookie.dmac_address + 1) & ~1;

		/* set up all of our mutexes and condition variables */
		mutex_init(&sf->sf_mutex, NULL, MUTEX_DRIVER, NULL);
		mutex_init(&sf->sf_cmd_mutex, NULL, MUTEX_DRIVER, NULL);
		mutex_init(&sf->sf_cr_mutex, NULL, MUTEX_DRIVER, NULL);
		mutex_init(&sf->sf_hp_daemon_mutex, NULL, MUTEX_DRIVER, NULL);
		cv_init(&sf->sf_cr_cv, NULL, CV_DRIVER, NULL);
		cv_init(&sf->sf_hp_daemon_cv, NULL, CV_DRIVER, NULL);

		mutex_initted = TRUE;

		/* create our devctl minor node */
		if (ddi_create_minor_node(dip, "devctl", S_IFCHR,
		    SF_INST2DEVCTL_MINOR(instance),
		    DDI_NT_NEXUS, 0) != DDI_SUCCESS) {
			cmn_err(CE_WARN, "sf%d: ddi_create_minor_node failed"
			    " for devctl", instance);
			goto fail;
		}

		/* create fc minor node */
		if (ddi_create_minor_node(dip, "fc", S_IFCHR,
```

```c
                    SF_INST2FC_MINOR(instance), DDI_NT_FC_ATTACHMENT_POINT,
                    0) != DDI_SUCCESS) {
                        cmn_err(CE_WARN, "sf%d: ddi_create_minor_node failed"
                            " for fc", instance);
                        goto fail;
                }
                /* allocate a SCSI transport structure */
                tran = scsi_hba_tran_alloc(dip, 0);
                if (tran == NULL) {
                        /* remove all minor nodes created */
                        ddi_remove_minor_node(dip, NULL);
                        cmn_err(CE_WARN, "sf%d: scsi_hba_tran_alloc failed",
                            instance);
                        goto fail;
                }

                /* Indicate that we are 'sizeof (scsi_*(9S))' clean. */
                scsi_size_clean(dip);           /* SCSI_SIZE_CLEAN_VERIFY ok */

                /* save ptr to new transport structure and fill it in */
                sf->sf_tran = tran;

                tran->tran_hba_private          = sf;
                tran->tran_tgt_private          = NULL;
                tran->tran_tgt_init             = sf_scsi_tgt_init;
                tran->tran_tgt_probe            = NULL;
                tran->tran_tgt_free             = sf_scsi_tgt_free;

                tran->tran_start                = sf_start;
                tran->tran_abort                = sf_abort;
                tran->tran_reset                = sf_reset;
                tran->tran_getcap               = sf_getcap;
                tran->tran_setcap               = sf_setcap;
                tran->tran_init_pkt             = sf_scsi_init_pkt;
                tran->tran_destroy_pkt          = sf_scsi_destroy_pkt;
                tran->tran_dmafree              = sf_scsi_dmafree;
                tran->tran_sync_pkt             = sf_scsi_sync_pkt;
                tran->tran_reset_notify         = sf_scsi_reset_notify;

                /*
                 * register event notification routines with scsa
                 */
                tran->tran_get_eventcookie      = sf_bus_get_eventcookie;
                tran->tran_add_eventcall        = sf_bus_add_eventcall;
                tran->tran_remove_eventcall     = sf_bus_remove_eventcall;
                tran->tran_post_event           = sf_bus_post_event;

                /*
                 * register bus configure/unconfigure
                 */
                tran->tran_bus_config           = sf_scsi_bus_config;
                tran->tran_bus_unconfig         = sf_scsi_bus_unconfig;

                /*
                 * allocate an ndi event handle
                 */
                sf->sf_event_defs = (ndi_event_definition_t *)
                    kmem_zalloc(sizeof (sf_event_defs), KM_SLEEP);

                bcopy(sf_event_defs, sf->sf_event_defs,
                    sizeof (sf_event_defs));

                (void) ndi_event_alloc_hdl(dip, NULL,
                    &sf->sf_event_hdl, NDI_SLEEP);

                sf->sf_events.ndi_events_version = NDI_EVENTS_REV1;
```

```c
                sf->sf_events.ndi_n_events = SF_N_NDI_EVENTS;
                sf->sf_events.ndi_event_defs = sf->sf_event_defs;

                if (ndi_event_bind_set(sf->sf_event_hdl,
                    &sf->sf_events, NDI_SLEEP) != NDI_SUCCESS) {
                        goto fail;
                }

                tran->tran_get_name             = sf_scsi_get_name;
                tran->tran_get_bus_addr         = sf_scsi_get_bus_addr;

                /* setup and attach SCSI hba transport */
                if (scsi_hba_attach_setup(dip, sf->sf_sochandle->
                    fcal_dmaattr, tran, SCSI_HBA_TRAN_CLONE) != DDI_SUCCESS) {
                        cmn_err(CE_WARN, "sf%d: scsi_hba_attach_setup failed",
                            instance);
                        goto fail;
                }

                /* set up kstats */
                if ((sf->sf_ksp = kstat_create("sf", instance, "statistics",
                    "controller", KSTAT_TYPE_RAW, sizeof (struct sf_stats),
                    KSTAT_FLAG_VIRTUAL)) == NULL) {
                        cmn_err(CE_WARN, "sf%d: failed to create kstat",
                            instance);
                } else {
                        sf->sf_stats.version = 2;
                        (void) sprintf(sf->sf_stats.drvr_name,
                        "%s: %s", SF_NAME, sf_version);
                        sf->sf_ksp->ks_data = (void *)&sf->sf_stats;
                        sf->sf_ksp->ks_private = sf;
                        sf->sf_ksp->ks_update = sf_kstat_update;
                        kstat_install(sf->sf_ksp);
                }

                /* create the hotplug thread */
                mutex_enter(&sf->sf_hp_daemon_mutex);
                tp = thread_create(NULL, 0,
                    (void (*)())sf_hp_daemon, sf, 0, &p0, TS_RUN, minclsyspri);
                sf->sf_hp_tid = tp->t_did;
                mutex_exit(&sf->sf_hp_daemon_mutex);

                /* add this soft state instance to the head of the list */
                mutex_enter(&sf_global_mutex);
                sf->sf_next = sf_head;
                tsf = sf_head;
                sf_head = sf;

                /*
                 * find entry in list that has the same FC-AL handle (if any)
                 */
                while (tsf != NULL) {
                        if (tsf->sf_socp == sf->sf_socp) {
                                break;          /* found matching entry */
                        }
                        tsf = tsf->sf_next;
                }

                if (tsf != NULL) {
                        /* if we found a matching entry keep track of it */
                        sf->sf_sibling = tsf;
                }

                /*
                 * increment watchdog init flag, setting watchdog timeout
                 * if we are the first (since somebody has to do it)
```

```
		 */
		if (!sf_watchdog_init++) {
			mutex_exit(&sf_global_mutex);
			sf_watchdog_tick = sf_watchdog_timeout *
			    drv_usectohz(1000000);
			sf_watchdog_id = timeout(sf_watch,
			    NULL, sf_watchdog_tick);
		} else {
			mutex_exit(&sf_global_mutex);
		}

		if (tsf != NULL) {
			/*
			 * set up matching entry to be our sibling
			 */
			mutex_enter(&tsf->sf_mutex);
			tsf->sf_sibling = sf;
			mutex_exit(&tsf->sf_mutex);
		}

		/*
		 * create this property so that PM code knows we want
		 * to be suspended at PM time
		 */
		(void) ddi_prop_update_string(DDI_DEV_T_NONE, dip,
		    PM_HARDWARE_STATE_PROP, PM_NEEDS_SUSPEND_RESUME);

		/* log the fact that we have a new device */
		ddi_report_dev(dip);

		/*
		 * force a login by setting our state to offline
		 */
		sf->sf_timer = sf_watchdog_time + SF_OFFLINE_TIMEOUT;
		sf->sf_state = SF_STATE_OFFLINE;

		/*
		 * call transport routine to register state change and
		 * ELS callback routines (to register us as a ULP)
		 */
		soc_add_ulp(sf->sf_sochandle, sf->sf_socp,
		    sf->sf_sochandle->fcal_portno, TYPE_SCSI_FCP,
		    sf_statec_callback, sf_unsol_els_callback, NULL, sf);

		/*
		 * call transport routine to force loop initialization
		 */
		(void) soc_force_lip(sf->sf_sochandle, sf->sf_socp,
		    sf->sf_sochandle->fcal_portno, 0, FCAL_NO_LIP);
		sf->sf_reset_time = ddi_get_lbolt64();
		return (DDI_SUCCESS);

	default:
		return (DDI_FAILURE);
	}

fail:
	cmn_err(CE_WARN, "sf%d: failed to attach", instance);

	/*
	 * Unbind and free event set
	 */
	if (sf->sf_event_hdl) {
		(void) ndi_event_unbind_set(sf->sf_event_hdl,
		    &sf->sf_events, NDI_SLEEP);
		(void) ndi_event_free_hdl(sf->sf_event_hdl);
```

```
	}

	if (sf->sf_event_defs) {
		kmem_free(sf->sf_event_defs, sizeof (sf_event_defs));
	}

	if (sf->sf_tran != NULL) {
		scsi_hba_tran_free(sf->sf_tran);
	}
	while (sf->sf_cr_pool != NULL) {
		sf_crpool_free(sf);
	}
	if (sf->sf_lilp_dmahandle != NULL) {
		if (handle_bound) {
			(void) ddi_dma_unbind_handle(sf->sf_lilp_dmahandle);
		}
		ddi_dma_free_handle(&sf->sf_lilp_dmahandle);
	}
	if (sf->sf_pkt_cache != NULL) {
		kmem_cache_destroy(sf->sf_pkt_cache);
	}
	if (sf->sf_lilp_map != NULL) {
		ddi_dma_mem_free(&sf->sf_lilp_acchandle);
	}
	if (sf->sf_ksp != NULL) {
		kstat_delete(sf->sf_ksp);
	}
	if (mutex_initted) {
		mutex_destroy(&sf->sf_mutex);
		mutex_destroy(&sf->sf_cmd_mutex);
		mutex_destroy(&sf->sf_cr_mutex);
		mutex_destroy(&sf->sf_hp_daemon_mutex);
		cv_destroy(&sf->sf_cr_cv);
		cv_destroy(&sf->sf_hp_daemon_cv);
	}
	mutex_enter(&sf_global_mutex);

	/*
	 * kill off the watchdog if we are the last instance
	 */
	if (!--sf_watchdog_init) {
		timeout_id_t tid = sf_watchdog_id;
		mutex_exit(&sf_global_mutex);
		(void) untimeout(tid);
	} else {
		mutex_exit(&sf_global_mutex);
	}

	ddi_soft_state_free(sf_state, instance);

	if (tran != NULL) {
		/* remove all minor nodes */
		ddi_remove_minor_node(dip, NULL);
	}

	return (DDI_FAILURE);
}


/* ARGSUSED */
static int
sf_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
	struct sf		*sf;
	int			instance;
	int			i;
	struct sf_target	*target;
	timeout_id_t		tid;



	/* NO OTHER THREADS ARE RUNNING */

	instance = ddi_get_instance(dip);

	if ((sf = ddi_get_soft_state(sf_state, instance)) == NULL) {
		cmn_err(CE_WARN, "sf_detach, sf%d: bad soft state", instance);
		return (DDI_FAILURE);
	}

	switch (cmd) {

	case DDI_SUSPEND:
		/*
		 * suspend our instance
		 */

		SF_DEBUG(2, (sf, CE_CONT,
		    "sf_detach: DDI_SUSPEND for sf%d\n", instance));
		/*
		 * There is a race condition in socal where while doing
		 * callbacks if a ULP removes it self from the callback list
		 * the for loop in socal may panic as cblist is junk and
		 * while trying to get cblist->next the system will panic.
		 */

		/* call transport to remove our unregister our callbacks */
		soc_remove_ulp(sf->sf_sochandle, sf->sf_socp,
		    sf->sf_sochandle->fcal_portno, TYPE_SCSI_FCP, sf);

		/*
		 * begin process of clearing outstanding commands
		 * by issuing a lip
		 */
		sf_force_lip(sf);

		/*
		 * toggle the device OFFLINE in order to cause
		 * outstanding commands to drain
		 */
		mutex_enter(&sf->sf_mutex);
		sf->sf_lip_cnt++;
		sf->sf_timer = sf_watchdog_time + SF_OFFLINE_TIMEOUT;
		sf->sf_state = (SF_STATE_OFFLINE | SF_STATE_SUSPENDED);
		for (i = 0; i < sf_max_targets; i++) {
			target = sf->sf_targets[i];
			if (target != NULL) {
				struct sf_target *ntarget;

				mutex_enter(&target->sft_mutex);
				if (!(target->sft_state & SF_TARGET_OFFLINE)) {
					target->sft_state |=
					    (SF_TARGET_BUSY | SF_TARGET_MARK);
				}
				/* do this for all LUNs as well */
				for (ntarget = target->sft_next_lun;
				    ntarget;
				    ntarget = ntarget->sft_next_lun) {
					mutex_enter(&ntarget->sft_mutex);
					if (!(ntarget->sft_state &
					    SF_TARGET_OFFLINE)) {
						ntarget->sft_state |=
```

```
						    (SF_TARGET_BUSY |
						    SF_TARGET_MARK);
					}
					mutex_exit(&ntarget->sft_mutex);
				}
				mutex_exit(&target->sft_mutex);
			}
		}
		mutex_exit(&sf->sf_mutex);
		mutex_enter(&sf_global_mutex);

		/*
		 * kill off the watchdog if we are the last instance
		 */
		if (!--sf_watchdog_init) {
			tid = sf_watchdog_id;
			mutex_exit(&sf_global_mutex);
			(void) untimeout(tid);
		} else {
			mutex_exit(&sf_global_mutex);
		}

		return (DDI_SUCCESS);

	case DDI_DETACH:
		/*
		 * detach this instance
		 */

		SF_DEBUG(2, (sf, CE_CONT,
		    "sf_detach: DDI_DETACH for sf%d\n", instance));

		/* remove this "sf" from the list of sf softstates */
		sf_softstate_unlink(sf);

		/*
		 * prior to taking any DDI_DETACH actions, toggle the
		 * device OFFLINE in order to cause outstanding
		 * commands to drain
		 */
		mutex_enter(&sf->sf_mutex);
		sf->sf_lip_cnt++;
		sf->sf_timer = sf_watchdog_time + SF_OFFLINE_TIMEOUT;
		sf->sf_state = SF_STATE_OFFLINE;
		for (i = 0; i < sf_max_targets; i++) {
			target = sf->sf_targets[i];
			if (target != NULL) {
				struct sf_target *ntarget;

				mutex_enter(&target->sft_mutex);
				if (!(target->sft_state & SF_TARGET_OFFLINE)) {
					target->sft_state |=
					    (SF_TARGET_BUSY | SF_TARGET_MARK);
				}
				for (ntarget = target->sft_next_lun;
				    ntarget;
				    ntarget = ntarget->sft_next_lun) {
					mutex_enter(&ntarget->sft_mutex);
					if (!(ntarget->sft_state &
					    SF_TARGET_OFFLINE)) {
						ntarget->sft_state |=
						    (SF_TARGET_BUSY |
						    SF_TARGET_MARK);
					}
					mutex_exit(&ntarget->sft_mutex);
				}
```

```
                                mutex_exit(&target->sft_mutex);
                        }
                }
                mutex_exit(&sf->sf_mutex);

                /* call transport to remove and unregister our callbacks */
                soc_remove_ulp(sf->sf_sochandle, sf->sf_socp,
                    sf->sf_sochandle->fcal_portno, TYPE_SCSI_FCP, sf);

                /*
                 * kill off the watchdog if we are the last instance
                 */
                mutex_enter(&sf_global_mutex);
                if (!--sf_watchdog_init) {
                        tid = sf_watchdog_id;
                        mutex_exit(&sf_global_mutex);
                        (void) untimeout(tid);
                } else {
                        mutex_exit(&sf_global_mutex);
                }

                /* signal sf_hp_daemon() to exit and wait for exit */
                mutex_enter(&sf->sf_hp_daemon_mutex);
                ASSERT(sf->sf_hp_tid);
                sf->sf_hp_exit = 1;             /* flag exit */
                cv_signal(&sf->sf_hp_daemon_cv);
                mutex_exit(&sf->sf_hp_daemon_mutex);
                thread_join(sf->sf_hp_tid);     /* wait for hotplug to exit */

                /*
                 * Unbind and free event set
                 */
                if (sf->sf_event_hdl) {
                        (void) ndi_event_unbind_set(sf->sf_event_hdl,
                            &sf->sf_events, NDI_SLEEP);
                        (void) ndi_event_free_hdl(sf->sf_event_hdl);
                }

                if (sf->sf_event_defs) {
                        kmem_free(sf->sf_event_defs, sizeof (sf_event_defs));
                }

                /* detach this instance of the HBA driver */
                (void) scsi_hba_detach(dip);
                scsi_hba_tran_free(sf->sf_tran);

                /* deallocate/unbind DMA handle for lilp map */
                if (sf->sf_lilp_map != NULL) {
                        (void) ddi_dma_unbind_handle(sf->sf_lilp_dmahandle);
                        if (sf->sf_lilp_dmahandle != NULL) {
                                ddi_dma_free_handle(&sf->sf_lilp_dmahandle);
                        }
                        ddi_dma_mem_free(&sf->sf_lilp_acchandle);
                }

                /*
                 * the kmem cache must be destroyed before free'ing
                 * up the crpools
                 *
                 * our finagle of "ntot" and "nfree"
                 * causes an ASSERT failure in "sf_cr_free()"
                 * if the kmem cache is free'd after invoking
                 * "sf_crpool_free()".
                 */
                kmem_cache_destroy(sf->sf_pkt_cache);
```

```
                SF_DEBUG(2, (sf, CE_CONT,
                    "sf_detach: sf_crpool_free() for instance 0x%x\n",
                    instance));
                while (sf->sf_cr_pool != NULL) {
                        /*
                         * set ntot to nfree for this particular entry
                         *
                         * this causes sf_crpool_free() to update
                         * the cr_pool list when deallocating this entry
                         */
                        sf->sf_cr_pool->ntot = sf->sf_cr_pool->nfree;
                        sf_crpool_free(sf);
                }

                /*
                 * now that the cr_pool's are gone it's safe
                 * to destroy all softstate mutex's and cv's
                 */
                mutex_destroy(&sf->sf_mutex);
                mutex_destroy(&sf->sf_cmd_mutex);
                mutex_destroy(&sf->sf_cr_mutex);
                mutex_destroy(&sf->sf_hp_daemon_mutex);
                cv_destroy(&sf->sf_cr_cv);
                cv_destroy(&sf->sf_hp_daemon_cv);

                /* remove all minor nodes from the device tree */
                ddi_remove_minor_node(dip, NULL);

                /* remove properties created during attach() */
                ddi_prop_remove_all(dip);

                /* remove kstat's if present */
                if (sf->sf_ksp != NULL) {
                        kstat_delete(sf->sf_ksp);
                }

                SF_DEBUG(2, (sf, CE_CONT,
                    "sf_detach: ddi_soft_state_free() for instance 0x%x\n",
                    instance));
                ddi_soft_state_free(sf_state, instance);
                return (DDI_SUCCESS);

        default:
                SF_DEBUG(2, (sf, CE_CONT, "sf_detach: sf%d unknown cmd %x\n",
                    instance, (int)cmd));
                return (DDI_FAILURE);
        }
}


/*
 * sf_softstate_unlink() - remove an sf instance from the list of softstates
 */
static void
sf_softstate_unlink(struct sf *sf)
{
        struct sf       *sf_ptr;
        struct sf       *sf_found_sibling;
        struct sf       *sf_reposition = NULL;


        mutex_enter(&sf_global_mutex);
        while (sf_watch_running) {
                /* Busy working the list -- wait */
                cv_wait(&sf_watch_cv, &sf_global_mutex);
        }
	if ((sf_found_sibling = sf->sf_sibling) != NULL) {
		/*
		 * we have a sibling so NULL out its reference to us
		 */
		mutex_enter(&sf_found_sibling->sf_mutex);
		sf_found_sibling->sf_sibling = NULL;
		mutex_exit(&sf_found_sibling->sf_mutex);
	}

	/* remove our instance from the global list */
	if (sf == sf_head) {
		/* we were at at head of the list */
		sf_head = sf->sf_next;
	} else {
		/* find us in the list */
		for (sf_ptr = sf_head;
		    sf_ptr != NULL;
		    sf_ptr = sf_ptr->sf_next) {
			if (sf_ptr == sf) {
				break;
			}
			/* remember this place */
			sf_reposition = sf_ptr;
		}
		ASSERT(sf_ptr == sf);
		ASSERT(sf_reposition != NULL);

		sf_reposition->sf_next = sf_ptr->sf_next;
	}
	mutex_exit(&sf_global_mutex);
}


static int
sf_scsi_bus_config(dev_info_t *parent, uint_t flag,
    ddi_bus_config_op_t op, void *arg, dev_info_t **childp)
{
	int64_t		reset_delay;
	struct sf	*sf;

	sf = ddi_get_soft_state(sf_state, ddi_get_instance(parent));
	ASSERT(sf);

	reset_delay = (int64_t)(USEC_TO_TICK(SF_INIT_WAIT_TIMEOUT)) -
	    (ddi_get_lbolt64() - sf->sf_reset_time);
	if (reset_delay < 0)
		reset_delay = 0;

	if (sf_bus_config_debug)
		flag |= NDI_DEVI_DEBUG;

	return (ndi_busop_bus_config(parent, flag, op,
	    arg, childp, (clock_t)reset_delay));
}

static int
sf_scsi_bus_unconfig(dev_info_t *parent, uint_t flag,
    ddi_bus_config_op_t op, void *arg)
{
	if (sf_bus_config_debug)
		flag |= NDI_DEVI_DEBUG;

	return (ndi_busop_bus_unconfig(parent, flag, op, arg));
}
```

```
/*
 * called by transport to initialize a SCSI target
 */
/* ARGSUSED */
static int
sf_scsi_tgt_init(dev_info_t *hba_dip, dev_info_t *tgt_dip,
    scsi_hba_tran_t *hba_tran, struct scsi_device *sd)
{
#ifdef RAID_LUNS
	int lun;
#else
	int64_t lun;
#endif
	struct sf_target *target;
	struct sf *sf = (struct sf *)hba_tran->tran_hba_private;
	int i, t_len;
	unsigned int lip_cnt;
	unsigned char wwn[FC_WWN_SIZE];


	/* get and validate our SCSI target ID */
	i = sd->sd_address.a_target;
	if (i >= sf_max_targets) {
		return (DDI_NOT_WELL_FORMED);
	}

	/* get our port WWN property */
	t_len = sizeof (wwn);
	if (ddi_prop_op(DDI_DEV_T_ANY, tgt_dip, PROP_LEN_AND_VAL_BUF,
	    DDI_PROP_DONTPASS | DDI_PROP_CANSLEEP, PORT_WWN_PROP,
	    (caddr_t)&wwn, &t_len) != DDI_SUCCESS) {
		/* no port WWN property - ignore the OBP stub node */
		return (DDI_NOT_WELL_FORMED);
	}

	/* get our LIP count property */
	t_len = sizeof (lip_cnt);
	if (ddi_prop_op(DDI_DEV_T_ANY, tgt_dip, PROP_LEN_AND_VAL_BUF,
	    DDI_PROP_DONTPASS | DDI_PROP_CANSLEEP, LIP_CNT_PROP,
	    (caddr_t)&lip_cnt, &t_len) != DDI_SUCCESS) {
		return (DDI_FAILURE);
	}
	/* and our LUN property */
	t_len = sizeof (lun);
	if (ddi_prop_op(DDI_DEV_T_ANY, tgt_dip, PROP_LEN_AND_VAL_BUF,
	    DDI_PROP_DONTPASS | DDI_PROP_CANSLEEP, "lun",
	    (caddr_t)&lun, &t_len) != DDI_SUCCESS) {
		return (DDI_FAILURE);
	}

	/* find the target structure for this instance */
	mutex_enter(&sf->sf_mutex);
	if ((target = sf_lookup_target(sf, wwn, lun)) == NULL) {
		mutex_exit(&sf->sf_mutex);
		return (DDI_FAILURE);
	}

	mutex_enter(&target->sft_mutex);
	if ((sf->sf_lip_cnt == lip_cnt) && !(target->sft_state
	    & SF_TARGET_INIT_DONE)) {
		/*
		 * set links between HBA transport and target structures
		 * and set done flag
		 */
		hba_tran->tran_tgt_private = target;
		target->sft_tran = hba_tran;
		target->sft_state |= SF_TARGET_INIT_DONE;
	} else {
		/* already initialized ?? */
		mutex_exit(&target->sft_mutex);
		mutex_exit(&sf->sf_mutex);
		return (DDI_FAILURE);
	}
	mutex_exit(&target->sft_mutex);
	mutex_exit(&sf->sf_mutex);

	return (DDI_SUCCESS);
}


/*
 * called by transport to free a target
 */
/* ARGSUSED */
static void
sf_scsi_tgt_free(dev_info_t *hba_dip, dev_info_t *tgt_dip,
    scsi_hba_tran_t *hba_tran, struct scsi_device *sd)
{
	struct sf_target *target = hba_tran->tran_tgt_private;

	if (target != NULL) {
		mutex_enter(&target->sft_mutex);
		target->sft_tran = NULL;
		target->sft_state &= ~SF_TARGET_INIT_DONE;
		mutex_exit(&target->sft_mutex);
	}
}


/*
 * allocator for non-std size cdb/pkt_private/status -- return TRUE iff
 * success, else return FALSE
 */
/*ARGSUSED*/
static int
sf_pkt_alloc_extern(struct sf *sf, struct sf_pkt *cmd,
    int tgtlen, int statuslen, int kf)
{
	caddr_t scbp, tgt;
	int failure = FALSE;
	struct scsi_pkt *pkt = CMD2PKT(cmd);


	tgt = scbp = NULL;

	if (tgtlen > PKT_PRIV_LEN) {
		if ((tgt = kmem_zalloc(tgtlen, kf)) == NULL) {
			failure = TRUE;
		} else {
			cmd->cmd_flags |= CFLAG_PRIVEXTERN;
			pkt->pkt_private = tgt;
		}
	}
	if (statuslen > EXTCMDS_STATUS_SIZE) {
		if ((scbp = kmem_zalloc((size_t)statuslen, kf)) == NULL) {
			failure = TRUE;
		} else {
			cmd->cmd_flags |= CFLAG_SCBEXTERN;
			pkt->pkt_scbp = (opaque_t)scbp;
		}
	}
	if (failure) {
```

```
		sf_pkt_destroy_extern(sf, cmd);
	}
	return (failure);
}


/*
 * deallocator for non-std size cdb/pkt_private/status
 */
static void
sf_pkt_destroy_extern(struct sf *sf, struct sf_pkt *cmd)
{
	struct scsi_pkt *pkt = CMD2PKT(cmd);

	if (cmd->cmd_flags & CFLAG_FREE) {
		cmn_err(CE_PANIC,
		    "sf_scsi_impl_pktfree: freeing free packet");
		_NOTE(NOT_REACHED)
		/* NOTREACHED */
	}
	if (cmd->cmd_flags & CFLAG_SCBEXTERN) {
		kmem_free((caddr_t)pkt->pkt_scbp,
		    (size_t)cmd->cmd_scblen);
	}
	if (cmd->cmd_flags & CFLAG_PRIVEXTERN) {
		kmem_free((caddr_t)pkt->pkt_private,
		    (size_t)cmd->cmd_privlen);
	}

	cmd->cmd_flags = CFLAG_FREE;
	kmem_cache_free(sf->sf_pkt_cache, (void *)cmd);
}


/*
 * create or initialize a SCSI packet -- called internally and
 * by the transport
 */
static struct scsi_pkt *
sf_scsi_init_pkt(struct scsi_address *ap, struct scsi_pkt *pkt,
    struct buf *bp, int cmdlen, int statuslen, int tgtlen,
    int flags, int (*callback)(), caddr_t arg)
{
	int kf;
	int failure = FALSE;
	struct sf_pkt *cmd;
	struct sf *sf = ADDR2SF(ap);
	struct sf_target *target = ADDR2TARGET(ap);
	struct sf_pkt	*new_cmd = NULL;
	struct fcal_packet	*fpkt;
	fc_frame_header_t	*hp;
	struct fcp_cmd *fcmd;


	/*
	 * If we've already allocated a pkt once,
	 * this request is for dma allocation only.
	 */
	if (pkt == NULL) {

		/*
		 * First step of sf_scsi_init_pkt:  pkt allocation
		 */
		if (cmdlen > FCP_CDB_SIZE) {
			return (NULL);
		}
		kf = (callback == SLEEP_FUNC)? KM_SLEEP: KM_NOSLEEP;

		if ((cmd = kmem_cache_alloc(sf->sf_pkt_cache, kf)) != NULL) {
			/*
			 * Selective zeroing of the pkt.
			 */

			cmd->cmd_flags = 0;
			cmd->cmd_forw = 0;
			cmd->cmd_back = 0;
			cmd->cmd_next = 0;
			cmd->cmd_pkt = (struct scsi_pkt *)((char *)cmd +
			    sizeof (struct sf_pkt) + sizeof (struct
			    fcal_packet));
			cmd->cmd_fp_pkt = (struct fcal_packet *)((char *)cmd +
			    sizeof (struct sf_pkt));
			cmd->cmd_fp_pkt->fcal_pkt_private = (opaque_t)cmd;
			cmd->cmd_state = SF_STATE_IDLE;
			cmd->cmd_pkt->pkt_ha_private = (opaque_t)cmd;
			cmd->cmd_pkt->pkt_scbp = (opaque_t)cmd->cmd_scsi_scb;
			cmd->cmd_pkt->pkt_comp  = NULL;
			cmd->cmd_pkt->pkt_flags = 0;
			cmd->cmd_pkt->pkt_time  = 0;
			cmd->cmd_pkt->pkt_resid = 0;
			cmd->cmd_pkt->pkt_reason = 0;
			cmd->cmd_cdblen = (uchar_t)cmdlen;
			cmd->cmd_scblen		= statuslen;
			cmd->cmd_privlen	= tgtlen;
			cmd->cmd_pkt->pkt_address = *ap;

			/* zero pkt_private */
			(int *)(cmd->cmd_pkt->pkt_private =
			    cmd->cmd_pkt_private);
			bzero((caddr_t)cmd->cmd_pkt->pkt_private,
			    PKT_PRIV_LEN);
		} else {
			failure = TRUE;
		}

		if (failure ||
		    (tgtlen > PKT_PRIV_LEN) ||
		    (statuslen > EXTCMDS_STATUS_SIZE)) {
			if (!failure) {
				/* need to allocate more space */
				failure = sf_pkt_alloc_extern(sf, cmd,
				    tgtlen, statuslen, kf);
			}
			if (failure) {
				return (NULL);
			}
		}

		fpkt = cmd->cmd_fp_pkt;
		if (cmd->cmd_block == NULL) {

			/* allocate cmd/response pool buffers */
			if (sf_cr_alloc(sf, cmd, callback) == DDI_FAILURE) {
				sf_pkt_destroy_extern(sf, cmd);
				return (NULL);
			}

			/* fill in the FC-AL packet */
			fpkt->fcal_pkt_cookie = sf->sf_socp;
			fpkt->fcal_pkt_comp = sf_cmd_callback;
			fpkt->fcal_pkt_flags = 0;
```

```
			fpkt->fcal_magic = FCALP_MAGIC;
			fpkt->fcal_socal_request.sr_soc_hdr.sh_flags =
			    (ushort_t)(SOC_FC_HEADER |
			    sf->sf_sochandle->fcal_portno);
			fpkt->fcal_socal_request.sr_soc_hdr.sh_class = 3;
			fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_count = 1;
			fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_flags = 0;
			fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_seqno = 0;
			fpkt->fcal_socal_request.sr_dataseg[0].fc_base =
			    (uint32_t)cmd->cmd_dmac;
			fpkt->fcal_socal_request.sr_dataseg[0].fc_count =
			    sizeof (struct fcp_cmd);
			fpkt->fcal_socal_request.sr_dataseg[1].fc_base =
			    (uint32_t)cmd->cmd_rsp_dmac;
			fpkt->fcal_socal_request.sr_dataseg[1].fc_count =
			    FCP_MAX_RSP_IU_SIZE;

			/* Fill in the Fabric Channel Header */
			hp = &fpkt->fcal_socal_request.sr_fc_frame_hdr;
			hp->r_ctl = R_CTL_COMMAND;
			hp->type = TYPE_SCSI_FCP;
			hp->f_ctl = F_CTL_SEQ_INITIATIVE | F_CTL_FIRST_SEQ;
			hp->reserved1 = 0;
			hp->seq_id = 0;
			hp->df_ctl  = 0;
			hp->seq_cnt = 0;
			hp->ox_id = 0xffff;
			hp->rx_id = 0xffff;
			hp->ro = 0;

			/* Establish the LUN */
			bcopy((caddr_t)&target->sft_lun.b,
			    (caddr_t)&cmd->cmd_block->fcp_ent_addr,
			    FCP_LUN_SIZE);
			*((int32_t *)&cmd->cmd_block->fcp_cntl) = 0;
		}
		cmd->cmd_pkt->pkt_cdbp = cmd->cmd_block->fcp_cdb;

		mutex_enter(&target->sft_pkt_mutex);

		target->sft_pkt_tail->cmd_forw = cmd;
		cmd->cmd_back = target->sft_pkt_tail;
		cmd->cmd_forw = (struct sf_pkt *)&target->sft_pkt_head;
		target->sft_pkt_tail = cmd;

		mutex_exit(&target->sft_pkt_mutex);
		new_cmd = cmd;		/* for later cleanup if needed */
	} else {
		/* pkt already exists -- just a request for DMA allocation */
		cmd = PKT2CMD(pkt);
		fpkt = cmd->cmd_fp_pkt;
	}

	/* zero cdb (bzero is too slow) */
	bzero((caddr_t)cmd->cmd_pkt->pkt_cdbp, cmdlen);

	/*
	 * Second step of sf_scsi_init_pkt:  dma allocation
	 * Set up dma info
	 */
	if ((bp != NULL) && (bp->b_bcount != 0)) {
		int cmd_flags, dma_flags;
		int rval = 0;
		uint_t dmacookie_count;

		/* there is a buffer and some data to transfer */
		/* set up command and DMA flags */
		cmd_flags = cmd->cmd_flags;
		if (bp->b_flags & B_READ) {
			/* a read */
			cmd_flags &= ~CFLAG_DMASEND;
			dma_flags = DDI_DMA_READ;
		} else {
			/* a write */
			cmd_flags |= CFLAG_DMASEND;
			dma_flags = DDI_DMA_WRITE;
		}
		if (flags & PKT_CONSISTENT) {
			cmd_flags |= CFLAG_CMDIOPB;
			dma_flags |= DDI_DMA_CONSISTENT;
		}

		/* ensure we have a DMA handle */
		if (cmd->cmd_dmahandle == NULL) {
			rval = ddi_dma_alloc_handle(sf->sf_dip,
			    sf->sf_sochandle->fcal_dmaattr, callback, arg,
			    &cmd->cmd_dmahandle);
		}

		if (rval == 0) {
			/* bind our DMA handle to our buffer */
			rval = ddi_dma_buf_bind_handle(cmd->cmd_dmahandle, bp,
			    dma_flags, callback, arg, &cmd->cmd_dmacookie,
			    &dmacookie_count);
		}

		if (rval != 0) {
			/* DMA failure */
			SF_DEBUG(2, (sf, CE_CONT, "ddi_dma_buf.. failed\n"));
			switch (rval) {
			case DDI_DMA_NORESOURCES:
				bioerror(bp, 0);
				break;
			case DDI_DMA_BADATTR:
			case DDI_DMA_NOMAPPING:
				bioerror(bp, EFAULT);
				break;
			case DDI_DMA_TOOBIG:
			default:
				bioerror(bp, EINVAL);
				break;
			}
			/* clear valid flag */
			cmd->cmd_flags = cmd_flags & ~CFLAG_DMAVALID;
			if (new_cmd != NULL) {
				/* destroy packet if we just created it */
				sf_scsi_destroy_pkt(ap, new_cmd->cmd_pkt);
			}
			return (NULL);
		}

		ASSERT(dmacookie_count == 1);
		/* set up amt to transfer and set valid flag */
		cmd->cmd_dmacount = bp->b_bcount;
		cmd->cmd_flags = cmd_flags | CFLAG_DMAVALID;

		ASSERT(cmd->cmd_dmahandle != NULL);
	}

	/* set up FC-AL packet */
	fcmd = cmd->cmd_block;
```

```
	if (cmd->cmd_flags & CFLAG_DMAVALID) {
		if (cmd->cmd_flags & CFLAG_DMASEND) {
			/* DMA write */
			fcmd->fcp_cntl.cntl_read_data = 0;
			fcmd->fcp_cntl.cntl_write_data = 1;
			fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_type =
			    CQ_TYPE_IO_WRITE;
		} else {
			/* DMA read */
			fcmd->fcp_cntl.cntl_read_data = 1;
			fcmd->fcp_cntl.cntl_write_data = 0;
			fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_type =
			    CQ_TYPE_IO_READ;
		}
		fpkt->fcal_socal_request.sr_dataseg[2].fc_base =
		    (uint32_t)cmd->cmd_dmacookie.dmac_address;
		fpkt->fcal_socal_request.sr_dataseg[2].fc_count =
		    cmd->cmd_dmacookie.dmac_size;
		fpkt->fcal_socal_request.sr_soc_hdr.sh_seg_cnt = 3;
		fpkt->fcal_socal_request.sr_soc_hdr.sh_byte_cnt =
		    cmd->cmd_dmacookie.dmac_size;
		fcmd->fcp_data_len = cmd->cmd_dmacookie.dmac_size;
	} else {
		/* not a read or write */
		fcmd->fcp_cntl.cntl_read_data = 0;
		fcmd->fcp_cntl.cntl_write_data = 0;
		fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_type = CQ_TYPE_SIMPLE;
		fpkt->fcal_socal_request.sr_soc_hdr.sh_seg_cnt = 2;
		fpkt->fcal_socal_request.sr_soc_hdr.sh_byte_cnt =
		    sizeof (struct fcp_cmd);
		fcmd->fcp_data_len = 0;
	}
	fcmd->fcp_cntl.cntl_qtype = FCP_QTYPE_SIMPLE;

	return (cmd->cmd_pkt);
}


/*
 * destroy a SCSI packet -- called internally and by the transport
 */
static void
sf_scsi_destroy_pkt(struct scsi_address *ap, struct scsi_pkt *pkt)
{
	struct sf_pkt *cmd = PKT2CMD(pkt);
	struct sf *sf = ADDR2SF(ap);
	struct sf_target *target = ADDR2TARGET(ap);
	struct fcal_packet	*fpkt = cmd->cmd_fp_pkt;


	if (cmd->cmd_flags & CFLAG_DMAVALID) {
		/* DMA was set up -- clean up */
		(void) ddi_dma_unbind_handle(cmd->cmd_dmahandle);
		cmd->cmd_flags ^= CFLAG_DMAVALID;
	}

	/* take this packet off the doubly-linked list */
	mutex_enter(&target->sft_pkt_mutex);
	cmd->cmd_back->cmd_forw = cmd->cmd_forw;
	cmd->cmd_forw->cmd_back = cmd->cmd_back;
	mutex_exit(&target->sft_pkt_mutex);

	fpkt->fcal_pkt_flags = 0;
	/* free the packet */
	if ((cmd->cmd_flags &
	    (CFLAG_FREE | CFLAG_PRIVEXTERN | CFLAG_SCBEXTERN)) == 0) {
		/* just a regular packet */
		ASSERT(cmd->cmd_state != SF_STATE_ISSUED);
		cmd->cmd_flags = CFLAG_FREE;
		kmem_cache_free(sf->sf_pkt_cache, (void *)cmd);
	} else {
		/* a packet with extra memory */
		sf_pkt_destroy_extern(sf, cmd);
	}
}


/*
 * called by transport to unbind DMA handle
 */
/* ARGSUSED */
static void
sf_scsi_dmafree(struct scsi_address *ap, struct scsi_pkt *pkt)
{
	struct sf_pkt *cmd = PKT2CMD(pkt);


	if (cmd->cmd_flags & CFLAG_DMAVALID) {
		(void) ddi_dma_unbind_handle(cmd->cmd_dmahandle);
		cmd->cmd_flags ^= CFLAG_DMAVALID;
	}

}


/*
 * called by transport to synchronize CPU and I/O views of memory
 */
/* ARGSUSED */
static void
sf_scsi_sync_pkt(struct scsi_address *ap, struct scsi_pkt *pkt)
{
	struct sf_pkt *cmd = PKT2CMD(pkt);


	if (cmd->cmd_flags & CFLAG_DMAVALID) {
		if (ddi_dma_sync(cmd->cmd_dmahandle, (off_t)0, (size_t)0,
		    (cmd->cmd_flags & CFLAG_DMASEND) ?
		    DDI_DMA_SYNC_FORDEV : DDI_DMA_SYNC_FORCPU) !=
		    DDI_SUCCESS) {
			cmn_err(CE_WARN, "sf: sync pkt failed");
		}
	}
}


/*
 * routine for reset notification setup, to register or cancel. -- called
 * by transport
 */
static int
sf_scsi_reset_notify(struct scsi_address *ap, int flag,
    void (*callback)(caddr_t), caddr_t arg)
{
	struct sf	*sf = ADDR2SF(ap);

	return (scsi_hba_reset_notify_setup(ap, flag, callback, arg,
	    &sf->sf_mutex, &sf->sf_reset_notify_listf));
}
```

```
/*
 * called by transport to get port WWN property (except sun4u)
 */
/* ARGSUSED */
static int
sf_scsi_get_name(struct scsi_device *sd, char *name, int len)
{
	char tbuf[(FC_WWN_SIZE*2)+1];
	unsigned char wwn[FC_WWN_SIZE];
	int i, lun;
	dev_info_t *tgt_dip;

	tgt_dip = sd->sd_dev;
	i = sizeof (wwn);
	if (ddi_prop_op(DDI_DEV_T_ANY, tgt_dip, PROP_LEN_AND_VAL_BUF,
	    DDI_PROP_DONTPASS | DDI_PROP_CANSLEEP, PORT_WWN_PROP,
	    (caddr_t)&wwn, &i) != DDI_SUCCESS) {
		name[0] = '\0';
		return (0);
	}
	i = sizeof (lun);
	if (ddi_prop_op(DDI_DEV_T_ANY, tgt_dip, PROP_LEN_AND_VAL_BUF,
	    DDI_PROP_DONTPASS | DDI_PROP_CANSLEEP, "lun",
	    (caddr_t)&lun, &i) != DDI_SUCCESS) {
		name[0] = '\0';
		return (0);
	}
	for (i = 0; i < FC_WWN_SIZE; i++)
		(void) sprintf(&tbuf[i << 1], "%02x", wwn[i]);
	(void) sprintf(name, "w%s,%x", tbuf, lun);
	return (1);
}


/*
 * called by transport to get target soft AL-PA (except sun4u)
 */
/* ARGSUSED */
static int
sf_scsi_get_bus_addr(struct scsi_device *sd, char *name, int len)
{
	struct sf_target *target = ADDR2TARGET(&sd->sd_address);

	if (target == NULL)
		return (0);

	(void) sprintf(name, "%x", target->sft_al_pa);
	return (1);
}


/*
 * add to the command/response buffer pool for this sf instance
 */
static int
sf_add_cr_pool(struct sf *sf)
{
	int		cmd_buf_size;
	size_t		real_cmd_buf_size;
	int		rsp_buf_size;
	size_t		real_rsp_buf_size;
	uint_t		i, ccount;
	struct sf_cr_pool	*ptr;
	struct sf_cr_free_elem *cptr;
	caddr_t dptr, eptr;
	ddi_dma_cookie_t	cmd_cookie;
	ddi_dma_cookie_t	rsp_cookie;
	int		cmd_bound = FALSE, rsp_bound = FALSE;


	/* allocate room for the pool */
	if ((ptr = kmem_zalloc(sizeof (struct sf_cr_pool), KM_NOSLEEP)) ==
	    NULL) {
		return (DDI_FAILURE);
	}

	/* allocate a DMA handle for the command pool */
	if (ddi_dma_alloc_handle(sf->sf_dip, sf->sf_sochandle->fcal_dmaattr,
	    DDI_DMA_DONTWAIT, NULL, &ptr->cmd_dma_handle) != DDI_SUCCESS) {
		goto fail;
	}

	/*
	 * Get a piece of memory in which to put commands
	 */
	cmd_buf_size = (sizeof (struct fcp_cmd) * SF_ELEMS_IN_POOL + 7) & ~7;
	if (ddi_dma_mem_alloc(ptr->cmd_dma_handle, cmd_buf_size,
	    sf->sf_sochandle->fcal_accattr, DDI_DMA_CONSISTENT,
	    DDI_DMA_DONTWAIT, NULL, (caddr_t *)&ptr->cmd_base,
	    &real_cmd_buf_size, &ptr->cmd_acc_handle) != DDI_SUCCESS) {
		goto fail;
	}

	/* bind the DMA handle to an address */
	if (ddi_dma_addr_bind_handle(ptr->cmd_dma_handle, NULL,
	    ptr->cmd_base, real_cmd_buf_size,
	    DDI_DMA_WRITE | DDI_DMA_CONSISTENT, DDI_DMA_DONTWAIT,
	    NULL, &cmd_cookie, &ccount) != DDI_DMA_MAPPED) {
		goto fail;
	}
	cmd_bound = TRUE;
	/* ensure only one cookie was allocated */
	if (ccount != 1) {
		goto fail;
	}

	/* allocate a DMA handle for the response pool */
	if (ddi_dma_alloc_handle(sf->sf_dip, sf->sf_sochandle->fcal_dmaattr,
	    DDI_DMA_DONTWAIT, NULL, &ptr->rsp_dma_handle) != DDI_SUCCESS) {
		goto fail;
	}

	/*
	 * Get a piece of memory in which to put responses
	 */
	rsp_buf_size = FCP_MAX_RSP_IU_SIZE * SF_ELEMS_IN_POOL;
	if (ddi_dma_mem_alloc(ptr->rsp_dma_handle, rsp_buf_size,
	    sf->sf_sochandle->fcal_accattr, DDI_DMA_CONSISTENT,
	    DDI_DMA_DONTWAIT, NULL, (caddr_t *)&ptr->rsp_base,
	    &real_rsp_buf_size, &ptr->rsp_acc_handle) != DDI_SUCCESS) {
		goto fail;
	}

	/* bind the DMA handle to an address */
	if (ddi_dma_addr_bind_handle(ptr->rsp_dma_handle, NULL,
	    ptr->rsp_base, real_rsp_buf_size,
	    DDI_DMA_READ | DDI_DMA_CONSISTENT, DDI_DMA_DONTWAIT,
	    NULL, &rsp_cookie, &ccount) != DDI_DMA_MAPPED) {
		goto fail;
	}
	rsp_bound = TRUE;
	/* ensure only one cookie was allocated */
```

```
	if (ccount != 1) {
		goto fail;
	}

	/*
	 * Generate a (cmd/rsp structure) free list
	 */
	/* ensure ptr points to start of long word (8-byte block) */
	dptr = (caddr_t)((uintptr_t)(ptr->cmd_base) + 7 & ~7);
	/* keep track of actual size after moving pointer */
	real_cmd_buf_size -= (dptr - ptr->cmd_base);
	eptr = ptr->rsp_base;

	/* set actual total number of entries */
	ptr->ntot = min((real_cmd_buf_size / sizeof (struct fcp_cmd)),
	    (real_rsp_buf_size / FCP_MAX_RSP_IU_SIZE));
	ptr->nfree = ptr->ntot;
	ptr->free = (struct sf_cr_free_elem *)ptr->cmd_base;
	ptr->sf = sf;

	/* set up DMA for each pair of entries */
	i = 0;
	while (i < ptr->ntot) {
		cptr = (struct sf_cr_free_elem *)dptr;
		dptr += sizeof (struct fcp_cmd);

		cptr->next = (struct sf_cr_free_elem *)dptr;
		cptr->rsp = eptr;

		cptr->cmd_dmac = cmd_cookie.dmac_address +
		    (uint32_t)((caddr_t)cptr - ptr->cmd_base);

		cptr->rsp_dmac = rsp_cookie.dmac_address +
		    (uint32_t)((caddr_t)eptr - ptr->rsp_base);

		eptr += FCP_MAX_RSP_IU_SIZE;
		i++;
	}

	/* terminate the list */
	cptr->next = NULL;

	/* add this list at front of current one */
	mutex_enter(&sf->sf_cr_mutex);
	ptr->next = sf->sf_cr_pool;
	sf->sf_cr_pool = ptr;
	sf->sf_cr_pool_cnt++;
	mutex_exit(&sf->sf_cr_mutex);

	return (DDI_SUCCESS);

fail:
	/* we failed so clean up */
	if (ptr->cmd_dma_handle != NULL) {
		if (cmd_bound) {
			(void) ddi_dma_unbind_handle(ptr->cmd_dma_handle);
		}
		ddi_dma_free_handle(&ptr->cmd_dma_handle);
	}

	if (ptr->rsp_dma_handle != NULL) {
		if (rsp_bound) {
			(void) ddi_dma_unbind_handle(ptr->rsp_dma_handle);
		}
		ddi_dma_free_handle(&ptr->rsp_dma_handle);
	}
```

```
        if (ptr->cmd_base != NULL) {
                ddi_dma_mem_free(&ptr->cmd_acc_handle);
        }

        if (ptr->rsp_base != NULL) {
                ddi_dma_mem_free(&ptr->rsp_acc_handle);
        }

        kmem_free((caddr_t)ptr, sizeof (struct sf_cr_pool));
        return (DDI_FAILURE);
}


/*
 * allocate a command/response buffer from the pool, allocating more
 * in the pool as needed
 */
static int
sf_cr_alloc(struct sf *sf, struct sf_pkt *cmd, int (*func)())
{
        struct sf_cr_pool *ptr;
        struct sf_cr_free_elem *cptr;


        mutex_enter(&sf->sf_cr_mutex);

try_again:

        /* find a free buffer in the existing pool */
        ptr = sf->sf_cr_pool;
        while (ptr != NULL) {
                if (ptr->nfree != 0) {
                        ptr->nfree--;
                        break;
                } else {
                        ptr = ptr->next;
                }
        }

        /* did we find a free buffer ? */
        if (ptr != NULL) {
                /* we found a free buffer -- take it off the free list */
                cptr = ptr->free;
                ptr->free = cptr->next;
                mutex_exit(&sf->sf_cr_mutex);
                /* set up the command to use the buffer pair */
                cmd->cmd_block = (struct fcp_cmd *)cptr;
                cmd->cmd_dmac = cptr->cmd_dmac;
                cmd->cmd_rsp_dmac = cptr->rsp_dmac;
                cmd->cmd_rsp_block = (struct fcp_rsp *)cptr->rsp;
                cmd->cmd_cr_pool = ptr;
                return (DDI_SUCCESS);           /* success */
        }

        /* no free buffer available -- can we allocate more ? */
        if (sf->sf_cr_pool_cnt < SF_CR_POOL_MAX) {
                /* we need to allocate more buffer pairs */
                if (sf->sf_cr_flag) {
                        /* somebody already allocating for this instance */
                        if (func == SLEEP_FUNC) {
                                /* user wants to wait */
                                cv_wait(&sf->sf_cr_cv, &sf->sf_cr_mutex);
                                /* we've been woken so go try again */
                                goto try_again;
                        }
```

```
                        /* user does not want to wait */
                        mutex_exit(&sf->sf_cr_mutex);
                        sf->sf_stats.cralloc_failures++;
                        return (DDI_FAILURE);   /* give up */
                }
                /* set flag saying we're allocating */
                sf->sf_cr_flag = 1;
                mutex_exit(&sf->sf_cr_mutex);
                /* add to our pool */
                if (sf_add_cr_pool(sf) != DDI_SUCCESS) {
                        /* couldn't add to our pool for some reason */
                        mutex_enter(&sf->sf_cr_mutex);
                        sf->sf_cr_flag = 0;
                        cv_broadcast(&sf->sf_cr_cv);
                        mutex_exit(&sf->sf_cr_mutex);
                        sf->sf_stats.cralloc_failures++;
                        return (DDI_FAILURE);   /* give up */
                }
                /*
                 * clear flag saying we're allocating and tell all other
                 * that care
                 */
                mutex_enter(&sf->sf_cr_mutex);
                sf->sf_cr_flag = 0;
                cv_broadcast(&sf->sf_cr_cv);
                /* now that we have more buffers try again */
                goto try_again;
        }

        /* we don't have room to allocate any more buffers */
        mutex_exit(&sf->sf_cr_mutex);
        sf->sf_stats.cralloc_failures++;
        return (DDI_FAILURE);                   /* give up */
}


/*
 * free a cmd/response buffer pair in our pool
 */
static void
sf_cr_free(struct sf_cr_pool *cp, struct sf_pkt *cmd)
{
        struct sf *sf = cp->sf;
        struct sf_cr_free_elem *elem;

        elem = (struct sf_cr_free_elem *)cmd->cmd_block;
        elem->rsp = (caddr_t)cmd->cmd_rsp_block;
        elem->cmd_dmac = cmd->cmd_dmac;
        elem->rsp_dmac = cmd->cmd_rsp_dmac;

        mutex_enter(&sf->sf_cr_mutex);
        cp->nfree++;
        ASSERT(cp->nfree <= cp->ntot);

        elem->next = cp->free;
        cp->free = elem;
        mutex_exit(&sf->sf_cr_mutex);
}


/*
 * free our pool of cmd/response buffers
 */
static void
sf_crpool_free(struct sf *sf)
{
        struct sf_cr_pool *cp, *prev;

        prev = NULL;
        mutex_enter(&sf->sf_cr_mutex);
        cp = sf->sf_cr_pool;
        while (cp != NULL) {
                if (cp->nfree == cp->ntot) {
                        if (prev != NULL) {
                                prev->next = cp->next;
                        } else {
                                sf->sf_cr_pool = cp->next;
                        }
                        sf->sf_cr_pool_cnt--;
                        mutex_exit(&sf->sf_cr_mutex);

                        (void) ddi_dma_unbind_handle(cp->cmd_dma_handle);
                        ddi_dma_free_handle(&cp->cmd_dma_handle);
                        (void) ddi_dma_unbind_handle(cp->rsp_dma_handle);
                        ddi_dma_free_handle(&cp->rsp_dma_handle);
                        ddi_dma_mem_free(&cp->cmd_acc_handle);
                        ddi_dma_mem_free(&cp->rsp_acc_handle);
                        kmem_free((caddr_t)cp, sizeof (struct sf_cr_pool));
                        return;
                }
                prev = cp;
                cp = cp->next;
        }
        mutex_exit(&sf->sf_cr_mutex);
}


/* ARGSUSED */
static int
sf_kmem_cache_constructor(void *buf, void *arg, int size)
{
        struct sf_pkt *cmd = buf;

        mutex_init(&cmd->cmd_abort_mutex, NULL, MUTEX_DRIVER, NULL);
        cmd->cmd_block = NULL;
        cmd->cmd_dmahandle = NULL;
        return (0);
}


/* ARGSUSED */
static void
sf_kmem_cache_destructor(void *buf, void *size)
{
        struct sf_pkt *cmd = buf;

        if (cmd->cmd_dmahandle != NULL) {
                ddi_dma_free_handle(&cmd->cmd_dmahandle);
        }

        if (cmd->cmd_block != NULL) {
                sf_cr_free(cmd->cmd_cr_pool, cmd);
        }
        mutex_destroy(&cmd->cmd_abort_mutex);
}


/*
 * called by transport when a state change occurs
 */
static void
sf_statec_callback(void *arg, int msg)
```

```
{
        struct sf *sf = (struct sf *)arg;
        struct sf_target        *target;
        int i;
        struct sf_pkt *cmd;
        struct scsi_pkt *pkt;



        switch (msg) {

        case FCAL_STATUS_LOOP_ONLINE: {
                uchar_t         al_pa;          /* to save AL-PA */
                int             ret;            /* ret value from getmap */
                int             lip_cnt;        /* to save current count */
                int             cnt;            /* map length */

                /*
                 * the loop has gone online
                 */
                SF_DEBUG(1, (sf, CE_CONT, "sf%d: loop online\n",
                    ddi_get_instance(sf->sf_dip)));
                mutex_enter(&sf->sf_mutex);
                sf->sf_lip_cnt++;
                sf->sf_state = SF_STATE_ONLINING;
                mutex_exit(&sf->sf_mutex);

                /* scan each target hash queue */
                for (i = 0; i < SF_NUM_HASH_QUEUES; i++) {
                        target = sf->sf_wwn_lists[i];
                        while (target != NULL) {
                                /*
                                 * foreach target, if it's not offline then
                                 * mark it as busy
                                 */
                                mutex_enter(&target->sft_mutex);
                                if (!(target->sft_state & SF_TARGET_OFFLINE))
                                        target->sft_state |= (SF_TARGET_BUSY
                                            | SF_TARGET_MARK);
#ifdef DEBUG
                                /*
                                 * for debugging, print out info on any
                                 * pending commands (left hanging)
                                 */
                                cmd = target->sft_pkt_head;
                                while (cmd != (struct sf_pkt *)&target->
                                    sft_pkt_head) {
                                        if (cmd->cmd_state ==
                                            SF_STATE_ISSUED) {
                                                SF_DEBUG(1, (sf, CE_CONT,
                                                    "cmd 0x%p pending "
                                                    "after lip\n",
                                                    (void *)cmd->cmd_fp_pkt));
                                        }
                                        cmd = cmd->cmd_forw;
                                }
#endif
                                mutex_exit(&target->sft_mutex);
                                target = target->sft_next;
                        }
                }

                /*
                 * since the loop has just gone online get a new map from
                 * the transport
                 */
```

```
		if ((ret = soc_get_lilp_map(sf->sf_sochandle, sf->sf_socp,
		    sf->sf_sochandle->fcal_portno, (uint32_t)sf->
		    sf_lilp_dmacookie.dmac_address, 1)) != FCAL_SUCCESS) {
			if (sf_core && (sf_core & SF_CORE_LILP_FAILED)) {
				(void) soc_take_core(sf->sf_sochandle,
				    sf->sf_socp);
				sf_core = 0;
			}
			sf_log(sf, CE_WARN,
			    "!soc lilp map failed status=0x%x\n", ret);
			mutex_enter(&sf->sf_mutex);
			sf->sf_timer = sf_watchdog_time + SF_OFFLINE_TIMEOUT;
			sf->sf_lip_cnt++;
			sf->sf_state = SF_STATE_OFFLINE;
			mutex_exit(&sf->sf_mutex);
			return;
		}

		/* ensure consistent view of DMA memory */
		(void) ddi_dma_sync(sf->sf_lilp_dmahandle, (off_t)0, (size_t)0,
		    DDI_DMA_SYNC_FORKERNEL);

		/* how many entries in map ? */
		cnt = sf->sf_lilp_map->lilp_length;
		if (cnt >= SF_MAX_LILP_ENTRIES) {
			sf_log(sf, CE_WARN, "invalid lilp map\n");
			return;
		}

		mutex_enter(&sf->sf_mutex);
		sf->sf_device_count = cnt - 1;
		sf->sf_al_pa = sf->sf_lilp_map->lilp_myalpa;
		lip_cnt = sf->sf_lip_cnt;
		al_pa = sf->sf_al_pa;

		SF_DEBUG(1, (sf, CE_CONT,
		    "!lilp map has %d entries, al_pa is %x\n", cnt, al_pa));

		/*
		 * since the last entry of the map may be mine (common) check
		 * for that, and if it is we have one less entry to look at
		 */
		if (sf->sf_lilp_map->lilp_alpalist[cnt-1] == al_pa) {
			cnt--;
		}
		/* If we didn’t get a valid loop map enable all targets */
		if (sf->sf_lilp_map->lilp_magic == FCAL_BADLILP_MAGIC) {
			for (i = 0; i < sizeof (sf_switch_to_alpa); i++)
				sf->sf_lilp_map->lilp_alpalist[i] =
				    sf_switch_to_alpa[i];
			cnt = i;
			sf->sf_device_count = cnt - 1;
		}
		if (sf->sf_device_count == 0) {
			sf_finish_init(sf, lip_cnt);
			mutex_exit(&sf->sf_mutex);
			break;
		}
		mutex_exit(&sf->sf_mutex);

		SF_DEBUG(2, (sf, CE_WARN,
		    "!statec_callback: starting with %d targets\n",
		    sf->sf_device_count));

		/* scan loop map, logging into all ports (except mine) */
		for (i = 0; i < cnt; i++) {
```

```
			SF_DEBUG(1, (sf, CE_CONT,
			    "!lilp map entry %d = %x,%x\n", i,
			    sf->sf_lilp_map->lilp_alpalist[i],
			    sf_alpa_to_switch[
			    sf->sf_lilp_map->lilp_alpalist[i]]));
			/* is this entry for somebody else ? */
			if (sf->sf_lilp_map->lilp_alpalist[i] != al_pa) {
				/* do a PLOGI to this port */
				if (!sf_login(sf, LA_ELS_PLOGI,
				    sf->sf_lilp_map->lilp_alpalist[i],
				    sf->sf_lilp_map->lilp_alpalist[cnt-1],
				    lip_cnt)) {
					/* a problem logging in */
					mutex_enter(&sf->sf_mutex);
					if (lip_cnt == sf->sf_lip_cnt) {
						/*
						 * problem not from a new LIP
						 */
						sf->sf_device_count--;
						ASSERT(sf->sf_device_count
						    >= 0);
						if (sf->sf_device_count == 0) {
							sf_finish_init(sf,
							    lip_cnt);
						}
					}
					mutex_exit(&sf->sf_mutex);
				}
			}
		}
		break;
	}

	case FCAL_STATUS_ERR_OFFLINE:
		/*
		 * loop has gone offline due to an error
		 */
		SF_DEBUG(1, (sf, CE_CONT, "sf%d: loop offline\n",
		    ddi_get_instance(sf->sf_dip)));
		mutex_enter(&sf->sf_mutex);
		sf->sf_lip_cnt++;
		sf->sf_timer = sf_watchdog_time + SF_OFFLINE_TIMEOUT;
		if (!sf->sf_online_timer) {
			sf->sf_online_timer = sf_watchdog_time +
			    SF_ONLINE_TIMEOUT;
		}
		/*
		 * if we are suspended, preserve the SF_STATE_SUSPENDED flag,
		 * since throttling logic in sf_watch() depends on
		 * preservation of this flag while device is suspended
		 */
		if (sf->sf_state & SF_STATE_SUSPENDED) {
			sf->sf_state |= SF_STATE_OFFLINE;
			SF_DEBUG(1, (sf, CE_CONT,
			    "sf_statec_callback, sf%d: "
			    "got FCAL_STATE_OFFLINE during DDI_SUSPEND\n",
			    ddi_get_instance(sf->sf_dip)));
		} else {
			sf->sf_state = SF_STATE_OFFLINE;
		}

		/* scan each possible target on the loop */
		for (i = 0; i < sf_max_targets; i++) {
			target = sf->sf_targets[i];
			while (target != NULL) {
				mutex_enter(&target->sft_mutex);
```

```
                                if (!(target->sft_state & SF_TARGET_OFFLINE))
                                        target->sft_state |= (SF_TARGET_BUSY
                                            | SF_TARGET_MARK);
                                mutex_exit(&target->sft_mutex);
                                target = target->sft_next_lun;
                        }
                }
                mutex_exit(&sf->sf_mutex);
                break;

        case FCAL_STATE_RESET: {
                struct sf_els_hdr       *privp; /* ptr to private list */
                struct sf_els_hdr       *tmpp1; /* tmp prev hdr ptr */
                struct sf_els_hdr       *tmpp2; /* tmp next hdr ptr */
                struct sf_els_hdr       *head;  /* to save our private list */
                struct fcal_packet      *fpkt;  /* ptr to pkt in hdr */

                /*
                 * a transport reset
                 */
                SF_DEBUG(1, (sf, CE_CONT, "!sf%d: soc reset\n",
                    ddi_get_instance(sf->sf_dip)));
                tmpp1 = head = NULL;
                mutex_enter(&sf->sf_mutex);
                sf->sf_lip_cnt++;
                sf->sf_timer = sf_watchdog_time + SF_RESET_TIMEOUT;
                /*
                 * if we are suspended, preserve the SF_STATE_SUSPENDED flag,
                 * since throttling logic in sf_watch() depends on
                 * preservation of this flag while device is suspended
                 */
                if (sf->sf_state & SF_STATE_SUSPENDED) {
                        sf->sf_state |= SF_STATE_OFFLINE;
                        SF_DEBUG(1, (sf, CE_CONT,
                            "sf_statec_callback, sf%d: "
                            "got FCAL_STATE_RESET during DDI_SUSPEND\n",
                            ddi_get_instance(sf->sf_dip)));
                } else {
                        sf->sf_state = SF_STATE_OFFLINE;
                }

                /*
                 * scan each possible target on the loop, looking for targets
                 * that need callbacks ran
                 */
                for (i = 0; i < sf_max_targets; i++) {
                        target = sf->sf_targets[i];
                        while (target != NULL) {
                                if (!(target->sft_state & SF_TARGET_OFFLINE)) {
                                        target->sft_state |= (SF_TARGET_BUSY
                                            | SF_TARGET_MARK);
                                        mutex_exit(&sf->sf_mutex);
                                        /*
                                         * run remove event callbacks for lun
                                         *
                                         * We have a nasty race condition here
                                         * 'cause we're dropping this mutex to
                                         * run the callback and expect the
                                         * linked list to be the same.
                                         */
                                        (void) ndi_event_retrieve_cookie(
                                            sf->sf_event_hdl, target->sft_dip,
                                            FCAL_REMOVE_EVENT, &sf_remove_eid,
                                            NDI_EVENT_NOPASS);
                                        (void) ndi_event_run_callbacks(
                                            sf->sf_event_hdl,
```

```
                                            target->sft_dip,
                                            sf_remove_eid, NULL);
                                        mutex_enter(&sf->sf_mutex);
                                }
                                target = target->sft_next_lun;
                        }
                }

                /*
                 * scan for ELS commands that are in transport, not complete,
                 * and have a valid timeout, building a private list
                 */
                privp = sf->sf_els_list;
                while (privp != NULL) {
                        fpkt = privp->fpkt;
                        if ((fpkt->fcal_cmd_state & FCAL_CMD_IN_TRANSPORT) &&
                            (!(fpkt->fcal_cmd_state & FCAL_CMD_COMPLETE)) &&
                            (privp->timeout != SF_INVALID_TIMEOUT)) {
                                /*
                                 * cmd in transport && not complete &&
                                 * timeout valid
                                 *
                                 * move this entry from ELS input list to our
                                 * private list
                                 */

                                tmpp2 = privp->next; /* save ptr to next */

                                /* push this on private list head */
                                privp->next = head;
                                head = privp;

                                /* remove this entry from input list */
                                if (tmpp1 != NULL) {
                                        /*
                                         * remove this entry from somewhere in
                                         * the middle of the list
                                         */
                                        tmpp1->next = tmpp2;
                                        if (tmpp2 != NULL) {
                                                tmpp2->prev = tmpp1;
                                        }
                                } else {
                                        /*
                                         * remove this entry from the head
                                         * of the list
                                         */
                                        sf->sf_els_list = tmpp2;
                                        if (tmpp2 != NULL) {
                                                tmpp2->prev = NULL;
                                        }
                                }
                                privp = tmpp2;  /* skip to next entry */
                        } else {
                                tmpp1 = privp;  /* save ptr to prev entry */
                                privp = privp->next; /* skip to next entry */
                        }
                }

                mutex_exit(&sf->sf_mutex);

                /*
                 * foreach cmd in our list free the ELS packet associated
                 * with it
                 */
                privp = head;
```

```
		while (privp != NULL) {
			fpkt = privp->fpkt;
			privp = privp->next;
			sf_els_free(fpkt);
		}

		/*
		 * scan for commands from each possible target
		 */
		for (i = 0; i < sf_max_targets; i++) {
			target = sf->sf_targets[i];
			while (target != NULL) {
				/*
				 * scan all active commands for this target,
				 * looking for commands that have been issued,
				 * are in transport, and are not yet complete
				 * (so we can terminate them because of the
				 * reset)
				 */
				mutex_enter(&target->sft_pkt_mutex);
				cmd = target->sft_pkt_head;
				while (cmd != (struct sf_pkt *)&target->
				    sft_pkt_head) {
					fpkt = cmd->cmd_fp_pkt;
					mutex_enter(&cmd->cmd_abort_mutex);
					if ((cmd->cmd_state ==
					    SF_STATE_ISSUED) &&
					    (fpkt->fcal_cmd_state &
					    FCAL_CMD_IN_TRANSPORT) &&
					    (!(fpkt->fcal_cmd_state &
					    FCAL_CMD_COMPLETE))) {
						/* a command to be reset */
						pkt = cmd->cmd_pkt;
						pkt->pkt_reason = CMD_RESET;
						pkt->pkt_statistics |=
						    STAT_BUS_RESET;
						cmd->cmd_state = SF_STATE_IDLE;
						mutex_exit(&cmd->
						    cmd_abort_mutex);
						mutex_exit(&target->
						    sft_pkt_mutex);
						if (pkt->pkt_comp != NULL) {
							(*pkt->pkt_comp)(pkt);
						}
						mutex_enter(&target->
						    sft_pkt_mutex);
						cmd = target->sft_pkt_head;
					} else {
						mutex_exit(&cmd->
						    cmd_abort_mutex);
						/* get next command */
						cmd = cmd->cmd_forw;
					}
				}
				mutex_exit(&target->sft_pkt_mutex);
				target = target->sft_next_lun;
			}
		}

		/*
		 * get packet queue for this target, resetting all remaining
		 * commands
		 */
		mutex_enter(&sf->sf_mutex);
		cmd = sf->sf_pkt_head;
		sf->sf_pkt_head = NULL;
```

```
		mutex_exit(&sf->sf_mutex);

		while (cmd != NULL) {
			pkt = cmd->cmd_pkt;
			cmd = cmd->cmd_next;
			pkt->pkt_reason = CMD_RESET;
			pkt->pkt_statistics |= STAT_BUS_RESET;
			if (pkt->pkt_comp != NULL) {
				(*pkt->pkt_comp)(pkt);
			}
		}
		break;
	}

	default:
		break;
	}
}


/*
 * called to send a PLOGI (N_port login) ELS request to a destination ID,
 * returning TRUE upon success, else returning FALSE
 */
static int
sf_login(struct sf *sf, uchar_t els_code, uchar_t dest_id, uint_t arg1,
    int lip_cnt)
{
	struct la_els_logi	*logi;
	struct  sf_els_hdr	*privp;


	if (sf_els_alloc(sf, dest_id, sizeof (struct sf_els_hdr),
	    sizeof (union sf_els_cmd), sizeof (union sf_els_rsp),
	    (caddr_t *)&privp, (caddr_t *)&logi) == NULL) {
		sf_log(sf, CE_WARN, "Cannot allocate PLOGI for target %x "
		    "due to DVMA shortage.\n", sf_alpa_to_switch[dest_id]);
		return (FALSE);
	}

	privp->lip_cnt = lip_cnt;
	if (els_code == LA_ELS_PLOGI) {
		bcopy((caddr_t)sf->sf_sochandle->fcal_loginparms,
		    (caddr_t)&logi->common_service, sizeof (struct la_els_logi)
		    - 4);
		bcopy((caddr_t)&sf->sf_sochandle->fcal_p_wwn,
		    (caddr_t)&logi->nport_ww_name, sizeof (la_wwn_t));
		bcopy((caddr_t)&sf->sf_sochandle->fcal_n_wwn,
		    (caddr_t)&logi->node_ww_name, sizeof (la_wwn_t));
		bzero((caddr_t)&logi->reserved, 16);
	} else if (els_code == LA_ELS_LOGO) {
		bcopy((caddr_t)&sf->sf_sochandle->fcal_p_wwn,
		    (caddr_t)&(((struct la_els_logo *)logi)->nport_ww_name), 8);
		((struct la_els_logo	*)logi)->reserved = 0;
		((struct la_els_logo	*)logi)->nport_id[0] = 0;
		((struct la_els_logo	*)logi)->nport_id[1] = 0;
		((struct la_els_logo	*)logi)->nport_id[2] = arg1;
	}

	privp->els_code = els_code;
	logi->ls_code = els_code;
	logi->mbz[0] = 0;
	logi->mbz[1] = 0;
	logi->mbz[2] = 0;

	privp->timeout = sf_watchdog_time + SF_ELS_TIMEOUT;
```

```
        return (sf_els_transport(sf, privp));
}


/*
 * send an ELS IU via the transport,
 * returning TRUE upon success, else returning FALSE
 */
static int
sf_els_transport(struct sf *sf, struct sf_els_hdr *privp)
{
        struct fcal_packet *fpkt = privp->fpkt;


        (void) ddi_dma_sync(privp->cmd_dma_handle, (off_t)0, (size_t)0,
            DDI_DMA_SYNC_FORDEV);
        privp->prev = NULL;
        mutex_enter(&sf->sf_mutex);
        privp->next = sf->sf_els_list;
        if (sf->sf_els_list != NULL) {
                sf->sf_els_list->prev = privp;
        }
        sf->sf_els_list = privp;
        mutex_exit(&sf->sf_mutex);

        /* call the transport to send a packet */
        if (soc_transport(sf->sf_sochandle, fpkt, FCAL_NOSLEEP,
            CQ_REQUEST_1) != FCAL_TRANSPORT_SUCCESS) {
                mutex_enter(&sf->sf_mutex);
                if (privp->prev != NULL) {
                        privp->prev->next = privp->next;
                }
                if (privp->next != NULL) {
                        privp->next->prev = privp->prev;
                }
                if (sf->sf_els_list == privp) {
                        sf->sf_els_list = privp->next;
                }
                mutex_exit(&sf->sf_mutex);
                sf_els_free(fpkt);
                return (FALSE);                 /* failure */
        }
        return (TRUE);                          /* success */
}


/*
 * called as the pkt_comp routine for ELS FC packets
 */
static void
sf_els_callback(struct fcal_packet *fpkt)
{
        struct sf_els_hdr *privp = fpkt->fcal_pkt_private;
        struct sf *sf = privp->sf;
        struct sf *tsf;
        int tgt_id;
        struct la_els_logi *ptr = (struct la_els_logi *)privp->rsp;
        struct la_els_adisc *adisc = (struct la_els_adisc *)ptr;
        struct  sf_target *target;
        short   ncmds;
        short   free_pkt = TRUE;


        /*
         * we’ve received an ELS callback, i.e. an ELS packet has arrived
         */
```

```

        /* take the current packet off of the queue */
        mutex_enter(&sf->sf_mutex);
        if (privp->timeout == SF_INVALID_TIMEOUT) {
                mutex_exit(&sf->sf_mutex);
                return;
        }
        if (privp->prev != NULL) {
                privp->prev->next = privp->next;
        }
        if (privp->next != NULL) {
                privp->next->prev = privp->prev;
        }
        if (sf->sf_els_list == privp) {
                sf->sf_els_list = privp->next;
        }
        privp->prev = privp->next = NULL;
        mutex_exit(&sf->sf_mutex);

        /* get # pkts in this callback */
        ncmds = fpkt->fcal_ncmds;
        ASSERT(ncmds >= 0);
        mutex_enter(&sf->sf_cmd_mutex);
        sf->sf_ncmds = ncmds;
        mutex_exit(&sf->sf_cmd_mutex);

        /* sync idea of memory */
        (void) ddi_dma_sync(privp->rsp_dma_handle, (off_t)0, (size_t)0,
            DDI_DMA_SYNC_FORKERNEL);

        /* was this an OK ACC msg ?? */
        if ((fpkt->fcal_pkt_status == FCAL_STATUS_OK) &&
            (ptr->ls_code == LA_ELS_ACC)) {

                /*
                 * this was an OK ACC pkt
                 */

                switch (privp->els_code) {
                case LA_ELS_PLOGI:
                        /*
                         * was able to to an N_port login
                         */
                        SF_DEBUG(2, (sf, CE_CONT,
                            "!PLOGI to al_pa %x succeeded, wwn %x%x\n",
                            privp->dest_nport_id,
                            *((int *)&ptr->nport_ww_name.raw_wwn[0]),
                            *((int *)&ptr->nport_ww_name.raw_wwn[4])));
                        /* try to do a process login */
                        if (!sf_do_prli(sf, privp, ptr)) {
                                free_pkt = FALSE;
                                goto fail;      /* PRLI failed */
                        }
                        break;
                case LA_ELS_PRLI:
                        /*
                         * was able to do a process login
                         */
                        SF_DEBUG(2, (sf, CE_CONT,
                            "!PRLI to al_pa %x succeeded\n",
                            privp->dest_nport_id));
                        /* try to do address discovery */
                        if (sf_do_adisc(sf, privp) != 1) {
                                free_pkt = FALSE;
                                goto fail;      /* ADISC failed */
                        }
                        break;
                case LA_ELS_ADISC:
                        /*
                         * found a target via ADISC
                         */

                        SF_DEBUG(2, (sf, CE_CONT,
                            "!ADISC to al_pa %x succeeded\n",
                            privp->dest_nport_id));

                        /* create the target info */
                        if ((target = sf_create_target(sf, privp,
                            sf_alpa_to_switch[(uchar_t)adisc->hard_address],
                            (int64_t)0))
                            == NULL) {
                                goto fail;      /* can't create target */
                        }

                        /*
                         * ensure address discovered matches what we thought
                         * it would be
                         */
                        if ((uchar_t)adisc->hard_address !=
                            privp->dest_nport_id) {
                                sf_log(sf, CE_WARN,
                                    "target 0x%x, AL-PA 0x%x and "
                                    "hard address 0x%x don't match\n",
                                    sf_alpa_to_switch[
                                    (uchar_t)privp->dest_nport_id],
                                    privp->dest_nport_id,
                                    (uchar_t)adisc->hard_address);
                                mutex_enter(&sf->sf_mutex);
                                sf_offline_target(sf, target);
                                mutex_exit(&sf->sf_mutex);
                                goto fail;      /* addr doesn't match */
                        }
                        /*
                         * get inquiry data from the target
                         */
                        if (!sf_do_reportlun(sf, privp, target)) {
                                mutex_enter(&sf->sf_mutex);
                                sf_offline_target(sf, target);
                                mutex_exit(&sf->sf_mutex);
                                free_pkt = FALSE;
                                goto fail;      /* inquiry failed */
                        }
                        break;
                default:
                        SF_DEBUG(2, (sf, CE_CONT,
                            "!ELS %x to al_pa %x succeeded\n",
                            privp->els_code, privp->dest_nport_id));
                        sf_els_free(fpkt);
                        break;
                }

        } else {

                /*
                 * oh oh -- this was not an OK ACC packet
                 */

                /* get target ID from dest loop address */
                tgt_id = sf_alpa_to_switch[(uchar_t)privp->dest_nport_id];

                /* keep track of failures */
                sf->sf_stats.tstats[tgt_id].els_failures++;
```

```
                if (++(privp->retries) < sf_els_retries &&
                    fpkt->fcal_pkt_status != FCAL_STATUS_OPEN_FAIL) {
                        if (fpkt->fcal_pkt_status ==
                            FCAL_STATUS_MAX_XCHG_EXCEEDED)  {
                                tsf = sf->sf_sibling;
                                if (tsf != NULL) {
                                        mutex_enter(&tsf->sf_cmd_mutex);
                                        tsf->sf_flag = 1;
                                        tsf->sf_throttle = SF_DECR_DELTA;
                                        mutex_exit(&tsf->sf_cmd_mutex);
                                }
                        }
                        privp->timeout = sf_watchdog_time + SF_ELS_TIMEOUT;
                        privp->prev = NULL;

                        mutex_enter(&sf->sf_mutex);

                        if (privp->lip_cnt == sf->sf_lip_cnt) {
                                SF_DEBUG(1, (sf, CE_WARN,
                                    "!ELS %x to al_pa %x failed, retrying",
                                    privp->els_code, privp->dest_nport_id));
                                privp->next = sf->sf_els_list;
                                if (sf->sf_els_list != NULL) {
                                        sf->sf_els_list->prev = privp;
                                }

                                sf->sf_els_list = privp;

                                mutex_exit(&sf->sf_mutex);
                                /* device busy?  wait a bit ... */
                                if (fpkt->fcal_pkt_status ==
                                    FCAL_STATUS_MAX_XCHG_EXCEEDED)  {
                                        privp->delayed_retry = 1;
                                        return;
                                }
                                /* call the transport to send a pkt */
                                if (soc_transport(sf->sf_sochandle, fpkt,
                                    FCAL_NOSLEEP, CQ_REQUEST_1) !=
                                    FCAL_TRANSPORT_SUCCESS) {
                                        mutex_enter(&sf->sf_mutex);
                                        if (privp->prev != NULL) {
                                                privp->prev->next =
                                                    privp->next;
                                        }
                                        if (privp->next != NULL) {
                                                privp->next->prev =
                                                    privp->prev;
                                        }
                                        if (sf->sf_els_list == privp) {
                                                sf->sf_els_list = privp->next;
                                        }
                                        mutex_exit(&sf->sf_mutex);
                                        goto fail;
                                } else
                                        return;
                        } else {
                                mutex_exit(&sf->sf_mutex);
                                goto fail;
                        }
                } else {
#ifdef  DEBUG
                        if (fpkt->fcal_pkt_status != 0x36 || sfdebug > 4) {
                        SF_DEBUG(2, (sf, CE_NOTE, "ELS %x to al_pa %x failed",
                            privp->els_code, privp->dest_nport_id));
                        if (fpkt->fcal_pkt_status == FCAL_STATUS_OK) {
                                SF_DEBUG(2, (sf, CE_NOTE,
                                    "els reply code = %x", ptr->ls_code));
                                if (ptr->ls_code == LA_ELS_RJT)
                                        SF_DEBUG(1, (sf, CE_CONT,
                                            "LS_RJT reason = %x\n",
                                            *(((uint_t *)ptr) + 1)));
                        } else
                                SF_DEBUG(2, (sf, CE_NOTE,
                                    "fc packet status = %x",
                                    fpkt->fcal_pkt_status));
                        }
#endif
                        goto fail;
                }
        }
        return;                                 /* success */
fail:
        mutex_enter(&sf->sf_mutex);
        if (sf->sf_lip_cnt == privp->lip_cnt) {
                sf->sf_device_count--;
                ASSERT(sf->sf_device_count >= 0);
                if (sf->sf_device_count == 0) {
                        sf_finish_init(sf, privp->lip_cnt);
                }
        }
        mutex_exit(&sf->sf_mutex);
        if (free_pkt) {
                sf_els_free(fpkt);
        }
}


/*
 * send a PRLI (process login) ELS IU via the transport,
 * returning TRUE upon success, else returning FALSE
 */
static int
sf_do_prli(struct sf *sf, struct sf_els_hdr *privp, struct la_els_logi *ptr)
{
        struct la_els_prli      *prli = (struct la_els_prli *)privp->cmd;
        struct fcp_prli         *fprli;
        struct  fcal_packet     *fpkt = privp->fpkt;


        fpkt->fcal_socal_request.sr_dataseg[0].fc_count =
            sizeof (struct la_els_prli);
        privp->els_code = LA_ELS_PRLI;
        fprli = (struct fcp_prli *)prli->service_params;
        prli->ls_code = LA_ELS_PRLI;
        prli->page_length = 0x10;
        prli->payload_length = sizeof (struct la_els_prli);
        fprli->type = 0x08;                     /* no define here? */
        fprli->resvd1 = 0;
        fprli->orig_process_assoc_valid = 0;
        fprli->resp_process_assoc_valid = 0;
        fprli->establish_image_pair = 1;
        fprli->resvd2 = 0;
        fprli->resvd3 = 0;
        fprli->data_overlay_allowed = 0;
        fprli->initiator_fn = 1;
        fprli->target_fn = 0;
        fprli->cmd_data_mixed = 0;
        fprli->data_resp_mixed = 0;
        fprli->read_xfer_rdy_disabled = 1;
        fprli->write_xfer_rdy_disabled = 0;

        bcopy((caddr_t)&ptr->nport_ww_name, (caddr_t)&privp->port_wwn,
```

```
            sizeof (privp->port_wwn));
        bcopy((caddr_t)&ptr->node_ww_name, (caddr_t)&privp->node_wwn,
            sizeof (privp->node_wwn));

        privp->timeout = sf_watchdog_time + SF_ELS_TIMEOUT;
        return (sf_els_transport(sf, privp));
}


/*
 * send an ADISC (address discovery) ELS IU via the transport,
 * returning TRUE upon success, else returning FALSE
 */
static int
sf_do_adisc(struct sf *sf, struct sf_els_hdr *privp)
{
        struct la_els_adisc     *adisc = (struct la_els_adisc *)privp->cmd;
        struct  fcal_packet     *fpkt = privp->fpkt;

        privp->els_code = LA_ELS_ADISC;
        adisc->ls_code = LA_ELS_ADISC;
        adisc->mbz[0] = 0;
        adisc->mbz[1] = 0;
        adisc->mbz[2] = 0;
        adisc->hard_address = 0; /* ??? */
        fpkt->fcal_socal_request.sr_dataseg[0].fc_count =
            sizeof (struct la_els_adisc);
        bcopy((caddr_t)&sf->sf_sochandle->fcal_p_wwn,
            (caddr_t)&adisc->port_wwn, sizeof (adisc->port_wwn));
        bcopy((caddr_t)&sf->sf_sochandle->fcal_n_wwn,
            (caddr_t)&adisc->node_wwn, sizeof (adisc->node_wwn));
        adisc->nport_id = sf->sf_al_pa;

        privp->timeout = sf_watchdog_time + SF_ELS_TIMEOUT;
        return (sf_els_transport(sf, privp));
}


static struct fcal_packet *
sf_els_alloc(struct sf *sf, uchar_t dest_id, int priv_size, int cmd_size,
    int rsp_size, caddr_t *rprivp, caddr_t *cmd_buf)
{
        struct  fcal_packet     *fpkt;
        ddi_dma_cookie_t        pcookie;
        ddi_dma_cookie_t        rcookie;
        struct  sf_els_hdr      *privp;
        ddi_dma_handle_t        cmd_dma_handle = NULL;
        ddi_dma_handle_t        rsp_dma_handle = NULL;
        ddi_acc_handle_t        cmd_acc_handle = NULL;
        ddi_acc_handle_t        rsp_acc_handle = NULL;
        size_t                  real_size;
        uint_t                  ccount;
        fc_frame_header_t       *hp;
        int                     cmd_bound = FALSE, rsp_bound = FALSE;
        caddr_t                 cmd = NULL;
        caddr_t                 rsp = NULL;

        if ((fpkt = (struct fcal_packet *)kmem_zalloc(
            sizeof (struct fcal_packet), KM_NOSLEEP)) == NULL) {
                SF_DEBUG(1, (sf, CE_WARN,
                        "Could not allocate fcal_packet for ELS\n"));
                return (NULL);
        }

        if ((privp = (struct sf_els_hdr *)kmem_zalloc(priv_size,
            KM_NOSLEEP)) == NULL) {
                SF_DEBUG(1, (sf, CE_WARN,
                    "Could not allocate sf_els_hdr for ELS\n"));
                goto fail;
        }

        privp->size = priv_size;
        fpkt->fcal_pkt_private = (caddr_t)privp;

        if (ddi_dma_alloc_handle(sf->sf_dip, sf->sf_sochandle->fcal_dmaattr,
            DDI_DMA_DONTWAIT, NULL, &cmd_dma_handle) != DDI_SUCCESS) {
                SF_DEBUG(1, (sf, CE_WARN,
                    "Could not allocate DMA handle for ELS\n"));
                goto fail;
        }

        if (ddi_dma_mem_alloc(cmd_dma_handle, cmd_size,
            sf->sf_sochandle->fcal_accattr, DDI_DMA_CONSISTENT,
            DDI_DMA_DONTWAIT, NULL, &cmd,
            &real_size, &cmd_acc_handle) != DDI_SUCCESS) {
                SF_DEBUG(1, (sf, CE_WARN,
                    "Could not allocate DMA memory for ELS\n"));
                goto fail;
        }

        if (real_size < cmd_size) {
                SF_DEBUG(1, (sf, CE_WARN,
                    "DMA memory too small for ELS\n"));
                goto fail;
        }

        if (ddi_dma_addr_bind_handle(cmd_dma_handle, NULL,
            cmd, real_size, DDI_DMA_WRITE | DDI_DMA_CONSISTENT,
            DDI_DMA_DONTWAIT, NULL, &pcookie, &ccount) != DDI_DMA_MAPPED) {
                SF_DEBUG(1, (sf, CE_WARN,
                    "Could not bind DMA memory for ELS\n"));
                goto fail;
        }
        cmd_bound = TRUE;

        if (ccount != 1) {
                SF_DEBUG(1, (sf, CE_WARN,
                    "Wrong cookie count for ELS\n"));
                goto fail;
        }

        if (ddi_dma_alloc_handle(sf->sf_dip, sf->sf_sochandle->fcal_dmaattr,
            DDI_DMA_DONTWAIT, NULL, &rsp_dma_handle) != DDI_SUCCESS) {
                SF_DEBUG(1, (sf, CE_WARN,
                    "Could not allocate DMA handle for ELS rsp\n"));
                goto fail;
        }
        if (ddi_dma_mem_alloc(rsp_dma_handle, rsp_size,
            sf->sf_sochandle->fcal_accattr, DDI_DMA_CONSISTENT,
            DDI_DMA_DONTWAIT, NULL, &rsp,
            &real_size, &rsp_acc_handle) != DDI_SUCCESS) {
                SF_DEBUG(1, (sf, CE_WARN,
                    "Could not allocate DMA memory for ELS rsp\n"));
                goto fail;
        }

        if (real_size < rsp_size) {
                SF_DEBUG(1, (sf, CE_WARN,
                    "DMA memory too small for ELS rsp\n"));
                goto fail;
        }
```

```
        if (ddi_dma_addr_bind_handle(rsp_dma_handle, NULL,
            rsp, real_size, DDI_DMA_READ | DDI_DMA_CONSISTENT,
            DDI_DMA_DONTWAIT, NULL, &rcookie, &ccount) != DDI_DMA_MAPPED) {
                SF_DEBUG(1, (sf, CE_WARN,
                    "Could not bind DMA memory for ELS rsp\n"));
                goto fail;
        }
        rsp_bound = TRUE;

        if (ccount != 1) {
                SF_DEBUG(1, (sf, CE_WARN,
                    "Wrong cookie count for ELS rsp\n"));
                goto fail;
        }

        privp->cmd = cmd;
        privp->sf = sf;
        privp->cmd_dma_handle = cmd_dma_handle;
        privp->cmd_acc_handle = cmd_acc_handle;
        privp->rsp = rsp;
        privp->rsp_dma_handle = rsp_dma_handle;
        privp->rsp_acc_handle = rsp_acc_handle;
        privp->dest_nport_id = dest_id;
        privp->fpkt = fpkt;

        fpkt->fcal_pkt_cookie = sf->sf_socp;
        fpkt->fcal_pkt_comp = sf_els_callback;
        fpkt->fcal_magic = FCALP_MAGIC;
        fpkt->fcal_pkt_flags = 0;
        fpkt->fcal_socal_request.sr_soc_hdr.sh_flags =
            (ushort_t)(SOC_FC_HEADER | sf->sf_sochandle->fcal_portno);
        fpkt->fcal_socal_request.sr_soc_hdr.sh_class = 3;
        fpkt->fcal_socal_request.sr_soc_hdr.sh_seg_cnt = 2;
        fpkt->fcal_socal_request.sr_soc_hdr.sh_byte_cnt = cmd_size;
        fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_count = 1;
        fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_flags = 0;
        fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_seqno = 0;
        fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_type = CQ_TYPE_SIMPLE;
        fpkt->fcal_socal_request.sr_dataseg[0].fc_base = (uint32_t)
            pcookie.dmac_address;
        fpkt->fcal_socal_request.sr_dataseg[0].fc_count = cmd_size;
        fpkt->fcal_socal_request.sr_dataseg[1].fc_base = (uint32_t)
            rcookie.dmac_address;
        fpkt->fcal_socal_request.sr_dataseg[1].fc_count = rsp_size;

        /* Fill in the Fabric Channel Header */
        hp = &fpkt->fcal_socal_request.sr_fc_frame_hdr;
        hp->r_ctl = R_CTL_ELS_REQ;
        hp->d_id = dest_id;
        hp->s_id = sf->sf_al_pa;
        hp->type = TYPE_EXTENDED_LS;
        hp->reserved1 = 0;
        hp->f_ctl = F_CTL_SEQ_INITIATIVE | F_CTL_FIRST_SEQ;
        hp->seq_id = 0;
        hp->df_ctl  = 0;
        hp->seq_cnt = 0;
        hp->ox_id = 0xffff;
        hp->rx_id = 0xffff;
        hp->ro = 0;

        *rprivp = (caddr_t)privp;
        *cmd_buf = cmd;
        return (fpkt);

fail:
        if (cmd_dma_handle != NULL) {
```

```
		if (cmd_bound) {
			(void) ddi_dma_unbind_handle(cmd_dma_handle);
		}
		ddi_dma_free_handle(&cmd_dma_handle);
		privp->cmd_dma_handle = NULL;
	}
	if (rsp_dma_handle != NULL) {
		if (rsp_bound) {
			(void) ddi_dma_unbind_handle(rsp_dma_handle);
		}
		ddi_dma_free_handle(&rsp_dma_handle);
		privp->rsp_dma_handle = NULL;
	}
	sf_els_free(fpkt);
	return (NULL);
}


static void
sf_els_free(struct fcal_packet *fpkt)
{
	struct	sf_els_hdr	*privp = fpkt->fcal_pkt_private;

	if (privp != NULL) {
		if (privp->cmd_dma_handle != NULL) {
			(void) ddi_dma_unbind_handle(privp->cmd_dma_handle);
			ddi_dma_free_handle(&privp->cmd_dma_handle);
		}
		if (privp->cmd != NULL) {
			ddi_dma_mem_free(&privp->cmd_acc_handle);
		}

		if (privp->rsp_dma_handle != NULL) {
			(void) ddi_dma_unbind_handle(privp->rsp_dma_handle);
			ddi_dma_free_handle(&privp->rsp_dma_handle);
		}

		if (privp->rsp != NULL) {
			ddi_dma_mem_free(&privp->rsp_acc_handle);
		}
		if (privp->data_dma_handle) {
			(void) ddi_dma_unbind_handle(privp->data_dma_handle);
			ddi_dma_free_handle(&privp->data_dma_handle);
		}
		if (privp->data_buf) {
			ddi_dma_mem_free(&privp->data_acc_handle);
		}
		kmem_free(privp, privp->size);
	}
	kmem_free(fpkt, sizeof (struct fcal_packet));
}


static struct sf_target *
sf_create_target(struct sf *sf, struct sf_els_hdr *privp, int tnum, int64_t lun)
{
	struct sf_target *target, *ntarget, *otarget, *ptarget;
	int hash;
#ifdef RAID_LUNS
	int64_t orig_lun = lun;

	/* XXXX Work around SCSA limitations. */
	lun = *((short *)&lun);
#endif
	ntarget = kmem_zalloc(sizeof (struct sf_target), KM_NOSLEEP);
	mutex_enter(&sf->sf_mutex);
```

```
	if (sf->sf_lip_cnt != privp->lip_cnt) {
		mutex_exit(&sf->sf_mutex);
		if (ntarget != NULL)
			kmem_free(ntarget, sizeof (struct sf_target));
		return (NULL);
	}

	target = sf_lookup_target(sf, privp->port_wwn, lun);
	if (lun != 0) {
		/*
		 * Since LUNs != 0 are queued up after LUN == 0, find LUN == 0
		 * and enqueue the new LUN.
		 */
		if ((ptarget = sf_lookup_target(sf, privp->port_wwn,
		    (int64_t)0)) ==	NULL) {
			/*
			 * Yeep -- no LUN 0?
			 */
			mutex_exit(&sf->sf_mutex);
			sf_log(sf, CE_WARN, "target 0x%x "
			    "lun %" PRIx64 ": No LUN 0\n", tnum, lun);
			if (ntarget != NULL)
				kmem_free(ntarget, sizeof (struct sf_target));
			return (NULL);
		}
		mutex_enter(&ptarget->sft_mutex);
		if (target != NULL && ptarget->sft_lip_cnt == sf->sf_lip_cnt &&
		    ptarget->sft_state&SF_TARGET_OFFLINE) {
			/* LUN 0 already finished, duplicate its state */
			mutex_exit(&ptarget->sft_mutex);
			sf_offline_target(sf, target);
			mutex_exit(&sf->sf_mutex);
			if (ntarget != NULL)
				kmem_free(ntarget, sizeof (struct sf_target));
			return (target);
		} else if (target != NULL) {
			/*
			 * LUN 0 online or not examined yet.
			 * Try to bring the LUN back online
			 */
			mutex_exit(&ptarget->sft_mutex);
			mutex_enter(&target->sft_mutex);
			target->sft_lip_cnt = privp->lip_cnt;
			target->sft_state |= SF_TARGET_BUSY;
			target->sft_state &= ~(SF_TARGET_OFFLINE|
			    SF_TARGET_MARK);
			target->sft_al_pa = (uchar_t)privp->dest_nport_id;
			target->sft_hard_address = sf_switch_to_alpa[tnum];
			mutex_exit(&target->sft_mutex);
			mutex_exit(&sf->sf_mutex);
			if (ntarget != NULL)
				kmem_free(ntarget, sizeof (struct sf_target));
			return (target);
		}
		mutex_exit(&ptarget->sft_mutex);
		if (ntarget == NULL) {
			mutex_exit(&sf->sf_mutex);
			return (NULL);
		}
		/* Initialize new target structure */
		bcopy((caddr_t)&privp->node_wwn,
		    (caddr_t)&ntarget->sft_node_wwn, sizeof (privp->node_wwn));
		bcopy((caddr_t)&privp->port_wwn,
		    (caddr_t)&ntarget->sft_port_wwn, sizeof (privp->port_wwn));
		ntarget->sft_lun.l = lun;
#ifdef RAID_LUNS
```

```
                ntarget->sft_lun.l = orig_lun;
                ntarget->sft_raid_lun = (uint_t)lun;
#endif
                mutex_init(&ntarget->sft_mutex, NULL, MUTEX_DRIVER, NULL);
                mutex_init(&ntarget->sft_pkt_mutex, NULL, MUTEX_DRIVER, NULL);
                /* Don't let anyone use this till we finishup init. */
                mutex_enter(&ntarget->sft_mutex);
                mutex_enter(&ntarget->sft_pkt_mutex);

                hash = SF_HASH(privp->port_wwn, lun);
                ntarget->sft_next = sf->sf_wwn_lists[hash];
                sf->sf_wwn_lists[hash] = ntarget;

                ntarget->sft_lip_cnt = privp->lip_cnt;
                ntarget->sft_al_pa = (uchar_t)privp->dest_nport_id;
                ntarget->sft_hard_address = sf_switch_to_alpa[tnum];
                ntarget->sft_device_type = DTYPE_UNKNOWN;
                ntarget->sft_state = SF_TARGET_BUSY;
                ntarget->sft_pkt_head = (struct sf_pkt *)&ntarget->
                    sft_pkt_head;
                ntarget->sft_pkt_tail = (struct sf_pkt *)&ntarget->
                    sft_pkt_head;

                mutex_enter(&ptarget->sft_mutex);
                /* Traverse the list looking for this target */
                for (target = ptarget; target->sft_next_lun;
                    target = target->sft_next_lun) {
                        otarget = target->sft_next_lun;
                }
                ntarget->sft_next_lun = target->sft_next_lun;
                target->sft_next_lun = ntarget;
                mutex_exit(&ptarget->sft_mutex);
                mutex_exit(&ntarget->sft_pkt_mutex);
                mutex_exit(&ntarget->sft_mutex);
                mutex_exit(&sf->sf_mutex);
                return (ntarget);

        }
        if (target != NULL && target->sft_lip_cnt == sf->sf_lip_cnt) {
                /* It's been touched this LIP -- duplicate WWNs */
                sf_offline_target(sf, target); /* And all the baby targets */
                mutex_exit(&sf->sf_mutex);
                sf_log(sf, CE_WARN, "target 0x%x, duplicate port wwns\n",
                    tnum);
                if (ntarget != NULL) {
                        kmem_free(ntarget, sizeof (struct sf_target));
                }
                return (NULL);
        }

        if ((otarget = sf->sf_targets[tnum]) != NULL) {
                /* Someone else is in our slot */
                mutex_enter(&otarget->sft_mutex);
                if (otarget->sft_lip_cnt == sf->sf_lip_cnt) {
                        mutex_exit(&otarget->sft_mutex);
                        sf_offline_target(sf, otarget);
                        if (target != NULL)
                                sf_offline_target(sf, target);
                        mutex_exit(&sf->sf_mutex);
                        sf_log(sf, CE_WARN,
                            "target 0x%x, duplicate switch settings\n", tnum);
                        if (ntarget != NULL)
                                kmem_free(ntarget, sizeof (struct sf_target));
                        return (NULL);
                }
                mutex_exit(&otarget->sft_mutex);
```

```
                if (bcmp((caddr_t)&privp->port_wwn, (caddr_t)&otarget->
                    sft_port_wwn, sizeof (privp->port_wwn))) {
                        sf_offline_target(sf, otarget);
                        mutex_exit(&sf->sf_mutex);
                        sf_log(sf, CE_WARN, "wwn changed on target 0x%x\n",
                            tnum);
                        bzero((caddr_t)&sf->sf_stats.tstats[tnum],
                            sizeof (struct sf_target_stats));
                        mutex_enter(&sf->sf_mutex);
                }
        }

        sf->sf_targets[tnum] = target;
        if ((target = sf->sf_targets[tnum]) == NULL) {
                if (ntarget == NULL) {
                        mutex_exit(&sf->sf_mutex);
                        return (NULL);
                }
                bcopy((caddr_t)&privp->node_wwn,
                    (caddr_t)&ntarget->sft_node_wwn, sizeof (privp->node_wwn));
                bcopy((caddr_t)&privp->port_wwn,
                    (caddr_t)&ntarget->sft_port_wwn, sizeof (privp->port_wwn));
                ntarget->sft_lun.l = lun;
#ifdef RAID_LUNS
                ntarget->sft_lun.l = orig_lun;
                ntarget->sft_raid_lun = (uint_t)lun;
#endif
                mutex_init(&ntarget->sft_mutex, NULL, MUTEX_DRIVER, NULL);
                mutex_init(&ntarget->sft_pkt_mutex, NULL, MUTEX_DRIVER, NULL);
                mutex_enter(&ntarget->sft_mutex);
                mutex_enter(&ntarget->sft_pkt_mutex);
                hash = SF_HASH(privp->port_wwn, lun); /* lun 0 */
                ntarget->sft_next = sf->sf_wwn_lists[hash];
                sf->sf_wwn_lists[hash] = ntarget;

                target = ntarget;
                target->sft_lip_cnt = privp->lip_cnt;
                target->sft_al_pa = (uchar_t)privp->dest_nport_id;
                target->sft_hard_address = sf_switch_to_alpa[tnum];
                target->sft_device_type = DTYPE_UNKNOWN;
                target->sft_state = SF_TARGET_BUSY;
                target->sft_pkt_head = (struct sf_pkt *)&target->
                    sft_pkt_head;
                target->sft_pkt_tail = (struct sf_pkt *)&target->
                    sft_pkt_head;
                sf->sf_targets[tnum] = target;
                mutex_exit(&ntarget->sft_mutex);
                mutex_exit(&ntarget->sft_pkt_mutex);
                mutex_exit(&sf->sf_mutex);
        } else {
                mutex_enter(&target->sft_mutex);
                target->sft_lip_cnt = privp->lip_cnt;
                target->sft_state |= SF_TARGET_BUSY;
                target->sft_state &= ~(SF_TARGET_OFFLINE|SF_TARGET_MARK);
                target->sft_al_pa = (uchar_t)privp->dest_nport_id;
                target->sft_hard_address = sf_switch_to_alpa[tnum];
                mutex_exit(&target->sft_mutex);
                mutex_exit(&sf->sf_mutex);
                if (ntarget != NULL)
                        kmem_free(ntarget, sizeof (struct sf_target));
        }
        return (target);
}


/*
```

```
 * find the target for a given sf instance
 */
/* ARGSUSED */
static struct sf_target *
#ifdef RAID_LUNS
sf_lookup_target(struct sf *sf, uchar_t *wwn, int lun)
#else
sf_lookup_target(struct sf *sf, uchar_t *wwn, int64_t lun)
#endif
{
        int hash;
        struct sf_target *target;

        ASSERT(mutex_owned(&sf->sf_mutex));
        hash = SF_HASH(wwn, lun);

        target = sf->sf_wwn_lists[hash];
        while (target != NULL) {

#ifndef RAID_LUNS
                if (bcmp((caddr_t)wwn, (caddr_t)&target->sft_port_wwn,
                    sizeof (target->sft_port_wwn)) == 0 &&
                        target->sft_lun.l == lun)
                        break;
#else
                if (bcmp((caddr_t)wwn, (caddr_t)&target->sft_port_wwn,
                    sizeof (target->sft_port_wwn)) == 0 &&
                        target->sft_raid_lun == lun)
                        break;
#endif
                target = target->sft_next;
        }

        return (target);
}


/*
 * Send out a REPORT_LUNS command.
 */
static int
sf_do_reportlun(struct sf *sf, struct sf_els_hdr *privp,
    struct sf_target *target)
{
        struct  fcal_packet     *fpkt = privp->fpkt;
        ddi_dma_cookie_t        pcookie;
        ddi_dma_handle_t        lun_dma_handle = NULL;
        ddi_acc_handle_t        lun_acc_handle;
        uint_t                  ccount;
        size_t                  real_size;
        caddr_t                 lun_buf = NULL;
        int                     handle_bound = 0;
        fc_frame_header_t       *hp = &fpkt->fcal_socal_request.sr_fc_frame_hdr;
        struct fcp_cmd          *reportlun = (struct fcp_cmd *)privp->cmd;
        char                    *msg = "Transport";

        if (ddi_dma_alloc_handle(sf->sf_dip, sf->sf_sochandle->fcal_dmaattr,
            DDI_DMA_DONTWAIT, NULL, &lun_dma_handle) != DDI_SUCCESS) {
                msg = "ddi_dma_alloc_handle()";
                goto fail;
        }

        if (ddi_dma_mem_alloc(lun_dma_handle, REPORT_LUNS_SIZE,
            sf->sf_sochandle->fcal_accattr, DDI_DMA_CONSISTENT,
            DDI_DMA_DONTWAIT, NULL, &lun_buf,
            &real_size, &lun_acc_handle) != DDI_SUCCESS) {
```

```
                msg = "ddi_dma_mem_alloc()";
                goto fail;
        }

        if (real_size < REPORT_LUNS_SIZE) {
                msg = "DMA mem < REPORT_LUNS_SIZE";
                goto fail;
        }

        if (ddi_dma_addr_bind_handle(lun_dma_handle, NULL,
            lun_buf, real_size, DDI_DMA_READ |
            DDI_DMA_CONSISTENT, DDI_DMA_DONTWAIT,
            NULL, &pcookie, &ccount) != DDI_DMA_MAPPED) {
                msg = "ddi_dma_addr_bind_handle()";
                goto fail;
        }
        handle_bound = 1;

        if (ccount != 1) {
                msg = "ccount != 1";
                goto fail;
        }
        privp->els_code = 0;
        privp->target = target;
        privp->data_dma_handle = lun_dma_handle;
        privp->data_acc_handle = lun_acc_handle;
        privp->data_buf = lun_buf;

        fpkt->fcal_pkt_comp = sf_reportlun_callback;
        fpkt->fcal_socal_request.sr_soc_hdr.sh_seg_cnt = 3;
        fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_type = CQ_TYPE_IO_READ;
        fpkt->fcal_socal_request.sr_dataseg[0].fc_count =
            sizeof (struct fcp_cmd);
        fpkt->fcal_socal_request.sr_dataseg[2].fc_base =
            (uint32_t)pcookie.dmac_address;
        fpkt->fcal_socal_request.sr_dataseg[2].fc_count = pcookie.dmac_size;
        fpkt->fcal_socal_request.sr_soc_hdr.sh_byte_cnt = pcookie.dmac_size;
        hp->r_ctl = R_CTL_COMMAND;
        hp->type = TYPE_SCSI_FCP;
        bzero((caddr_t)reportlun, sizeof (struct fcp_cmd));
        ((union scsi_cdb *)reportlun->fcp_cdb)->scc_cmd = SCMD_REPORT_LUNS;
        /* Now set the buffer size.  If DDI gave us extra, that's O.K. */
        ((union scsi_cdb *)reportlun->fcp_cdb)->scc5_count0 =
            (real_size&0x0ff);
        ((union scsi_cdb *)reportlun->fcp_cdb)->scc5_count1 =
            (real_size>>8)&0x0ff;
        ((union scsi_cdb *)reportlun->fcp_cdb)->scc5_count2 =
            (real_size>>16)&0x0ff;
        ((union scsi_cdb *)reportlun->fcp_cdb)->scc5_count3 =
            (real_size>>24)&0x0ff;
        reportlun->fcp_cntl.cntl_read_data = 1;
        reportlun->fcp_cntl.cntl_write_data = 0;
        reportlun->fcp_data_len = pcookie.dmac_size;
        reportlun->fcp_cntl.cntl_qtype = FCP_QTYPE_SIMPLE;

        (void) ddi_dma_sync(lun_dma_handle, 0, 0, DDI_DMA_SYNC_FORDEV);
        /* We know he's there, so this should be fast */
        privp->timeout = sf_watchdog_time + SF_FCP_TIMEOUT;
        if (sf_els_transport(sf, privp) == 1)
                return (1);

fail:
        sf_log(sf, CE_WARN,
            "%s failure for REPORTLUN to target 0x%x\n",
            msg, sf_alpa_to_switch[privp->dest_nport_id]);
        sf_els_free(fpkt);
```

```
	if (lun_dma_handle != NULL) {
		if (handle_bound)
			(void) ddi_dma_unbind_handle(lun_dma_handle);
		ddi_dma_free_handle(&lun_dma_handle);
	}
	if (lun_buf != NULL) {
		ddi_dma_mem_free(&lun_acc_handle);
	}
	return (0);
}

/*
 * Handle the results of a REPORT_LUNS command:
 *	Create additional targets if necessary
 *	Initiate INQUIRYs on all LUNs.
 */
static void
sf_reportlun_callback(struct fcal_packet *fpkt)
{
	struct sf_els_hdr *privp = (struct sf_els_hdr *)fpkt->
	    fcal_pkt_private;
	struct scsi_report_luns *ptr =
	    (struct scsi_report_luns *)privp->data_buf;
	struct sf *sf = privp->sf;
	struct sf_target *target = privp->target;
	struct fcp_rsp *rsp = NULL;
	int delayed_retry = 0;
	int tid = sf_alpa_to_switch[target->sft_hard_address];
	int i, free_pkt = 1;
	short	ncmds;

	mutex_enter(&sf->sf_mutex);
	/* use as temporary state variable */
	if (privp->timeout == SF_INVALID_TIMEOUT) {
		mutex_exit(&sf->sf_mutex);
		return;
	}
	if (privp->prev)
		privp->prev->next = privp->next;
	if (privp->next)
		privp->next->prev = privp->prev;
	if (sf->sf_els_list == privp)
		sf->sf_els_list = privp->next;
	privp->prev = privp->next = NULL;
	mutex_exit(&sf->sf_mutex);
	ncmds = fpkt->fcal_ncmds;
	ASSERT(ncmds >= 0);
	mutex_enter(&sf->sf_cmd_mutex);
	sf->sf_ncmds = ncmds;
	mutex_exit(&sf->sf_cmd_mutex);

	if (fpkt->fcal_pkt_status == FCAL_STATUS_OK) {
		(void) ddi_dma_sync(privp->rsp_dma_handle, 0,
		    0, DDI_DMA_SYNC_FORKERNEL);

		rsp = (struct fcp_rsp *)privp->rsp;
	}
	SF_DEBUG(1, (sf, CE_CONT,
	    "!REPORTLUN to al_pa %x pkt status %x scsi status %x\n",
	    privp->dest_nport_id,
	    fpkt->fcal_pkt_status,
	    rsp?rsp->fcp_u.fcp_status.scsi_status:0));

		/* See if target simply does not support REPORT_LUNS. */
	if (rsp && rsp->fcp_u.fcp_status.scsi_status == STATUS_CHECK &&
	    rsp->fcp_u.fcp_status.sense_len_set &&
```

```
	    rsp->fcp_sense_len >=
		offsetof(struct scsi_extended_sense, es_qual_code)) {
			struct scsi_extended_sense *sense;
			sense = (struct scsi_extended_sense *)
			((caddr_t)rsp + sizeof (struct fcp_rsp)
				+ rsp->fcp_response_len);
			if (sense->es_key == KEY_ILLEGAL_REQUEST) {
				if (sense->es_add_code == 0x20) {
					/* Fake LUN 0 */
				SF_DEBUG(1, (sf, CE_CONT,
					"!REPORTLUN Faking good "
					"completion for alpa %x\n",
					privp->dest_nport_id));
					ptr->lun_list_len = FCP_LUN_SIZE;
					ptr->lun[0] = 0;
					rsp->fcp_u.fcp_status.scsi_status =
						STATUS_GOOD;
				} else if (sense->es_add_code == 0x25) {
					SF_DEBUG(1, (sf, CE_CONT,
					    "!REPORTLUN device alpa %x "
					    "key %x code %x\n",
					    privp->dest_nport_id,
					    sense->es_key, sense->es_add_code));
					    goto fail;
				}
			} else if (sense->es_key ==
				KEY_UNIT_ATTENTION &&
				sense->es_add_code == 0x29) {
				SF_DEBUG(1, (sf, CE_CONT,
					"!REPORTLUN device alpa %x was reset\n",
					privp->dest_nport_id));
			} else {
				SF_DEBUG(1, (sf, CE_CONT,
					"!REPORTLUN device alpa %x "
					"key %x code %x\n",
					privp->dest_nport_id,
					sense->es_key, sense->es_add_code));
/* XXXXXX The following is here to handle broken targets -- remove it later */
				if (sf_reportlun_forever &&
					sense->es_key == KEY_UNIT_ATTENTION)
					goto retry;
/* XXXXXX */
				if (sense->es_key == KEY_NOT_READY)
					delayed_retry = 1;
				}
		}

	if (rsp && rsp->fcp_u.fcp_status.scsi_status == STATUS_GOOD) {
		struct fcp_rsp_info *bep;

		bep = (struct fcp_rsp_info *)(&rsp->
		    fcp_response_len + 1);
		if (!rsp->fcp_u.fcp_status.rsp_len_set ||
		    bep->rsp_code == FCP_NO_FAILURE) {
			(void) ddi_dma_sync(privp->data_dma_handle,
			    0, 0, DDI_DMA_SYNC_FORKERNEL);

			/* Convert from #bytes to #ints */
			ptr->lun_list_len = ptr->lun_list_len >> 3;
			SF_DEBUG(2, (sf, CE_CONT,
			    "!REPORTLUN to al_pa %x succeeded: %d LUNs\n",
			    privp->dest_nport_id, ptr->lun_list_len));
			if (!ptr->lun_list_len) {
				/* No LUNs? Ya gotta be kidding... */
				sf_log(sf, CE_WARN,
				    "SCSI violation -- "
```

```
                                        "target 0x%x reports no LUNs\n",
                                        sf_alpa_to_switch[
                                        privp->dest_nport_id]);
                                ptr->lun_list_len = 1;
                                ptr->lun[0] = 0;
                        }

                        mutex_enter(&sf->sf_mutex);
                        if (sf->sf_lip_cnt == privp->lip_cnt) {
                                sf->sf_device_count += ptr->lun_list_len - 1;
                        }

                        mutex_exit(&sf->sf_mutex);
                        for (i = 0; i < ptr->lun_list_len && privp->lip_cnt ==
                            sf->sf_lip_cnt; i++) {
                                struct sf_els_hdr *nprivp;
                                struct fcal_packet *nfpkt;

                                /* LUN 0 is already in 'target' */
                                if (ptr->lun[i] != 0) {
                                        target = sf_create_target(sf,
                                            privp, tid, ptr->lun[i]);
                                }
                                nprivp = NULL;
                                nfpkt = NULL;
                                if (target) {
                                        nfpkt = sf_els_alloc(sf,
                                            target->sft_al_pa,
                                            sizeof (struct sf_els_hdr),
                                            sizeof (union sf_els_cmd),
                                            sizeof (union sf_els_rsp),
                                            (caddr_t *)&nprivp,
                                            (caddr_t *)&rsp);
                                        if (nprivp)
                                                nprivp->lip_cnt =
                                                    privp->lip_cnt;
                                }
                                if (nfpkt && nprivp &&
                                    (sf_do_inquiry(sf, nprivp, target) ==
                                    0)) {
                                        mutex_enter(&sf->sf_mutex);
                                        if (sf->sf_lip_cnt == privp->
                                            lip_cnt) {
                                                sf->sf_device_count --;
                                        }
                                        sf_offline_target(sf, target);
                                        mutex_exit(&sf->sf_mutex);
                                }
                        }
                        sf_els_free(fpkt);
                        return;
                } else {
                        SF_DEBUG(1, (sf, CE_CONT,
                            "!REPORTLUN al_pa %x fcp failure, "
                            "fcp_rsp_code %x scsi status %x\n",
                            privp->dest_nport_id, bep->rsp_code,
                            rsp ? rsp->fcp_u.fcp_status.scsi_status:0));
                        goto fail;
                }
        }
        if (rsp && ((rsp->fcp_u.fcp_status.scsi_status == STATUS_BUSY) ||
            (rsp->fcp_u.fcp_status.scsi_status == STATUS_QFULL))) {
                delayed_retry = 1;
        }

        if (++(privp->retries) < sf_els_retries ||
```

```
            (delayed_retry && privp->retries < SF_BSY_RETRIES)) {
/* XXXXXX The following is here to handle broken targets -- remove it later */
retry:
/* XXXXXX */
                if (delayed_retry) {
                        privp->retries--;
                        privp->timeout = sf_watchdog_time + SF_BSY_TIMEOUT;
                        privp->delayed_retry = 1;
                } else {
                        privp->timeout = sf_watchdog_time + SF_FCP_TIMEOUT;
                }

                privp->prev = NULL;
                mutex_enter(&sf->sf_mutex);
                if (privp->lip_cnt == sf->sf_lip_cnt) {
                        if (!delayed_retry)
                                SF_DEBUG(1, (sf, CE_WARN,
                                    "!REPORTLUN to al_pa %x failed, retrying\n",
                                    privp->dest_nport_id));
                        privp->next = sf->sf_els_list;
                        if (sf->sf_els_list != NULL)
                                sf->sf_els_list->prev = privp;
                        sf->sf_els_list = privp;
                        mutex_exit(&sf->sf_mutex);
                        if (!delayed_retry && soc_transport(sf->sf_sochandle,
                            fpkt, FCAL_NOSLEEP, CQ_REQUEST_1) !=
                            FCAL_TRANSPORT_SUCCESS) {
                                mutex_enter(&sf->sf_mutex);
                                if (privp->prev)
                                        privp->prev->next = privp->next;
                                if (privp->next)
                                        privp->next->prev = privp->prev;
                                if (sf->sf_els_list == privp)
                                        sf->sf_els_list = privp->next;
                                mutex_exit(&sf->sf_mutex);
                                goto fail;
                        } else
                                return;
                } else {
                        mutex_exit(&sf->sf_mutex);
                }
        } else {
fail:

                /* REPORT_LUN failed -- try inquiry */
                if (sf_do_inquiry(sf, privp, target) != 0) {
                        return;
                } else {
                        free_pkt = 0;
                }
                mutex_enter(&sf->sf_mutex);
                if (sf->sf_lip_cnt == privp->lip_cnt) {
                        sf_log(sf, CE_WARN,
                            "!REPORTLUN to target 0x%x failed\n",
                            sf_alpa_to_switch[privp->dest_nport_id]);
                        sf_offline_target(sf, target);
                        sf->sf_device_count--;
                        ASSERT(sf->sf_device_count >= 0);
                        if (sf->sf_device_count == 0)
                        sf_finish_init(sf, privp->lip_cnt);
                }
                mutex_exit(&sf->sf_mutex);
        }
        if (free_pkt) {
                sf_els_free(fpkt);
        }
```

```c
}

static int
sf_do_inquiry(struct sf *sf, struct sf_els_hdr *privp,
    struct sf_target *target)
{
        struct  fcal_packet     *fpkt = privp->fpkt;
        ddi_dma_cookie_t        pcookie;
        ddi_dma_handle_t        inq_dma_handle = NULL;
        ddi_acc_handle_t        inq_acc_handle;
        uint_t                  ccount;
        size_t                  real_size;
        caddr_t                 inq_buf = NULL;
        int                     handle_bound = FALSE;
        fc_frame_header_t *hp = &fpkt->fcal_socal_request.sr_fc_frame_hdr;
        struct fcp_cmd          *inq = (struct fcp_cmd *)privp->cmd;
        char                    *msg = "Transport";


        if (ddi_dma_alloc_handle(sf->sf_dip, sf->sf_sochandle->fcal_dmaattr,
            DDI_DMA_DONTWAIT, NULL, &inq_dma_handle) != DDI_SUCCESS) {
                msg = "ddi_dma_alloc_handle()";
                goto fail;
        }

        if (ddi_dma_mem_alloc(inq_dma_handle, SUN_INQSIZE,
            sf->sf_sochandle->fcal_accattr, DDI_DMA_CONSISTENT,
            DDI_DMA_DONTWAIT, NULL, &inq_buf,
            &real_size, &inq_acc_handle) != DDI_SUCCESS) {
                msg = "ddi_dma_mem_alloc()";
                goto fail;
        }

        if (real_size < SUN_INQSIZE) {
                msg = "DMA mem < inquiry size";
                goto fail;
        }

        if (ddi_dma_addr_bind_handle(inq_dma_handle, NULL,
            inq_buf, real_size, DDI_DMA_READ | DDI_DMA_CONSISTENT,
            DDI_DMA_DONTWAIT, NULL, &pcookie, &ccount) != DDI_DMA_MAPPED) {
                msg = "ddi_dma_addr_bind_handle()";
                goto fail;
        }
        handle_bound = TRUE;

        if (ccount != 1) {
                msg = "ccount != 1";
                goto fail;
        }
        privp->els_code = 0;                    /* not an ELS command */
        privp->target = target;
        privp->data_dma_handle = inq_dma_handle;
        privp->data_acc_handle = inq_acc_handle;
        privp->data_buf = inq_buf;
        fpkt->fcal_pkt_comp = sf_inq_callback;
        fpkt->fcal_socal_request.sr_soc_hdr.sh_seg_cnt = 3;
        fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_type = CQ_TYPE_IO_READ;
        fpkt->fcal_socal_request.sr_dataseg[0].fc_count =
            sizeof (struct fcp_cmd);
        fpkt->fcal_socal_request.sr_dataseg[2].fc_base =
            (uint32_t)pcookie.dmac_address;
        fpkt->fcal_socal_request.sr_dataseg[2].fc_count = pcookie.dmac_size;
        fpkt->fcal_socal_request.sr_soc_hdr.sh_byte_cnt = pcookie.dmac_size;
        hp->r_ctl = R_CTL_COMMAND;
        hp->type = TYPE_SCSI_FCP;
```

```c
        bzero((caddr_t)inq, sizeof (struct fcp_cmd));
        ((union scsi_cdb *)inq->fcp_cdb)->scc_cmd = SCMD_INQUIRY;
        ((union scsi_cdb *)inq->fcp_cdb)->g0_count0 = SUN_INQSIZE;
        bcopy((caddr_t)&target->sft_lun.b, (caddr_t)&inq->fcp_ent_addr,
            FCP_LUN_SIZE);
        inq->fcp_cntl.cntl_read_data = 1;
        inq->fcp_cntl.cntl_write_data = 0;
        inq->fcp_data_len = pcookie.dmac_size;
        inq->fcp_cntl.cntl_qtype = FCP_QTYPE_SIMPLE;

        (void) ddi_dma_sync(inq_dma_handle, (off_t)0, (size_t)0,
            DDI_DMA_SYNC_FORDEV);
        privp->timeout = sf_watchdog_time + SF_FCP_TIMEOUT;
        SF_DEBUG(5, (sf, CE_WARN,
            "!Sending INQUIRY to al_pa %x lun %" PRIx64 "\n",
            privp->dest_nport_id,
            SCSA_LUN(target)));
        return (sf_els_transport(sf, privp));

fail:
        sf_log(sf, CE_WARN,
            "%s failure for INQUIRY to target 0x%x\n",
            msg, sf_alpa_to_switch[privp->dest_nport_id]);
        sf_els_free(fpkt);
        if (inq_dma_handle != NULL) {
                if (handle_bound) {
                        (void) ddi_dma_unbind_handle(inq_dma_handle);
                }
                ddi_dma_free_handle(&inq_dma_handle);
        }
        if (inq_buf != NULL) {
                ddi_dma_mem_free(&inq_acc_handle);
        }
        return (FALSE);
}


/*
 * called as the pkt_comp routine for INQ packets
 */
static void
sf_inq_callback(struct fcal_packet *fpkt)
{
        struct sf_els_hdr *privp = (struct sf_els_hdr *)fpkt->
            fcal_pkt_private;
        struct scsi_inquiry *prt = (struct scsi_inquiry *)privp->data_buf;
        struct sf *sf = privp->sf;
        struct sf *tsf;
        struct sf_target *target = privp->target;
        struct fcp_rsp *rsp;
        int delayed_retry = FALSE;
        short   ncmds;


        mutex_enter(&sf->sf_mutex);
        /* use as temporary state variable */
        if (privp->timeout == SF_INVALID_TIMEOUT) {
                mutex_exit(&sf->sf_mutex);
                return;
        }
        if (privp->prev != NULL) {
                privp->prev->next = privp->next;
        }
        if (privp->next != NULL) {
                privp->next->prev = privp->prev;
        }
	if (sf->sf_els_list == privp) {
		sf->sf_els_list = privp->next;
	}
	privp->prev = privp->next = NULL;
	mutex_exit(&sf->sf_mutex);
	ncmds = fpkt->fcal_ncmds;
	ASSERT(ncmds >= 0);
	mutex_enter(&sf->sf_cmd_mutex);
	sf->sf_ncmds = ncmds;
	mutex_exit(&sf->sf_cmd_mutex);

	if (fpkt->fcal_pkt_status == FCAL_STATUS_OK) {

		(void) ddi_dma_sync(privp->rsp_dma_handle, (off_t)0,
		    (size_t)0, DDI_DMA_SYNC_FORKERNEL);

		rsp = (struct fcp_rsp *)privp->rsp;
		SF_DEBUG(2, (sf, CE_CONT,
		    "!INQUIRY to al_pa %x scsi status %x",
		    privp->dest_nport_id, rsp->fcp_u.fcp_status.scsi_status));

		if ((rsp->fcp_u.fcp_status.scsi_status == STATUS_GOOD) &&
		    !rsp->fcp_u.fcp_status.resid_over &&
		    (!rsp->fcp_u.fcp_status.resid_under ||
		    ((SUN_INQSIZE - rsp->fcp_resid) >= SUN_MIN_INQLEN))) {
			struct fcp_rsp_info *bep;

			bep = (struct fcp_rsp_info *)(&rsp->
			    fcp_response_len + 1);

			if (!rsp->fcp_u.fcp_status.rsp_len_set ||
			    (bep->rsp_code == FCP_NO_FAILURE)) {

				SF_DEBUG(2, (sf, CE_CONT,
				    "!INQUIRY to al_pa %x lun %" PRIx64
				    " succeeded\n",
				    privp->dest_nport_id, SCSA_LUN(target)));

				(void) ddi_dma_sync(privp->data_dma_handle,
				    (off_t)0, (size_t)0,
				    DDI_DMA_SYNC_FORKERNEL);

				mutex_enter(&sf->sf_mutex);

				if (sf->sf_lip_cnt == privp->lip_cnt) {
					mutex_enter(&target->sft_mutex);
					target->sft_device_type =
					    prt->inq_dtype;
					bcopy(prt, &target->sft_inq,
					    sizeof (*prt));
					mutex_exit(&target->sft_mutex);
					sf->sf_device_count--;
					ASSERT(sf->sf_device_count >= 0);
					if (sf->sf_device_count == 0) {
						sf_finish_init(sf,
						    privp->lip_cnt);
					}
				}
				mutex_exit(&sf->sf_mutex);
				sf_els_free(fpkt);
				return;
			}
		} else if ((rsp->fcp_u.fcp_status.scsi_status ==
		    STATUS_BUSY) ||
		    (rsp->fcp_u.fcp_status.scsi_status == STATUS_QFULL) ||
		    (rsp->fcp_u.fcp_status.scsi_status == STATUS_CHECK)) {
```

```
			delayed_retry = TRUE;
		}
	} else {
		SF_DEBUG(2, (sf, CE_CONT, "!INQUIRY to al_pa %x fc status %x",
		    privp->dest_nport_id, fpkt->fcal_pkt_status));
	}

	if (++(privp->retries) < sf_els_retries ||
	    (delayed_retry && privp->retries < SF_BSY_RETRIES)) {
		if (fpkt->fcal_pkt_status == FCAL_STATUS_MAX_XCHG_EXCEEDED)  {
			tsf = sf->sf_sibling;
			if (tsf != NULL) {
				mutex_enter(&tsf->sf_cmd_mutex);
				tsf->sf_flag = 1;
				tsf->sf_throttle = SF_DECR_DELTA;
				mutex_exit(&tsf->sf_cmd_mutex);
			}
			delayed_retry = 1;
		}
		if (delayed_retry) {
			privp->retries--;
			privp->timeout = sf_watchdog_time + SF_BSY_TIMEOUT;
			privp->delayed_retry = TRUE;
		} else {
			privp->timeout = sf_watchdog_time + SF_FCP_TIMEOUT;
		}

		privp->prev = NULL;
		mutex_enter(&sf->sf_mutex);
		if (privp->lip_cnt == sf->sf_lip_cnt) {
			if (!delayed_retry) {
				SF_DEBUG(1, (sf, CE_WARN,
				    "INQUIRY to al_pa %x failed, retrying",
				    privp->dest_nport_id));
			}
			privp->next = sf->sf_els_list;
			if (sf->sf_els_list != NULL) {
				sf->sf_els_list->prev = privp;
			}
			sf->sf_els_list = privp;
			mutex_exit(&sf->sf_mutex);
			/* if not delayed call transport to send a pkt */
			if (!delayed_retry &&
			    (soc_transport(sf->sf_sochandle, fpkt,
			    FCAL_NOSLEEP, CQ_REQUEST_1) !=
			    FCAL_TRANSPORT_SUCCESS)) {
				mutex_enter(&sf->sf_mutex);
				if (privp->prev != NULL) {
					privp->prev->next = privp->next;
				}
				if (privp->next != NULL) {
					privp->next->prev = privp->prev;
				}
				if (sf->sf_els_list == privp) {
					sf->sf_els_list = privp->next;
				}
				mutex_exit(&sf->sf_mutex);
				goto fail;
			}
			return;
		}
		mutex_exit(&sf->sf_mutex);
	} else {
fail:
		mutex_enter(&sf->sf_mutex);
		if (sf->sf_lip_cnt == privp->lip_cnt) {
```

```
                        sf_offline_target(sf, target);
                        sf_log(sf, CE_NOTE,
                            "INQUIRY to target 0x%x lun %" PRIx64 " failed. "
                            "Retry Count: %d\n",
                            sf_alpa_to_switch[privp->dest_nport_id],
                            SCSA_LUN(target),
                            privp->retries);
                        sf->sf_device_count--;
                        ASSERT(sf->sf_device_count >= 0);
                        if (sf->sf_device_count == 0) {
                                sf_finish_init(sf, privp->lip_cnt);
                        }
                }
                mutex_exit(&sf->sf_mutex);
        }
        sf_els_free(fpkt);
}


static void
sf_finish_init(struct sf *sf, int lip_cnt)
{
        int                     i;              /* loop index */
        int                     cflag;
        struct sf_target        *target;        /* current target */
        dev_info_t              *dip;
        struct sf_hp_elem       *elem;          /* hotplug element created */

        SF_DEBUG(1, (sf, CE_WARN, "!sf_finish_init\n"));
        ASSERT(mutex_owned(&sf->sf_mutex));

        /* scan all hash queues */
        for (i = 0; i < SF_NUM_HASH_QUEUES; i++) {
                target = sf->sf_wwn_lists[i];
                while (target != NULL) {
                        mutex_enter(&target->sft_mutex);

                        /* see if target is not offline */
                        if ((target->sft_state & SF_TARGET_OFFLINE)) {
                                /*
                                 * target already offline
                                 */
                                mutex_exit(&target->sft_mutex);
                                goto next_entry;
                        }

                        /*
                         * target is not already offline -- see if it has
                         * already been marked as ready to go offline
                         */
                        if (target->sft_state & SF_TARGET_MARK) {
                                /*
                                 * target already marked, so take it offline
                                 */
                                mutex_exit(&target->sft_mutex);
                                sf_offline_target(sf, target);
                                goto next_entry;
                        }

                        /* clear target busy flag */
                        target->sft_state &= ~SF_TARGET_BUSY;

                        /* is target init not yet done ?? */
                        cflag = !(target->sft_state & SF_TARGET_INIT_DONE);

                        /* get pointer to target dip */
```

```
                        dip = target->sft_dip;

                        mutex_exit(&target->sft_mutex);
                        mutex_exit(&sf->sf_mutex);

                        if (cflag && (dip == NULL)) {
                                /*
                                 * target init not yet done &&
                                 * devinfo not yet created
                                 */
                                sf_create_devinfo(sf, target, lip_cnt);
                                mutex_enter(&sf->sf_mutex);
                                goto next_entry;
                        }

                        /*
                         * target init already done || devinfo already created
                         */
                        ASSERT(dip != NULL);
                        if (!sf_create_props(dip, target, lip_cnt)) {
                                /* a problem creating properties */
                                mutex_enter(&sf->sf_mutex);
                                goto next_entry;
                        }

                        /* create a new element for the hotplug list */
                        if ((elem = kmem_zalloc(sizeof (struct sf_hp_elem),
                            KM_NOSLEEP)) != NULL) {

                                /* fill in the new element */
                                elem->dip = dip;
                                elem->target = target;
                                elem->what = SF_ONLINE;

                                /* add the new element into the hotplug list */
                                mutex_enter(&sf->sf_hp_daemon_mutex);
                                if (sf->sf_hp_elem_tail != NULL) {
                                        sf->sf_hp_elem_tail->next = elem;
                                        sf->sf_hp_elem_tail = elem;
                                } else {
                                        /* this is the first element in list */
                                        sf->sf_hp_elem_head =
                                            sf->sf_hp_elem_tail =
                                            elem;
                                }
                                cv_signal(&sf->sf_hp_daemon_cv);
                                mutex_exit(&sf->sf_hp_daemon_mutex);
                        } else {
                                /* could not allocate memory for element ?? */
                                (void) ndi_devi_online_async(dip, 0);
                        }

                        mutex_enter(&sf->sf_mutex);

next_entry:
                        /* ensure no new LIPs have occurred */
                        if (sf->sf_lip_cnt != lip_cnt) {
                                return;
                        }
                        target = target->sft_next;
                }

                /* done scanning all targets in this queue */
        }

        /* done with all hash queues */
        sf->sf_state = SF_STATE_ONLINE;
        sf->sf_online_timer = 0;
}


/*
 * create devinfo node
 */
static void
sf_create_devinfo(struct sf *sf, struct sf_target *target, int lip_cnt)
{
        dev_info_t              *cdip = NULL;
        char                    *nname = NULL;
        char                    **compatible = NULL;
        int                     ncompatible;
        struct scsi_inquiry     *inq = &target->sft_inq;
        char                    *scsi_binding_set;

        /* get the 'scsi-binding-set' property */
        if (ddi_prop_lookup_string(DDI_DEV_T_ANY, sf->sf_dip,
            DDI_PROP_NOTPROM | DDI_PROP_DONTPASS, "scsi-binding-set",
            &scsi_binding_set) != DDI_PROP_SUCCESS)
                scsi_binding_set = NULL;

        /* determine the node name and compatible */
        scsi_hba_nodename_compatible_get(inq, scsi_binding_set,
            inq->inq_dtype, NULL, &nname, &compatible, &ncompatible);
        if (scsi_binding_set)
                ddi_prop_free(scsi_binding_set);

        /* if nodename can't be determined then print a message and skip it */
        if (nname == NULL) {
#ifndef RAID_LUNS
                sf_log(sf, CE_WARN, "%s%d: no driver for device "
                    "@w%02x%02x%02x%02x%02x%02x%02x%02x,%x\n"
                    "    compatible: %s",
                    ddi_driver_name(sf->sf_dip), ddi_get_instance(sf->sf_dip),
                    target->sft_port_wwn[0], target->sft_port_wwn[1],
                    target->sft_port_wwn[2], target->sft_port_wwn[3],
                    target->sft_port_wwn[4], target->sft_port_wwn[5],
                    target->sft_port_wwn[6], target->sft_port_wwn[7],
                    target->sft_lun.l, *compatible);
#else
                sf_log(sf, CE_WARN, "%s%d: no driver for device "
                    "@w%02x%02x%02x%02x%02x%02x%02x%02x,%x\n"
                    "    compatible: %s",
                    ddi_driver_name(sf->sf_dip), ddi_get_instance(sf->sf_dip),
                    target->sft_port_wwn[0], target->sft_port_wwn[1],
                    target->sft_port_wwn[2], target->sft_port_wwn[3],
                    target->sft_port_wwn[4], target->sft_port_wwn[5],
                    target->sft_port_wwn[6], target->sft_port_wwn[7],
                    target->sft_raid_lun, *compatible);
#endif
                goto fail;
        }

        /* allocate the node */
        if (ndi_devi_alloc(sf->sf_dip, nname,
            DEVI_SID_NODEID, &cdip) != NDI_SUCCESS) {
                goto fail;
        }

        /* decorate the node with compatible */
        if (ndi_prop_update_string_array(DDI_DEV_T_NONE, cdip,
            "compatible", compatible, ncompatible) != DDI_PROP_SUCCESS) {
```

```
                goto fail;
        }

        /* add addressing properties to the node */
        if (sf_create_props(cdip, target, lip_cnt) != 1) {
                goto fail;
        }

        mutex_enter(&target->sft_mutex);
        if (target->sft_dip != NULL) {
                mutex_exit(&target->sft_mutex);
                goto fail;
        }
        target->sft_dip = cdip;
        mutex_exit(&target->sft_mutex);

        if (ndi_devi_online_async(cdip, 0) != DDI_SUCCESS) {
                goto fail;
        }

        scsi_hba_nodename_compatible_free(nname, compatible);
        return;

fail:
        scsi_hba_nodename_compatible_free(nname, compatible);
        if (cdip != NULL) {
                (void) ndi_prop_remove(DDI_DEV_T_NONE, cdip, NODE_WWN_PROP);
                (void) ndi_prop_remove(DDI_DEV_T_NONE, cdip, PORT_WWN_PROP);
                (void) ndi_prop_remove(DDI_DEV_T_NONE, cdip, LIP_CNT_PROP);
                (void) ndi_prop_remove(DDI_DEV_T_NONE, cdip, TARGET_PROP);
                (void) ndi_prop_remove(DDI_DEV_T_NONE, cdip, LUN_PROP);
                if (ndi_devi_free(cdip) != NDI_SUCCESS) {
                        sf_log(sf, CE_WARN, "ndi_devi_free failed\n");
                } else {
                        mutex_enter(&target->sft_mutex);
                        if (cdip == target->sft_dip) {
                                target->sft_dip = NULL;
                        }
                        mutex_exit(&target->sft_mutex);
                }
        }
}

/*
 * create required properties, returning TRUE iff we succeed, else
 * returning FALSE
 */
static int
sf_create_props(dev_info_t *cdip, struct sf_target *target, int lip_cnt)
{
        int tgt_id = sf_alpa_to_switch[target->sft_al_pa];


        if (ndi_prop_update_byte_array(DDI_DEV_T_NONE,
            cdip, NODE_WWN_PROP, target->sft_node_wwn, FC_WWN_SIZE) !=
            DDI_PROP_SUCCESS) {
                return (FALSE);
        }

        if (ndi_prop_update_byte_array(DDI_DEV_T_NONE,
            cdip, PORT_WWN_PROP, target->sft_port_wwn, FC_WWN_SIZE) !=
            DDI_PROP_SUCCESS) {
                return (FALSE);
        }

        if (ndi_prop_update_int(DDI_DEV_T_NONE,
            cdip, LIP_CNT_PROP, lip_cnt) != DDI_PROP_SUCCESS) {
                return (FALSE);
        }

        if (ndi_prop_update_int(DDI_DEV_T_NONE,
            cdip, TARGET_PROP, tgt_id) != DDI_PROP_SUCCESS) {
                return (FALSE);
        }

#ifndef RAID_LUNS
        if (ndi_prop_update_int(DDI_DEV_T_NONE,
            cdip, LUN_PROP, target->sft_lun.l) != DDI_PROP_SUCCESS) {
                return (0);
        }
#else
        if (ndi_prop_update_int(DDI_DEV_T_NONE,
            cdip, LUN_PROP, target->sft_raid_lun) != DDI_PROP_SUCCESS) {
                return (0);
        }
#endif

        return (TRUE);
}


/*
 * called by the transport to offline a target
 */
/* ARGSUSED */
static void
sf_offline_target(struct sf *sf, struct sf_target *target)
{
        dev_info_t *dip;
        struct sf_target *next_target = NULL;
        struct sf_hp_elem       *elem;

        ASSERT(mutex_owned(&sf->sf_mutex));

        if (sf_core && (sf_core & SF_CORE_OFFLINE_TARGET)) {
                (void) soc_take_core(sf->sf_sochandle, sf->sf_socp);
                sf_core = 0;
        }

        while (target != NULL) {
                sf_log(sf, CE_NOTE,
                    "!target 0x%x al_pa 0x%x lun %" PRIx64 " offlined\n",
                    sf_alpa_to_switch[target->sft_al_pa],
                    target->sft_al_pa, SCSA_LUN(target));
                mutex_enter(&target->sft_mutex);
                target->sft_state &= ~(SF_TARGET_BUSY|SF_TARGET_MARK);
                target->sft_state |= SF_TARGET_OFFLINE;
                mutex_exit(&target->sft_mutex);
                mutex_exit(&sf->sf_mutex);

                /* XXXX if this is LUN 0, offline all other LUNs */
                if (next_target || target->sft_lun.l == 0)
                        next_target = target->sft_next_lun;

                /* abort all cmds for this target */
                sf_abort_all(sf, target, FALSE, sf->sf_lip_cnt, FALSE);

                mutex_enter(&sf->sf_mutex);
                mutex_enter(&target->sft_mutex);
                if (target->sft_state & SF_TARGET_INIT_DONE) {
                        dip = target->sft_dip;
                        mutex_exit(&target->sft_mutex);
```

```
                        mutex_exit(&sf->sf_mutex);
                        (void) ndi_prop_remove(DDI_DEV_T_NONE, dip,
                            TARGET_PROP);
                        (void) ndi_event_retrieve_cookie(sf->sf_event_hdl,
                            dip, FCAL_REMOVE_EVENT, &sf_remove_eid,
                            NDI_EVENT_NOPASS);
                        (void) ndi_event_run_callbacks(sf->sf_event_hdl,
                            target->sft_dip, sf_remove_eid, NULL);

                        elem = kmem_zalloc(sizeof (struct sf_hp_elem),
                            KM_NOSLEEP);
                        if (elem != NULL) {
                                elem->dip = dip;
                                elem->target = target;
                                elem->what = SF_OFFLINE;
                                mutex_enter(&sf->sf_hp_daemon_mutex);
                                if (sf->sf_hp_elem_tail != NULL) {
                                        sf->sf_hp_elem_tail->next = elem;
                                        sf->sf_hp_elem_tail = elem;
                                } else {
                                        sf->sf_hp_elem_head =
                                            sf->sf_hp_elem_tail =
                                            elem;
                                }
                                cv_signal(&sf->sf_hp_daemon_cv);
                                mutex_exit(&sf->sf_hp_daemon_mutex);
                        } else {
                                /* don't do NDI_DEVI_REMOVE for now */
                                if (ndi_devi_offline(dip, 0) != NDI_SUCCESS) {
                                        SF_DEBUG(1, (sf, CE_WARN,
                                            "target %x lun %" PRIx64 ", "
                                            "device offline failed",
                                            sf_alpa_to_switch[target->
                                            sft_al_pa],
                                            SCSA_LUN(target)));
                                } else {
                                        SF_DEBUG(1, (sf, CE_NOTE,
                                            "target %x, lun %" PRIx64 ", "
                                            "device offline succeeded\n",
                                            sf_alpa_to_switch[target->
                                            sft_al_pa],
                                            SCSA_LUN(target)));
                                }
                        }
                        mutex_enter(&sf->sf_mutex);
                } else {
                        mutex_exit(&target->sft_mutex);
                }
                target = next_target;
        }
}


/*
 * routine to get/set a capability
 *
 * returning:
 *      1 (TRUE)        boolean capability is true (on get)
 *      0 (FALSE)       invalid capability, can't set capability (on set),
 *                      or boolean capability is false (on get)
 *      -1 (UNDEFINED)  can't find capability (SCSA) or unsupported capability
 *      3               when getting SCSI version number
 *      AL_PA           when getting port initiator ID
 */
static int
sf_commoncap(struct scsi_address *ap, char *cap,
    int val, int tgtonly, int doset)
{
        struct sf *sf = ADDR2SF(ap);
        int cidx;
        int rval = FALSE;


        if (cap == NULL) {
                SF_DEBUG(3, (sf, CE_WARN, "sf_commoncap: invalid arg"));
                return (rval);
        }

        /* get index of capability string */
        if ((cidx = scsi_hba_lookup_capstr(cap)) == -1) {
                /* can't find capability */
                return (UNDEFINED);
        }

        if (doset) {
                /*
                 * Process setcap request.
                 */

                /*
                 * At present, we can only set binary (0/1) values
                 */
                switch (cidx) {
                case SCSI_CAP_ARQ:      /* can't set this capability */
                        break;
                default:
                        SF_DEBUG(3, (sf, CE_WARN,
                            "sf_setcap: unsupported %d", cidx));
                        rval = UNDEFINED;
                        break;
                }

                SF_DEBUG(4, (sf, CE_NOTE,
                    "set cap: cap=%s,val=0x%x,tgtonly=0x%x"
                    ",doset=0x%x,rval=%d\n",
                    cap, val, tgtonly, doset, rval));

        } else {
                /*
                 * Process getcap request.
                 */
                switch (cidx) {
                case SCSI_CAP_DMA_MAX:
                        break;          /* don't' have this capability */
                case SCSI_CAP_INITIATOR_ID:
                        rval = sf->sf_al_pa;
                        break;
                case SCSI_CAP_ARQ:
                        rval = TRUE;    /* do have this capability */
                        break;
                case SCSI_CAP_RESET_NOTIFICATION:
                case SCSI_CAP_TAGGED_QING:
                        rval = TRUE;    /* do have this capability */
                        break;
                case SCSI_CAP_SCSI_VERSION:
                        rval = 3;
                        break;
                case SCSI_CAP_INTERCONNECT_TYPE:
                        rval = INTERCONNECT_FIBRE;
                        break;
                default:
                        SF_DEBUG(4, (sf, CE_WARN,
```

```
                            "sf_scsi_getcap: unsupported"));
                        rval = UNDEFINED;
                        break;
                }
                SF_DEBUG(4, (sf, CE_NOTE,
                    "get cap: cap=%s,val=0x%x,tgtonly=0x%x,"
                    "doset=0x%x,rval=%d\n",
                    cap, val, tgtonly, doset, rval));
        }

        return (rval);
}


/*
 * called by the transport to get a capability
 */
static int
sf_getcap(struct scsi_address *ap, char *cap, int whom)
{
        return (sf_commoncap(ap, cap, 0, whom, FALSE));
}


/*
 * called by the transport to set a capability
 */
static int
sf_setcap(struct scsi_address *ap, char *cap, int value, int whom)
{
        return (sf_commoncap(ap, cap, value, whom, TRUE));
}


/*
 * called by the transport to abort a target
 */
static int
sf_abort(struct scsi_address *ap, struct scsi_pkt *pkt)
{
        struct sf *sf = ADDR2SF(ap);
        struct sf_target *target = ADDR2TARGET(ap);
        struct sf_pkt *cmd, *ncmd, *pcmd;
        struct fcal_packet *fpkt;
        int     rval = 0, t, my_rval = FALSE;
        int     old_target_state;
        int     lip_cnt;
        int     tgt_id;
        fc_frame_header_t       *hp;
        int     deferred_destroy;

        deferred_destroy = 0;

        if (pkt != NULL) {
                cmd = PKT2CMD(pkt);
                fpkt = cmd->cmd_fp_pkt;
                SF_DEBUG(2, (sf, CE_NOTE, "sf_abort packet %p\n",
                    (void *)fpkt));
                pcmd = NULL;
                mutex_enter(&sf->sf_cmd_mutex);
                ncmd = sf->sf_pkt_head;
                while (ncmd != NULL) {
                        if (ncmd == cmd) {
                                if (pcmd != NULL) {
                                        pcmd->cmd_next = cmd->cmd_next;
                                } else {
```

```
					sf->sf_pkt_head = cmd->cmd_next;
				}
				cmd->cmd_flags &= ~CFLAG_IN_QUEUE;
				cmd->cmd_state = SF_STATE_IDLE;
				pkt->pkt_reason = CMD_ABORTED;
				pkt->pkt_statistics |= STAT_ABORTED;
				my_rval = TRUE;
				break;
			} else {
				pcmd = ncmd;
				ncmd = ncmd->cmd_next;
			}
		}
		mutex_exit(&sf->sf_cmd_mutex);
		if (ncmd == NULL) {
			mutex_enter(&cmd->cmd_abort_mutex);
			if (cmd->cmd_state == SF_STATE_ISSUED) {
				cmd->cmd_state = SF_STATE_ABORTING;
				cmd->cmd_timeout = sf_watchdog_time + 20;
				mutex_exit(&cmd->cmd_abort_mutex);
				/* call transport to abort command */
				if (((rval = soc_abort(sf->sf_sochandle,
				    sf->sf_socp, sf->sf_sochandle->fcal_portno,
				    fpkt, 1)) == FCAL_ABORTED) ||
				    (rval == FCAL_ABORT_FAILED)) {
					my_rval = TRUE;
					pkt->pkt_reason = CMD_ABORTED;
					pkt->pkt_statistics |= STAT_ABORTED;
					cmd->cmd_state = SF_STATE_IDLE;
				} else if (rval == FCAL_BAD_ABORT) {
					cmd->cmd_timeout = sf_watchdog_time
					    + 20;
					my_rval = FALSE;
				} else {
					SF_DEBUG(1, (sf, CE_NOTE,
					    "Command Abort failed\n"));
				}
			} else {
				mutex_exit(&cmd->cmd_abort_mutex);
			}
		}
	} else {
		SF_DEBUG(2, (sf, CE_NOTE, "sf_abort target\n"));
		mutex_enter(&sf->sf_mutex);
		lip_cnt = sf->sf_lip_cnt;
		mutex_enter(&target->sft_mutex);
		if (target->sft_state & (SF_TARGET_BUSY |
		    SF_TARGET_OFFLINE)) {
			mutex_exit(&target->sft_mutex);
			return (rval);
		}
		old_target_state = target->sft_state;
		target->sft_state |= SF_TARGET_BUSY;
		mutex_exit(&target->sft_mutex);
		mutex_exit(&sf->sf_mutex);

		if ((pkt = sf_scsi_init_pkt(ap, NULL, NULL, 0,
		    0, 0, 0, NULL, 0)) != NULL) {

			cmd = PKT2CMD(pkt);
			cmd->cmd_block->fcp_cntl.cntl_abort_tsk = 1;
			cmd->cmd_fp_pkt->fcal_pkt_comp = NULL;
			cmd->cmd_pkt->pkt_flags |= FLAG_NOINTR;

			/* prepare the packet for transport */
			if (sf_prepare_pkt(sf, cmd, target) == TRAN_ACCEPT) {
```

```

				cmd->cmd_state = SF_STATE_ISSUED;
				/*
				 * call transport to send a pkt polled
				 *
				 * if that fails call the transport to abort it
				 */
				if (soc_transport_poll(sf->sf_sochandle,
				    cmd->cmd_fp_pkt, SF_ABORT_TIMEOUT,
				    CQ_REQUEST_1) == FCAL_TRANSPORT_SUCCESS) {
					(void) ddi_dma_sync(
					    cmd->cmd_cr_pool->rsp_dma_handle,
					    (off_t)
					    ((caddr_t)cmd->cmd_rsp_block -
					    cmd->cmd_cr_pool->rsp_base),
					    FCP_MAX_RSP_IU_SIZE,
					    DDI_DMA_SYNC_FORKERNEL);
					if (((struct fcp_rsp_info *)
					    (&cmd->cmd_rsp_block->
					    fcp_response_len + 1))->
					    rsp_code == FCP_NO_FAILURE) {
						/* abort cmds for this targ */
						sf_abort_all(sf, target, TRUE,
						    lip_cnt, TRUE);
					} else {
						hp = &cmd->cmd_fp_pkt->
						    fcal_socal_request.
						    sr_fc_frame_hdr;
						tgt_id = sf_alpa_to_switch[
						    (uchar_t)hp->d_id];
						sf->sf_stats.tstats[tgt_id].
						    task_mgmt_failures++;
						SF_DEBUG(1, (sf, CE_NOTE,
						    "Target %d Abort Task "
						    "Set failed\n", hp->d_id));
					}
				} else {
					mutex_enter(&cmd->cmd_abort_mutex);
					if (cmd->cmd_state == SF_STATE_ISSUED) {
					cmd->cmd_state = SF_STATE_ABORTING;
					cmd->cmd_timeout = sf_watchdog_time
					    + 20;
					mutex_exit(&cmd->cmd_abort_mutex);
					if ((t = soc_abort(sf->sf_sochandle,
					    sf->sf_socp, sf->sf_sochandle->
					    fcal_portno, cmd->cmd_fp_pkt, 1)) !=
					    FCAL_ABORTED &&
					    (t != FCAL_ABORT_FAILED)) {
						sf_log(sf, CE_NOTE,
						    "sf_abort failed, "
						    "initiating LIP\n");
						sf_force_lip(sf);
						deferred_destroy = 1;
					}
					} else {
					mutex_exit(&cmd->cmd_abort_mutex);
					}
				}
			}
			if (!deferred_destroy) {
				cmd->cmd_fp_pkt->fcal_pkt_comp =
				    sf_cmd_callback;
				cmd->cmd_block->fcp_cntl.cntl_abort_tsk = 0;
				sf_scsi_destroy_pkt(ap, pkt);
				my_rval = TRUE;
			}
```

```
		}
		mutex_enter(&sf->sf_mutex);
		if (lip_cnt == sf->sf_lip_cnt) {
			mutex_enter(&target->sft_mutex);
			target->sft_state = old_target_state;
			mutex_exit(&target->sft_mutex);
		}
		mutex_exit(&sf->sf_mutex);
	}
	return (my_rval);
}


/*
 * called by the transport and internally to reset a target
 */
static int
sf_reset(struct scsi_address *ap, int level)
{
	struct scsi_pkt *pkt;
	struct fcal_packet *fpkt;
	struct sf *sf = ADDR2SF(ap);
	struct sf_target *target = ADDR2TARGET(ap), *ntarget;
	struct sf_pkt *cmd;
	int	rval = FALSE, t;
	int	lip_cnt;
	int	tgt_id, ret;
	fc_frame_header_t	*hp;
	int	deferred_destroy;

	/* We don't support RESET_LUN yet. */
	if (level == RESET_TARGET) {
		struct sf_reset_list *p;

		if ((p = kmem_alloc(sizeof (struct sf_reset_list), KM_NOSLEEP))
		    == NULL)
			return (rval);

		SF_DEBUG(2, (sf, CE_NOTE, "sf_reset target\n"));
		mutex_enter(&sf->sf_mutex);
		/* All target resets go to LUN 0 */
		if (target->sft_lun.l) {
			target = sf_lookup_target(sf, target->sft_port_wwn, 0);
		}
		mutex_enter(&target->sft_mutex);
		if (target->sft_state & (SF_TARGET_BUSY |
		    SF_TARGET_OFFLINE)) {
			mutex_exit(&target->sft_mutex);
			mutex_exit(&sf->sf_mutex);
			kmem_free(p, sizeof (struct sf_reset_list));
			return (rval);
		}
		lip_cnt = sf->sf_lip_cnt;
		target->sft_state |= SF_TARGET_BUSY;
		for (ntarget = target->sft_next_lun;
		    ntarget;
		    ntarget = ntarget->sft_next_lun) {
			mutex_enter(&ntarget->sft_mutex);
			/*
			 * XXXX If we supported RESET_LUN we should check here
			 * to see if any LUN were being reset and somehow fail
			 * that operation.
			 */
			ntarget->sft_state |= SF_TARGET_BUSY;
			mutex_exit(&ntarget->sft_mutex);
		}
```

```
		mutex_exit(&target->sft_mutex);
		mutex_exit(&sf->sf_mutex);

		deferred_destroy = 0;
		if ((pkt = sf_scsi_init_pkt(ap, NULL, NULL, 0,
		    0, 0, 0, NULL, 0)) != NULL) {
			cmd = PKT2CMD(pkt);
			cmd->cmd_block->fcp_cntl.cntl_reset = 1;
			cmd->cmd_fp_pkt->fcal_pkt_comp = NULL;
			cmd->cmd_pkt->pkt_flags |= FLAG_NOINTR;

			/* prepare the packet for transport */
			if (sf_prepare_pkt(sf, cmd, target) == TRAN_ACCEPT) {
				/* call transport to send a pkt polled */
				cmd->cmd_state = SF_STATE_ISSUED;
				if ((ret = soc_transport_poll(sf->sf_sochandle,
				    cmd->cmd_fp_pkt, SF_ABORT_TIMEOUT,
				    CQ_REQUEST_1)) == FCAL_TRANSPORT_SUCCESS) {
					(void) ddi_dma_sync(cmd->cmd_cr_pool->
					    rsp_dma_handle, (caddr_t)cmd->
					    cmd_rsp_block - cmd->cmd_cr_pool->
					    rsp_base, FCP_MAX_RSP_IU_SIZE,
					    DDI_DMA_SYNC_FORKERNEL);
					fpkt = cmd->cmd_fp_pkt;
					if ((fpkt->fcal_pkt_status ==
					    FCAL_STATUS_OK) &&
					    (((struct fcp_rsp_info *)
					    (&cmd->cmd_rsp_block->
					    fcp_response_len + 1))->
					    rsp_code == FCP_NO_FAILURE)) {
						sf_log(sf, CE_NOTE,
						    "!sf%d: Target 0x%x Reset "
						    "successful\n",
						    ddi_get_instance(\
						    sf->sf_dip),
						    sf_alpa_to_switch[
						    target->sft_al_pa]);
						rval = TRUE;
					} else {
						hp = &cmd->cmd_fp_pkt->
						    fcal_socal_request.
						    sr_fc_frame_hdr;
						tgt_id = sf_alpa_to_switch[
						    (uchar_t)hp->d_id];
						sf->sf_stats.tstats[tgt_id].
						    task_mgmt_failures++;
						sf_log(sf, CE_NOTE,
						    "!sf%d: Target 0x%x "
						    "Reset failed."
						    "Status code 0x%x "
						    "Resp code 0x%x\n",
						    ddi_get_instance(\
						    sf->sf_dip),
						    tgt_id,
						    fpkt->fcal_pkt_status,
						    ((struct fcp_rsp_info *)
						    (&cmd->cmd_rsp_block->
						    fcp_response_len + 1))->
						    rsp_code);
					}
				} else {
					sf_log(sf, CE_NOTE, "!sf%d: Target "
					    "0x%x Reset Failed. Ret=%x\n",
					    ddi_get_instance(sf->sf_dip),
					    sf_alpa_to_switch[
					    target->sft_al_pa], ret);
```

```
                                        mutex_enter(&cmd->cmd_abort_mutex);
                                        if (cmd->cmd_state == SF_STATE_ISSUED) {
                                        /* call the transport to abort a cmd */
                                        cmd->cmd_timeout = sf_watchdog_time
                                            + 20;
                                        cmd->cmd_state = SF_STATE_ABORTING;
                                        mutex_exit(&cmd->cmd_abort_mutex);
                                        if (((t = soc_abort(sf->sf_sochandle,
                                            sf->sf_socp,
                                            sf->sf_sochandle->fcal_portno,
                                            cmd->cmd_fp_pkt, 1)) !=
                                            FCAL_ABORTED) &&
                                            (t != FCAL_ABORT_FAILED)) {
                                                sf_log(sf, CE_NOTE,
                                                    "!sf%d: Target 0x%x Reset "
                                                    "failed. Abort Failed, "
                                                    "forcing LIP\n",
                                                    ddi_get_instance(
                                                    sf->sf_dip),
                                                    sf_alpa_to_switch[
                                                    target->sft_al_pa]);
                                                sf_force_lip(sf);
                                                rval = TRUE;
                                                deferred_destroy = 1;
                                        }
                                        } else {
                                                mutex_exit
                                                    (&cmd->cmd_abort_mutex);
                                        }
                                }
                        }
                        /*
                         * Defer releasing the packet if we abort returned with
                         * a BAD_ABORT or timed out, because there is a
                         * possibility that the ucode might return it.
                         * We wait for at least 20s and let it be released
                         * by the sf_watch thread
                         */
                        if (!deferred_destroy) {
                                cmd->cmd_block->fcp_cntl.cntl_reset = 0;
                                cmd->cmd_fp_pkt->fcal_pkt_comp =
                                    sf_cmd_callback;
                                cmd->cmd_state = SF_STATE_IDLE;
                                /* for cache */
                                sf_scsi_destroy_pkt(ap, pkt);
                        }
                } else {
                        cmn_err(CE_WARN, "!sf%d: Target 0x%x Reset Failed. "
                            "Resource allocation error.\n",
                            ddi_get_instance(sf->sf_dip),
                            sf_alpa_to_switch[target->sft_al_pa]);
                }
                mutex_enter(&sf->sf_mutex);
                if ((rval == TRUE) && (lip_cnt == sf->sf_lip_cnt)) {
                        p->target = target;
                        p->lip_cnt = lip_cnt;
                        p->timeout = ddi_get_lbolt() +
                            drv_usectohz(SF_TARGET_RESET_DELAY);
                        p->next = sf->sf_reset_list;
                        sf->sf_reset_list = p;
                        mutex_exit(&sf->sf_mutex);
                        mutex_enter(&sf_global_mutex);
                        if (sf_reset_timeout_id == 0) {
                                sf_reset_timeout_id = timeout(
                                    sf_check_reset_delay, NULL,
                                    drv_usectohz(SF_TARGET_RESET_DELAY));
```

```
                        }
                        mutex_exit(&sf_global_mutex);
                } else {
                        if (lip_cnt == sf->sf_lip_cnt) {
                                mutex_enter(&target->sft_mutex);
                                target->sft_state &= ~SF_TARGET_BUSY;
                                for (ntarget = target->sft_next_lun;
                                    ntarget;
                                    ntarget = ntarget->sft_next_lun) {
                                        mutex_enter(&ntarget->sft_mutex);
                                        ntarget->sft_state &= ~SF_TARGET_BUSY;
                                        mutex_exit(&ntarget->sft_mutex);
                                }
                                mutex_exit(&target->sft_mutex);
                        }
                        mutex_exit(&sf->sf_mutex);
                        kmem_free(p, sizeof (struct sf_reset_list));
                }
        } else {
                mutex_enter(&sf->sf_mutex);
                if ((sf->sf_state == SF_STATE_OFFLINE) &&
                    (sf_watchdog_time < sf->sf_timer)) {
                        /*
                         * We are currently in a lip, so let this one
                         * finish before forcing another one.
                         */
                        mutex_exit(&sf->sf_mutex);
                        return (TRUE);
                }
                mutex_exit(&sf->sf_mutex);
                sf_log(sf, CE_NOTE, "!sf:Target driver initiated lip\n");
                sf_force_lip(sf);
                rval = TRUE;
        }
        return (rval);
}


/*
 * abort all commands for a target
 *
 * if try_abort is set then send an abort
 * if abort is set then this is abort, else this is a reset
 */
static void
sf_abort_all(struct sf *sf, struct sf_target *target, int abort, int
    lip_cnt, int try_abort)
{
        struct sf_target *ntarget;
        struct sf_pkt *cmd, *head = NULL, *tail = NULL, *pcmd = NULL, *tcmd;
        struct fcal_packet *fpkt;
        struct scsi_pkt *pkt;
        int rval = FCAL_ABORTED;

        /*
         * First pull all commands for all LUNs on this target out of the
         * overflow list.  We can tell it's the same target by comparing
         * the node WWN.
         */
        mutex_enter(&sf->sf_mutex);
        if (lip_cnt == sf->sf_lip_cnt) {
                mutex_enter(&sf->sf_cmd_mutex);
                cmd = sf->sf_pkt_head;
                while (cmd != NULL) {
                        ntarget = ADDR2TARGET(&cmd->cmd_pkt->
                            pkt_address);
                        if (ntarget == target) {
                                if (pcmd != NULL)
                                        pcmd->cmd_next = cmd->cmd_next;
                                else
                                        sf->sf_pkt_head = cmd->cmd_next;
                                if (sf->sf_pkt_tail == cmd) {
                                        sf->sf_pkt_tail = pcmd;
                                        if (pcmd != NULL)
                                                pcmd->cmd_next = NULL;
                                }
                                tcmd = cmd->cmd_next;
                                if (head == NULL) {
                                        head = cmd;
                                        tail = cmd;
                                } else {
                                        tail->cmd_next = cmd;
                                        tail = cmd;
                                }
                                cmd->cmd_next = NULL;
                                cmd = tcmd;
                        } else {
                                pcmd = cmd;
                                cmd = cmd->cmd_next;
                        }
                }
                mutex_exit(&sf->sf_cmd_mutex);
        }
        mutex_exit(&sf->sf_mutex);

        /*
         * Now complete all the commands on our list.  In the process,
         * the completion routine may take the commands off the target
         * lists.
         */
        cmd = head;
        while (cmd != NULL) {
                pkt = cmd->cmd_pkt;
                if (abort) {
                        pkt->pkt_reason = CMD_ABORTED;
                        pkt->pkt_statistics |= STAT_ABORTED;
                } else {
                        pkt->pkt_reason = CMD_RESET;
                        pkt->pkt_statistics |= STAT_DEV_RESET;
                }
                cmd->cmd_flags &= ~CFLAG_IN_QUEUE;
                cmd->cmd_state = SF_STATE_IDLE;
                cmd = cmd->cmd_next;
                /*
                 * call the packet completion routine only for
                 * non-polled commands. Ignore the polled commands as
                 * they timeout and will be handled differently
                 */
                if ((pkt->pkt_comp) && !(pkt->pkt_flags & FLAG_NOINTR))
                        (*pkt->pkt_comp)(pkt);

        }

        /*
         * Finally get all outstanding commands for each LUN, and abort them if
         * they've been issued, and call the completion routine.
         * For the case where sf_offline_target is called from sf_watch
         * due to a Offline Timeout, it is quite possible that the soc+
         * ucode is hosed and therefore  cannot return the commands.
         * Clear up all the issued commands as well.
         * Try_abort will be false only if sf_abort_all is coming from
         * sf_target_offline.
```

```
         */

        if (try_abort || sf->sf_state == SF_STATE_OFFLINE) {
                mutex_enter(&target->sft_pkt_mutex);
                cmd = tcmd = target->sft_pkt_head;
                while (cmd != (struct sf_pkt *)&target->sft_pkt_head) {
                        fpkt = cmd->cmd_fp_pkt;
                        pkt = cmd->cmd_pkt;
                        mutex_enter(&cmd->cmd_abort_mutex);
                        if ((cmd->cmd_state == SF_STATE_ISSUED) &&
                            (fpkt->fcal_cmd_state &
                            FCAL_CMD_IN_TRANSPORT) &&
                            ((fpkt->fcal_cmd_state & FCAL_CMD_COMPLETE) ==
                            0) && !(pkt->pkt_flags & FLAG_NOINTR)) {
                                cmd->cmd_state = SF_STATE_ABORTING;
                                cmd->cmd_timeout = sf_watchdog_time +
                                    cmd->cmd_pkt->pkt_time + 20;
                                mutex_exit(&cmd->cmd_abort_mutex);
                                mutex_exit(&target->sft_pkt_mutex);
                                if (try_abort) {
                                        /* call the transport to abort a pkt */
                                        rval = soc_abort(sf->sf_sochandle,
                                            sf->sf_socp,
                                            sf->sf_sochandle->fcal_portno,
                                            fpkt, 1);
                                }
                                if ((rval == FCAL_ABORTED) ||
                                    (rval == FCAL_ABORT_FAILED)) {
                                        if (abort) {
                                                pkt->pkt_reason = CMD_ABORTED;
                                                pkt->pkt_statistics |=
                                                    STAT_ABORTED;
                                        } else {
                                                pkt->pkt_reason = CMD_RESET;
                                                pkt->pkt_statistics |=
                                                    STAT_DEV_RESET;
                                        }
                                        cmd->cmd_state = SF_STATE_IDLE;
                                        if (pkt->pkt_comp)
                                                (*pkt->pkt_comp)(pkt);
                                }
                                mutex_enter(&sf->sf_mutex);
                                if (lip_cnt != sf->sf_lip_cnt) {
                                        mutex_exit(&sf->sf_mutex);
                                        return;
                                }
                                mutex_exit(&sf->sf_mutex);
                                mutex_enter(&target->sft_pkt_mutex);
                                cmd = target->sft_pkt_head;
                        } else {
                                mutex_exit(&cmd->cmd_abort_mutex);
                                cmd = cmd->cmd_forw;
                        }
                }
                mutex_exit(&target->sft_pkt_mutex);
        }
}


/*
 * called by the transport to start a packet
 */
static int
sf_start(struct scsi_address *ap, struct scsi_pkt *pkt)
{
        struct sf *sf = ADDR2SF(ap);
	struct sf_target *target = ADDR2TARGET(ap);
	struct sf_pkt *cmd = PKT2CMD(pkt);
	int rval;


	SF_DEBUG(6, (sf, CE_NOTE, "sf_start\n"));

	if (cmd->cmd_state == SF_STATE_ISSUED) {
		cmn_err(CE_PANIC, "sf: issuing packet twice 0x%p\n",
		    (void *)cmd);
	}

	/* prepare the packet for transport */
	if ((rval = sf_prepare_pkt(sf, cmd, target)) != TRAN_ACCEPT) {
		return (rval);
	}

	if (target->sft_state & (SF_TARGET_BUSY|SF_TARGET_OFFLINE)) {
		if (target->sft_state & SF_TARGET_OFFLINE) {
			return (TRAN_FATAL_ERROR);
		}
		if (pkt->pkt_flags & FLAG_NOINTR) {
			return (TRAN_BUSY);
		}
		mutex_enter(&sf->sf_cmd_mutex);
		sf->sf_use_lock = TRUE;
		goto enque;
	}


	/* if no interrupts then do polled I/O */
	if (pkt->pkt_flags & FLAG_NOINTR) {
		return (sf_dopoll(sf, cmd));
	}

	/* regular interrupt-driven I/O */

	if (!sf->sf_use_lock) {

		/* locking no needed */

		cmd->cmd_timeout = cmd->cmd_pkt->pkt_time ?
		    sf_watchdog_time + cmd->cmd_pkt->pkt_time : 0;
		cmd->cmd_state = SF_STATE_ISSUED;

		/* call the transport to send a pkt */
		if (soc_transport(sf->sf_sochandle, cmd->cmd_fp_pkt,
		    FCAL_NOSLEEP, CQ_REQUEST_1) != FCAL_TRANSPORT_SUCCESS) {
			cmd->cmd_state = SF_STATE_IDLE;
			return (TRAN_BADPKT);
		}
		return (TRAN_ACCEPT);
	}

	/* regular I/O using locking */

	mutex_enter(&sf->sf_cmd_mutex);
	if ((sf->sf_ncmds >= sf->sf_throttle) ||
	    (sf->sf_pkt_head != NULL)) {
enque:
		/*
		 * either we're throttling back or there are already commands
		 * on the queue, so enqueue this one for later
		 */
		cmd->cmd_flags |= CFLAG_IN_QUEUE;
		if (sf->sf_pkt_head != NULL) {
```

```
			/* add to the queue */
			sf->sf_pkt_tail->cmd_next = cmd;
			cmd->cmd_next = NULL;
			sf->sf_pkt_tail = cmd;
		} else {
			/* this is the first entry in the queue */
			sf->sf_pkt_head = sf->sf_pkt_tail = cmd;
			cmd->cmd_next = NULL;
		}
		mutex_exit(&sf->sf_cmd_mutex);
		return (TRAN_ACCEPT);
	}

	/*
	 * start this packet now
	 */

	/* still have cmd mutex */
	return (sf_start_internal(sf, cmd));
}


/*
 * internal routine to start a packet from the queue now
 *
 * enter with cmd mutex held and leave with it released
 */
static int
sf_start_internal(struct sf *sf, struct sf_pkt *cmd)
{
	/* we have the cmd mutex */
	sf->sf_ncmds++;
	mutex_exit(&sf->sf_cmd_mutex);

	ASSERT(cmd->cmd_state != SF_STATE_ISSUED);
	SF_DEBUG(6, (sf, CE_NOTE, "sf_start_internal\n"));

	cmd->cmd_timeout = cmd->cmd_pkt->pkt_time ? sf_watchdog_time +
	    cmd->cmd_pkt->pkt_time : 0;
	cmd->cmd_state = SF_STATE_ISSUED;

	/* call transport to send the pkt */
	if (soc_transport(sf->sf_sochandle, cmd->cmd_fp_pkt, FCAL_NOSLEEP,
	    CQ_REQUEST_1) != FCAL_TRANSPORT_SUCCESS) {
		cmd->cmd_state = SF_STATE_IDLE;
		mutex_enter(&sf->sf_cmd_mutex);
		sf->sf_ncmds--;
		mutex_exit(&sf->sf_cmd_mutex);
		return (TRAN_BADPKT);
	}
	return (TRAN_ACCEPT);
}


/*
 * prepare a packet for transport
 */
static int
sf_prepare_pkt(struct sf *sf, struct sf_pkt *cmd, struct sf_target *target)
{
	struct fcp_cmd *fcmd = cmd->cmd_block;

/* XXXX Need to set the LUN ? */
	bcopy((caddr_t)&target->sft_lun.b,
	    (caddr_t)&fcmd->fcp_ent_addr,
	    FCP_LUN_SIZE);
	cmd->cmd_pkt->pkt_reason = CMD_CMPLT;
	cmd->cmd_pkt->pkt_state = 0;
	cmd->cmd_pkt->pkt_statistics = 0;


	if ((cmd->cmd_pkt->pkt_comp == NULL) &&
	    ((cmd->cmd_pkt->pkt_flags & FLAG_NOINTR) == 0)) {
		return (TRAN_BADPKT);
	}

	/* invalidate imp field(s) of rsp block */
	cmd->cmd_rsp_block->fcp_u.i_fcp_status = SF_BAD_DMA_MAGIC;

	/* set up amt of I/O to do */
	if (cmd->cmd_flags & CFLAG_DMAVALID) {
		cmd->cmd_pkt->pkt_resid = cmd->cmd_dmacount;
		if (cmd->cmd_flags & CFLAG_CMDIOPB) {
			(void) ddi_dma_sync(cmd->cmd_dmahandle, 0, 0,
			    DDI_DMA_SYNC_FORDEV);
		}
	} else {
		cmd->cmd_pkt->pkt_resid = 0;
	}

	/* set up the Tagged Queuing type */
	if (cmd->cmd_pkt->pkt_flags & FLAG_HTAG) {
		fcmd->fcp_cntl.cntl_qtype = FCP_QTYPE_HEAD_OF_Q;
	} else if (cmd->cmd_pkt->pkt_flags & FLAG_OTAG) {
		fcmd->fcp_cntl.cntl_qtype = FCP_QTYPE_ORDERED;
	}

	/*
	 * Sync the cmd segment
	 */
	(void) ddi_dma_sync(cmd->cmd_cr_pool->cmd_dma_handle,
	    (caddr_t)fcmd - cmd->cmd_cr_pool->cmd_base,
	    sizeof (struct fcp_cmd), DDI_DMA_SYNC_FORDEV);

	sf_fill_ids(sf, cmd, target);
	return (TRAN_ACCEPT);
}


/*
 * fill in packet hdr source and destination IDs and hdr byte count
 */
static void
sf_fill_ids(struct sf *sf, struct sf_pkt *cmd, struct sf_target *target)
{
	struct fcal_packet *fpkt = cmd->cmd_fp_pkt;
	fc_frame_header_t	*hp;


	hp = &fpkt->fcal_socal_request.sr_fc_frame_hdr;
	hp->d_id = target->sft_al_pa;
	hp->s_id = sf->sf_al_pa;
	fpkt->fcal_socal_request.sr_soc_hdr.sh_byte_cnt =
	    cmd->cmd_dmacookie.dmac_size;
}


/*
 * do polled I/O using transport
 */
static int
sf_dopoll(struct sf *sf, struct sf_pkt *cmd)
```

```
{
	int timeout;
	int rval;


	mutex_enter(&sf->sf_cmd_mutex);
	sf->sf_ncmds++;
	mutex_exit(&sf->sf_cmd_mutex);

	timeout = cmd->cmd_pkt->pkt_time ? cmd->cmd_pkt->pkt_time
	    : SF_POLL_TIMEOUT;
	cmd->cmd_timeout = 0;
	cmd->cmd_fp_pkt->fcal_pkt_comp = NULL;
	cmd->cmd_state = SF_STATE_ISSUED;

	/* call transport to send a pkt polled */
	rval = soc_transport_poll(sf->sf_sochandle, cmd->cmd_fp_pkt,
	    timeout*1000000, CQ_REQUEST_1);
	mutex_enter(&cmd->cmd_abort_mutex);
	cmd->cmd_fp_pkt->fcal_pkt_comp = sf_cmd_callback;
	if (rval != FCAL_TRANSPORT_SUCCESS) {
		if (rval == FCAL_TRANSPORT_TIMEOUT) {
			cmd->cmd_state = SF_STATE_ABORTING;
			mutex_exit(&cmd->cmd_abort_mutex);
			(void) sf_target_timeout(sf, cmd);
		} else {
			mutex_exit(&cmd->cmd_abort_mutex);
		}
		cmd->cmd_state = SF_STATE_IDLE;
		cmd->cmd_fp_pkt->fcal_pkt_comp = sf_cmd_callback;
		mutex_enter(&sf->sf_cmd_mutex);
		sf->sf_ncmds--;
		mutex_exit(&sf->sf_cmd_mutex);
		return (TRAN_BADPKT);
	}
	mutex_exit(&cmd->cmd_abort_mutex);
	cmd->cmd_fp_pkt->fcal_pkt_comp = sf_cmd_callback;
	sf_cmd_callback(cmd->cmd_fp_pkt);
	return (TRAN_ACCEPT);
}


/* a shortcut for defining debug messages below */
#ifdef	DEBUG
#define SF_DMSG1(s)		msg1 = s
#else
#define SF_DMSG1(s)		/* do nothing */
#endif


/*
 * the pkt_comp callback for command packets
 */
static void
sf_cmd_callback(struct fcal_packet *fpkt)
{
	struct sf_pkt *cmd = (struct sf_pkt *)fpkt->fcal_pkt_private;
	struct scsi_pkt *pkt = cmd->cmd_pkt;
	struct sf *sf = ADDR2SF(&pkt->pkt_address);
	struct sf_target *target = ADDR2TARGET(&pkt->pkt_address);
	struct fcp_rsp *rsp;
	char *msg1 = NULL;
	char *msg2 = NULL;
	short ncmds;
	int tgt_id;
	int good_scsi_status = TRUE;
```

```
        if (cmd->cmd_state == SF_STATE_IDLE) {
                cmn_err(CE_PANIC, "sf: completing idle packet 0x%p\n",
                    (void *)cmd);
        }

        mutex_enter(&cmd->cmd_abort_mutex);
        if (cmd->cmd_state == SF_STATE_ABORTING) {
                /* cmd already being aborted -- nothing to do */
                mutex_exit(&cmd->cmd_abort_mutex);
                return;
        }

        cmd->cmd_state = SF_STATE_IDLE;
        mutex_exit(&cmd->cmd_abort_mutex);

        if (fpkt->fcal_pkt_status == FCAL_STATUS_OK) {

                (void) ddi_dma_sync(cmd->cmd_cr_pool->rsp_dma_handle,
                    (caddr_t)cmd->cmd_rsp_block - cmd->cmd_cr_pool->rsp_base,
                    FCP_MAX_RSP_IU_SIZE, DDI_DMA_SYNC_FORKERNEL);

                rsp = (struct fcp_rsp *)cmd->cmd_rsp_block;

                if (rsp->fcp_u.i_fcp_status == SF_BAD_DMA_MAGIC) {

                        if (sf_core && (sf_core & SF_CORE_BAD_DMA)) {
                                sf_token = (int *)(uintptr_t)
                                    fpkt->fcal_socal_request.\
                                    sr_soc_hdr.sh_request_token;
                                (void) soc_take_core(sf->sf_sochandle,
                                    sf->sf_socp);
                        }

                        pkt->pkt_reason = CMD_INCOMPLETE;
                        pkt->pkt_state = STATE_GOT_BUS;
                        pkt->pkt_statistics |= STAT_ABORTED;

                } else {

                        pkt->pkt_state = STATE_GOT_BUS | STATE_GOT_TARGET |
                            STATE_SENT_CMD | STATE_GOT_STATUS;
                        pkt->pkt_resid = 0;
                        if (cmd->cmd_flags & CFLAG_DMAVALID) {
                                pkt->pkt_state |= STATE_XFERRED_DATA;
                        }

                        if ((pkt->pkt_scbp != NULL) &&
                            ((*(pkt->pkt_scbp) =
                            rsp->fcp_u.fcp_status.scsi_status)
                            != STATUS_GOOD)) {
                                good_scsi_status = FALSE;
                        /*
                         * The next two checks make sure that if there
                         * is no sense data or a valid response and
                         * the command came back with check condition,
                         * the command should be retried
                         */
                                if (!rsp->fcp_u.fcp_status.rsp_len_set &&
                                    !rsp->fcp_u.fcp_status.sense_len_set) {
                                        pkt->pkt_state &= ~STATE_XFERRED_DATA;
                                        pkt->pkt_resid = cmd->cmd_dmacount;
                                }
                        }
```

```
                        if ((cmd->cmd_flags & CFLAG_CMDIOPB) &&
                            (pkt->pkt_state & STATE_XFERRED_DATA)) {
                                (void) ddi_dma_sync(cmd->cmd_dmahandle, 0,
                                    (uint_t)0, DDI_DMA_SYNC_FORCPU);
                        }
                        /*
                         * Update the transfer resid, if appropriate
                         */
                        if (rsp->fcp_u.fcp_status.resid_over ||
                            rsp->fcp_u.fcp_status.resid_under)
                                pkt->pkt_resid = rsp->fcp_resid;

                        /*
                         * Check to see if the SCSI command failed.
                         *
                         */

                        /*
                         * First see if we got a FCP protocol error.
                         */
                        if (rsp->fcp_u.fcp_status.rsp_len_set) {
                                struct fcp_rsp_info *bep;

                                bep = (struct fcp_rsp_info *)
                                    (&rsp->fcp_response_len + 1);
                                if (bep->rsp_code != FCP_NO_FAILURE) {
                                                pkt->pkt_reason = CMD_TRAN_ERR;
                                        tgt_id = pkt->pkt_address.a_target;
                                        switch (bep->rsp_code) {
                                        case FCP_CMND_INVALID:
                                                SF_DMSG1("FCP_RSP FCP_CMND "
                                                    "fields invalid");
                                                break;
                                        case FCP_TASK_MGMT_NOT_SUPPTD:
                                                SF_DMSG1("FCP_RSP Task"
                                                    "Management Function"
                                                    "Not Supported");
                                                break;
                                        case FCP_TASK_MGMT_FAILED:
                                                SF_DMSG1("FCP_RSP Task "
                                                    "Management Function"
                                                    "Failed");
                                                sf->sf_stats.tstats[tgt_id].
                                                    task_mgmt_failures++;
                                                break;
                                        case FCP_DATA_RO_MISMATCH:
                                                SF_DMSG1("FCP_RSP FCP_DATA RO "
                                                    "mismatch with "
                                                    "FCP_XFER_RDY DATA_RO");
                                                sf->sf_stats.tstats[tgt_id].
                                                    data_ro_mismatches++;
                                                break;
                                        case FCP_DL_LEN_MISMATCH:
                                                SF_DMSG1("FCP_RSP FCP_DATA "
                                                    "length "
                                                    "different than BURST_LEN");
                                                sf->sf_stats.tstats[tgt_id].
                                                    dl_len_mismatches++;
                                                break;
                                        default:
                                                SF_DMSG1("FCP_RSP invalid "
                                                    "RSP_CODE");
                                                break;
                                        }
                                }
                        }

                        /*
                         * See if we got a SCSI error with sense data
                         */
                        if (rsp->fcp_u.fcp_status.sense_len_set) {
                                uchar_t rqlen = min(rsp->fcp_sense_len,
                                    sizeof (struct scsi_extended_sense));
                                caddr_t sense = (caddr_t)rsp +
                                    sizeof (struct fcp_rsp) +
                                    rsp->fcp_response_len;
                                struct scsi_arq_status *arq;
                                struct scsi_extended_sense *sensep =
                                    (struct scsi_extended_sense *)sense;

                                if (rsp->fcp_u.fcp_status.scsi_status !=
                                    STATUS_GOOD) {
                                if (rsp->fcp_u.fcp_status.scsi_status
                                    == STATUS_CHECK) {
                                        if (sensep->es_key ==
                                            KEY_RECOVERABLE_ERROR)
                                                good_scsi_status = 1;
                                        if (sensep->es_key ==
                                            KEY_UNIT_ATTENTION &&
                                            sensep->es_add_code == 0x3f &&
                                            sensep->es_qual_code == 0x0e) {
                                                /* REPORT_LUNS_HAS_CHANGED */
                                                sf_log(sf, CE_NOTE,
                                                "!REPORT_LUNS_HAS_CHANGED\n");
                                                sf_force_lip(sf);
                                        }
                                }
                                }

                                if ((pkt->pkt_scbp != NULL) &&
                                    (cmd->cmd_scblen >=
                                        sizeof (struct scsi_arq_status))) {

                                pkt->pkt_state |= STATE_ARQ_DONE;

                                arq = (struct scsi_arq_status *)pkt->pkt_scbp;
                                /*
                                 * copy out sense information
                                 */
                                bcopy(sense, (caddr_t)&arq->sts_sensedata,
                                    rqlen);
                                arq->sts_rqpkt_resid =
                                    sizeof (struct scsi_extended_sense) -
                                        rqlen;
                                *((uchar_t *)&arq->sts_rqpkt_status) =
                                    STATUS_GOOD;
                                arq->sts_rqpkt_reason = 0;
                                arq->sts_rqpkt_statistics = 0;
                                arq->sts_rqpkt_state = STATE_GOT_BUS |
                                    STATE_GOT_TARGET | STATE_SENT_CMD |
                                    STATE_GOT_STATUS | STATE_ARQ_DONE |
                                    STATE_XFERRED_DATA;
                            }
                                target->sft_alive = TRUE;
                        }

                        /*
                         * The firmware returns the number of bytes actually
                         * xfered into/out of host. Compare this with what
                         * we asked and if it is different, we lost frames ?
                         */
```

```
                        if ((pkt->pkt_reason == 0) && (pkt->pkt_resid == 0) &&
                            (good_scsi_status) &&
                            (pkt->pkt_state & STATE_XFERRED_DATA) &&
                            (!(cmd->cmd_flags & CFLAG_CMDIOPB)) &&
                            (target->sft_device_type != DTYPE_ESI)) {
                                int byte_cnt =
                                    fpkt->fcal_socal_request.
                                    sr_soc_hdr.sh_byte_cnt;
                                if (cmd->cmd_flags & CFLAG_DMASEND) {
                                        if (byte_cnt != 0) {
                                        sf_log(sf, CE_NOTE,
                                            "!sf_cmd_callback: Lost Frame: "
                                            "(write) received 0x%x expected"
                                            " 0x%x target 0x%x\n",
                                            byte_cnt, cmd->cmd_dmacount,
                                            sf_alpa_to_switch[
                                            target->sft_al_pa]);
                                        pkt->pkt_reason = CMD_INCOMPLETE;
                                        pkt->pkt_statistics |= STAT_ABORTED;
                                        }
                                } else if (byte_cnt < cmd->cmd_dmacount) {
                                        sf_log(sf, CE_NOTE,
                                            "!sf_cmd_callback: "
                                            "Lost Frame: (read) "
                                            "received 0x%x expected 0x%x "
                                            "target 0x%x\n", byte_cnt,
                                            cmd->cmd_dmacount,
                                            sf_alpa_to_switch[
                                            target->sft_al_pa]);
                                        pkt->pkt_reason = CMD_INCOMPLETE;
                                        pkt->pkt_statistics |= STAT_ABORTED;
                                }
                        }
                }

        } else {

                /* pkt status was not ok */

                switch (fpkt->fcal_pkt_status) {

                case FCAL_STATUS_ERR_OFFLINE:
                        SF_DMSG1("Fibre Channel Offline");
                        mutex_enter(&target->sft_mutex);
                        if (!(target->sft_state & SF_TARGET_OFFLINE)) {
                                target->sft_state |= (SF_TARGET_BUSY
                                    | SF_TARGET_MARK);
                        }
                        mutex_exit(&target->sft_mutex);
                        (void) ndi_event_retrieve_cookie(sf->sf_event_hdl,
                            target->sft_dip, FCAL_REMOVE_EVENT,
                            &sf_remove_eid, NDI_EVENT_NOPASS);
                        (void) ndi_event_run_callbacks(sf->sf_event_hdl,
                            target->sft_dip, sf_remove_eid, NULL);
                        pkt->pkt_reason = CMD_TRAN_ERR;
                        pkt->pkt_statistics |= STAT_BUS_RESET;
                        break;

                case FCAL_STATUS_MAX_XCHG_EXCEEDED:
                        sf_throttle(sf);
                        sf->sf_use_lock = TRUE;
                        pkt->pkt_reason = CMD_TRAN_ERR;
                        pkt->pkt_state = STATE_GOT_BUS;
                        pkt->pkt_statistics |= STAT_ABORTED;
                        break;
                case FCAL_STATUS_TIMEOUT:
                        SF_DMSG1("Fibre Channel Timeout");
                        pkt->pkt_reason = CMD_TIMEOUT;
                        break;

                case FCAL_STATUS_ERR_OVERRUN:
                        SF_DMSG1("CMD_DATA_OVR");
                        pkt->pkt_reason = CMD_DATA_OVR;
                        break;

                case FCAL_STATUS_UNKNOWN_CQ_TYPE:
                        SF_DMSG1("Unknown CQ type");
                        pkt->pkt_reason = CMD_TRAN_ERR;
                        break;

                case FCAL_STATUS_BAD_SEG_CNT:
                        SF_DMSG1("Bad SEG CNT");
                        pkt->pkt_reason = CMD_TRAN_ERR;
                        break;

                case FCAL_STATUS_BAD_XID:
                        SF_DMSG1("Fibre Channel Invalid X_ID");
                        pkt->pkt_reason = CMD_TRAN_ERR;
                        break;

                case FCAL_STATUS_XCHG_BUSY:
                        SF_DMSG1("Fibre Channel Exchange Busy");
                        pkt->pkt_reason = CMD_TRAN_ERR;
                        break;

                case FCAL_STATUS_INSUFFICIENT_CQES:
                        SF_DMSG1("Insufficient CQEs");
                        pkt->pkt_reason = CMD_TRAN_ERR;
                        break;

                case FCAL_STATUS_ALLOC_FAIL:
                        SF_DMSG1("ALLOC FAIL");
                        pkt->pkt_reason = CMD_TRAN_ERR;
                        break;

                case FCAL_STATUS_BAD_SID:
                        SF_DMSG1("Fibre Channel Invalid S_ID");
                        pkt->pkt_reason = CMD_TRAN_ERR;
                        break;

                case FCAL_STATUS_INCOMPLETE_DMA_ERR:
                        if (sf_core && (sf_core & SF_CORE_INCOMPLETE_DMA)) {
                                sf_token = (int *)(uintptr_t)
                                    fpkt->fcal_socal_request.\
                                    sr_soc_hdr.sh_request_token;
                                (void) soc_take_core(sf->sf_sochandle,
                                    sf->sf_socp);
                                sf_core = 0;
                        }
                        msg2 =
                        "INCOMPLETE DMA XFER due to bad SOC+ card, replace HBA";
                        pkt->pkt_reason = CMD_INCOMPLETE;
                        pkt->pkt_state = STATE_GOT_BUS;
                        pkt->pkt_statistics |= STAT_ABORTED;
                        break;

                case FCAL_STATUS_CRC_ERR:
                        msg2 = "Fibre Channel CRC Error on frames";
                        pkt->pkt_reason = CMD_INCOMPLETE;
                        pkt->pkt_state = STATE_GOT_BUS;
                        pkt->pkt_statistics |= STAT_ABORTED;
```

```
                        break;

                case FCAL_STATUS_NO_SEQ_INIT:
                        SF_DMSG1("Fibre Channel Seq Init Error");
                        pkt->pkt_reason = CMD_TRAN_ERR;
                        break;

                case  FCAL_STATUS_OPEN_FAIL:
                        pkt->pkt_reason = CMD_TRAN_ERR;
                        SF_DMSG1("Fibre Channel Open Failure");
                        if ((target->sft_state & (SF_TARGET_BUSY |
                            SF_TARGET_MARK | SF_TARGET_OFFLINE)) == 0) {
                                sf_log(sf, CE_NOTE,
                                    "!Open failure to target 0x%x "
                                    "forcing LIP\n",
                                    sf_alpa_to_switch[target->sft_al_pa]);
                                sf_force_lip(sf);
                        }
                        break;


                case FCAL_STATUS_ONLINE_TIMEOUT:
                        SF_DMSG1("Fibre Channel Online Timeout");
                        pkt->pkt_reason = CMD_TRAN_ERR;
                        break;

                default:
                        SF_DMSG1("Unknown FC Status");
                        pkt->pkt_reason = CMD_TRAN_ERR;
                        break;
                }
        }

#ifdef  DEBUG
        /*
         * msg1 will be non-NULL if we've detected some sort of error
         */
        if (msg1 != NULL && sfdebug >= 4) {
                sf_log(sf, CE_WARN,
                    "!Transport error on cmd=0x%p target=0x%x:  %s\n",
                    (void *)fpkt, pkt->pkt_address.a_target, msg1);
        }
#endif

        if (msg2 != NULL) {
                sf_log(sf, CE_WARN, "!Transport error on target=0x%x:  %s\n",
                    pkt->pkt_address.a_target, msg2);
        }

        ncmds = fpkt->fcal_ncmds;
        ASSERT(ncmds >= 0);
        if (ncmds >= (sf->sf_throttle - SF_HI_CMD_DELTA)) {
#ifdef DEBUG
                if (!sf->sf_use_lock) {
                        SF_DEBUG(4, (sf, CE_NOTE, "use lock flag on\n"));
                }
#endif
                sf->sf_use_lock = TRUE;
        }

        mutex_enter(&sf->sf_cmd_mutex);
        sf->sf_ncmds = ncmds;
        sf_throttle_start(sf);
        mutex_exit(&sf->sf_cmd_mutex);

        if (!msg1 && !msg2)
```

```
		SF_DEBUG(6, (sf, CE_NOTE, "Completing pkt 0x%p\n",
		    (void *)pkt));
	if (pkt->pkt_comp != NULL) {
		(*pkt->pkt_comp)(pkt);
	}
}

#undef	SF_DMSG1



/*
 * start throttling for this instance
 */
static void
sf_throttle_start(struct sf *sf)
{
	struct sf_pkt *cmd, *prev_cmd = NULL;
	struct scsi_pkt *pkt;
	struct sf_target *target;


	ASSERT(mutex_owned(&sf->sf_cmd_mutex));

	cmd = sf->sf_pkt_head;
	while ((cmd != NULL) &&
	    (sf->sf_state == SF_STATE_ONLINE) &&
	    (sf->sf_ncmds < sf->sf_throttle)) {

		pkt = CMD2PKT(cmd);

		target = ADDR2TARGET(&pkt->pkt_address);
		if (target->sft_state & SF_TARGET_BUSY) {
			/* this command is busy -- go to next */
			ASSERT(cmd->cmd_state != SF_STATE_ISSUED);
			prev_cmd = cmd;
			cmd = cmd->cmd_next;
			continue;
		}

		ASSERT(cmd->cmd_state != SF_STATE_ISSUED);

		/* this cmd not busy and not issued */

		/* remove this packet from the queue */
		if (sf->sf_pkt_head == cmd) {
			/* this was the first packet */
			sf->sf_pkt_head = cmd->cmd_next;
		} else if (sf->sf_pkt_tail == cmd) {
			/* this was the last packet */
			sf->sf_pkt_tail = prev_cmd;
			if (prev_cmd != NULL) {
				prev_cmd->cmd_next = NULL;
			}
		} else {
			/* some packet in the middle of the queue */
			ASSERT(prev_cmd != NULL);
			prev_cmd->cmd_next = cmd->cmd_next;
		}
		cmd->cmd_flags &= ~CFLAG_IN_QUEUE;

		if (target->sft_state & SF_TARGET_OFFLINE) {
			mutex_exit(&sf->sf_cmd_mutex);
			pkt->pkt_reason = CMD_TRAN_ERR;
			if (pkt->pkt_comp != NULL) {
				(*pkt->pkt_comp)(cmd->cmd_pkt);
```

```
			}
		} else {
			sf_fill_ids(sf, cmd, target);
			if (sf_start_internal(sf, cmd) != TRAN_ACCEPT) {
				pkt->pkt_reason = CMD_TRAN_ERR;
				if (pkt->pkt_comp != NULL) {
					(*pkt->pkt_comp)(cmd->cmd_pkt);
				}
			}
		}
		mutex_enter(&sf->sf_cmd_mutex);
		cmd = sf->sf_pkt_head;
		prev_cmd = NULL;
	}
}


/*
 * called when the max exchange value is exceeded to throttle back commands
 */
static void
sf_throttle(struct sf *sf)
{
	int cmdmax = sf->sf_sochandle->fcal_cmdmax;


	mutex_enter(&sf->sf_cmd_mutex);

	sf->sf_flag = TRUE;

	if (sf->sf_ncmds > (cmdmax / 2)) {
		sf->sf_throttle = cmdmax / 2;
	} else {
		if (sf->sf_ncmds > SF_DECR_DELTA) {
			sf->sf_throttle = sf->sf_ncmds - SF_DECR_DELTA;
		} else {
			/*
			 * This case is just a safeguard, should not really
			 * happen(ncmds < SF_DECR_DELTA and MAX_EXCHG exceed
			 */
			sf->sf_throttle = SF_DECR_DELTA;
		}
	}
	mutex_exit(&sf->sf_cmd_mutex);

	sf = sf->sf_sibling;
	if (sf != NULL) {
		mutex_enter(&sf->sf_cmd_mutex);
		sf->sf_flag = TRUE;
		if (sf->sf_ncmds >= (cmdmax / 2)) {
			sf->sf_throttle = cmdmax / 2;
		} else {
			if (sf->sf_ncmds > SF_DECR_DELTA) {
				sf->sf_throttle = sf->sf_ncmds - SF_DECR_DELTA;
			} else {
				sf->sf_throttle = SF_DECR_DELTA;
			}
		}

		mutex_exit(&sf->sf_cmd_mutex);
	}
}


/*
 * sf watchdog routine, called for a timeout
 */
/*ARGSUSED*/
static void
sf_watch(void *arg)
{
        struct sf *sf;
        struct sf_els_hdr       *privp;
        static int count = 0, pscan_count = 0;
        int cmdmax, i, mescount = 0;
        struct sf_target *target;


        sf_watchdog_time += sf_watchdog_timeout;
        count++;
        pscan_count++;

        mutex_enter(&sf_global_mutex);
        sf_watch_running = 1;
        for (sf = sf_head; sf != NULL; sf = sf->sf_next) {

                mutex_exit(&sf_global_mutex);

                /* disable throttling while we're suspended */
                mutex_enter(&sf->sf_mutex);
                if (sf->sf_state & SF_STATE_SUSPENDED) {
                        mutex_exit(&sf->sf_mutex);
                        SF_DEBUG(1, (sf, CE_CONT,
                            "sf_watch, sf%d:throttle disabled "
                            "due to DDI_SUSPEND\n",
                            ddi_get_instance(sf->sf_dip)));
                        mutex_enter(&sf_global_mutex);
                        continue;
                }
                mutex_exit(&sf->sf_mutex);

                cmdmax = sf->sf_sochandle->fcal_cmdmax;

                if (sf->sf_take_core) {
                        (void) soc_take_core(sf->sf_sochandle, sf->sf_socp);
                }

                mutex_enter(&sf->sf_cmd_mutex);

                if (!sf->sf_flag) {
                        if (sf->sf_throttle < (cmdmax / 2)) {
                                sf->sf_throttle = cmdmax / 2;
                        } else if ((sf->sf_throttle += SF_INCR_DELTA) >
                            cmdmax) {
                                sf->sf_throttle = cmdmax;
                        }
                } else {
                        sf->sf_flag = FALSE;
                }

                sf->sf_ncmds_exp_avg = (sf->sf_ncmds + sf->sf_ncmds_exp_avg)
                    >> 2;
                if ((sf->sf_ncmds <= (sf->sf_throttle - SF_LO_CMD_DELTA)) &&
                    (sf->sf_pkt_head == NULL)) {
#ifdef DEBUG
                        if (sf->sf_use_lock) {
                                SF_DEBUG(4, (sf, CE_NOTE,
                                    "use lock flag off\n"));
                        }
#endif
                        sf->sf_use_lock = FALSE;
                }
```

```

                if (sf->sf_state == SF_STATE_ONLINE && sf->sf_pkt_head &&
                    sf->sf_ncmds < sf->sf_throttle) {
                        sf_throttle_start(sf);
                }

                mutex_exit(&sf->sf_cmd_mutex);

                if (pscan_count >= sf_pool_scan_cnt) {
                        if (sf->sf_ncmds_exp_avg < (sf->sf_cr_pool_cnt <<
                            SF_LOG2_ELEMS_IN_POOL) - SF_FREE_CR_EPSILON) {
                                sf_crpool_free(sf);
                        }
                }
                mutex_enter(&sf->sf_mutex);

                privp = sf->sf_els_list;
                while (privp != NULL) {
                        if (privp->timeout < sf_watchdog_time) {
                                /* timeout this command */
                                privp = sf_els_timeout(sf, privp);
                        } else if ((privp->timeout == SF_INVALID_TIMEOUT) &&
                            (privp->lip_cnt != sf->sf_lip_cnt)) {
                                if (privp->prev != NULL) {
                                        privp->prev->next = privp->next;
                                }
                                if (sf->sf_els_list == privp) {
                                        sf->sf_els_list = privp->next;
                                }
                                if (privp->next != NULL) {
                                        privp->next->prev = privp->prev;
                                }
                                mutex_exit(&sf->sf_mutex);
                                sf_els_free(privp->fpkt);
                                mutex_enter(&sf->sf_mutex);
                                privp = sf->sf_els_list;
                        } else {
                                privp = privp->next;
                        }
                }

                if (sf->sf_online_timer && sf->sf_online_timer <
                    sf_watchdog_time) {
                        for (i = 0; i < sf_max_targets; i++) {
                                target = sf->sf_targets[i];
                                if (target != NULL) {
                                        if (!mescount && target->sft_state &
                                            SF_TARGET_BUSY) {
                                                sf_log(sf, CE_WARN, "!Loop "
                                                    "Unstable: Failed to bring "
                                                    "Loop Online\n");
                                                mescount = 1;
                                        }
                                        target->sft_state |= SF_TARGET_MARK;
                                }
                        }
                        sf_finish_init(sf, sf->sf_lip_cnt);
                        sf->sf_state = SF_STATE_INIT;
                        sf->sf_online_timer = 0;
                }

                if (sf->sf_state == SF_STATE_ONLINE) {
                        mutex_exit(&sf->sf_mutex);
                        if (count >= sf_pkt_scan_cnt) {
                                sf_check_targets(sf);
                        }
```

```c
                } else if ((sf->sf_state == SF_STATE_OFFLINE) &&
                    (sf->sf_timer < sf_watchdog_time)) {
                        for (i = 0; i < sf_max_targets; i++) {
                                target = sf->sf_targets[i];
                                if ((target != NULL) &&
                                    (target->sft_state &
                                    SF_TARGET_BUSY)) {
                                        sf_log(sf, CE_WARN,
                                            "!Offline Timeout\n");
                                        if (sf_core && (sf_core &
                                            SF_CORE_OFFLINE_TIMEOUT)) {
                                                (void) soc_take_core(
                                                    sf->sf_sochandle,
                                                    sf->sf_socp);
                                                sf_core = 0;
                                        }
                                        break;
                                }
                        }
                        sf_finish_init(sf, sf->sf_lip_cnt);
                        sf->sf_state = SF_STATE_INIT;
                        mutex_exit(&sf->sf_mutex);
                } else {
                        mutex_exit(&sf->sf_mutex);
                }
                mutex_enter(&sf_global_mutex);
        }
        mutex_exit(&sf_global_mutex);
        if (count >= sf_pkt_scan_cnt) {
                count = 0;
        }
        if (pscan_count >= sf_pool_scan_cnt) {
                pscan_count = 0;
        }

        /* reset timeout */
        sf_watchdog_id = timeout(sf_watch, (caddr_t)0, sf_watchdog_tick);

        /* signal waiting thread */
        mutex_enter(&sf_global_mutex);
        sf_watch_running = 0;
        cv_broadcast(&sf_watch_cv);
        mutex_exit(&sf_global_mutex);
}


/*
 * called during a timeout to check targets
 */
static void
sf_check_targets(struct sf *sf)
{
        struct sf_target *target;
        int i;
        struct sf_pkt *cmd;
        struct scsi_pkt *pkt;
        int lip_cnt;

        mutex_enter(&sf->sf_mutex);
        lip_cnt = sf->sf_lip_cnt;
        mutex_exit(&sf->sf_mutex);

        /* check scan all possible targets */
        for (i = 0; i < sf_max_targets; i++) {
                target = sf->sf_targets[i];
                while (target != NULL) {
```

```c
                        mutex_enter(&target->sft_pkt_mutex);
                        if (target->sft_alive && target->sft_scan_count !=
                            sf_target_scan_cnt) {
                                target->sft_alive = 0;
                                target->sft_scan_count++;
                                mutex_exit(&target->sft_pkt_mutex);
                                return;
                        }
                        target->sft_alive = 0;
                        target->sft_scan_count = 0;
                        cmd = target->sft_pkt_head;
                        while (cmd != (struct sf_pkt *)&target->sft_pkt_head) {
                                mutex_enter(&cmd->cmd_abort_mutex);
                                if (cmd->cmd_state == SF_STATE_ISSUED &&
                                    ((cmd->cmd_timeout && sf_watchdog_time >
#ifdef  DEBUG
                                    cmd->cmd_timeout) || sf_abort_flag)) {
                                        sf_abort_flag = 0;
#else
                                        cmd->cmd_timeout))) {
#endif
                                        cmd->cmd_timeout = 0;
        /* prevent reset from getting at this packet */
                                        cmd->cmd_state = SF_STATE_ABORTING;
                                        mutex_exit(&cmd->cmd_abort_mutex);
                                        mutex_exit(&target->sft_pkt_mutex);
                                        sf->sf_stats.tstats[i].timeouts++;
                                        if (sf_target_timeout(sf, cmd))
                                                return;
                                        else {
                                                if (lip_cnt != sf->sf_lip_cnt) {
                                                        return;
                                                } else {
                                                        mutex_enter(&target->
                                                            sft_pkt_mutex);
                                                        cmd = target->
                                                            sft_pkt_head;
                                                }
                                        }
        /*
         * if the abort and lip fail, a reset will be carried out.
         * But the reset will ignore this packet. We have waited at least
         * 20 seconds after the initial timeout. Now, complete it here.
         * This also takes care of spurious bad aborts.
         */
                                } else if ((cmd->cmd_state ==
                                    SF_STATE_ABORTING) && (cmd->cmd_timeout
                                    <= sf_watchdog_time)) {
                                        cmd->cmd_state = SF_STATE_IDLE;
                                        mutex_exit(&cmd->cmd_abort_mutex);
                                        mutex_exit(&target->sft_pkt_mutex);
                                        SF_DEBUG(1, (sf, CE_NOTE,
                                            "Command 0x%p to sft 0x%p"
                                            " delayed release\n",
                                            (void *)cmd, (void *)target));
                                        pkt = cmd->cmd_pkt;
                                        pkt->pkt_statistics |=
                                            (STAT_TIMEOUT|STAT_ABORTED);
                                        pkt->pkt_reason = CMD_TIMEOUT;
                                        if (pkt->pkt_comp) {
                                                scsi_hba_pkt_comp(pkt);
                                        /* handle deferred_destroy case */
                                        } else {
                                                if ((cmd->cmd_block->fcp_cntl.
                                                    cntl_reset == 1) ||
                                                    (cmd->cmd_block->
                                                    fcp_cntl.cntl_abort_tsk ==
                                                    1)) {
                                                        cmd->cmd_block->
                                                            fcp_cntl.
                                                            cntl_reset = 0;
                                                        cmd->cmd_block->
                                                            fcp_cntl.
                                                            cntl_abort_tsk = 0;
                                                        cmd->cmd_fp_pkt->
                                                            fcal_pkt_comp =
                                                            sf_cmd_callback;
                                                        /* for cache */
                                                        sf_scsi_destroy_pkt
                                                            (&pkt->pkt_address,
                                                            pkt);
                                                }
                                        }
                                        mutex_enter(&target->sft_pkt_mutex);
                                        cmd = target->sft_pkt_head;
                                } else {
                                        mutex_exit(&cmd->cmd_abort_mutex);
                                        cmd = cmd->cmd_forw;
                                }
                        }
                        mutex_exit(&target->sft_pkt_mutex);
                        target = target->sft_next_lun;
                }
        }
}


/*
 * a command to a target has timed out
 * return TRUE iff cmd abort failed or timed out, else return FALSE
 */
static int
sf_target_timeout(struct sf *sf, struct sf_pkt *cmd)
{
        int rval;
        struct scsi_pkt *pkt;
        struct fcal_packet *fpkt;
        int tgt_id;
        int retval = FALSE;


        SF_DEBUG(1, (sf, CE_NOTE, "Command 0x%p to target %x timed out\n",
            (void *)cmd->cmd_fp_pkt, cmd->cmd_pkt->pkt_address.a_target));

        fpkt = cmd->cmd_fp_pkt;

        if (sf_core && (sf_core & SF_CORE_CMD_TIMEOUT)) {
                sf_token = (int *)(uintptr_t)
                    fpkt->fcal_socal_request.sr_soc_hdr.\
                    sh_request_token;
                (void) soc_take_core(sf->sf_sochandle, sf->sf_socp);
                sf_core = 0;
        }

        /* call the transport to abort a command */
        rval = soc_abort(sf->sf_sochandle, sf->sf_socp,
            sf->sf_sochandle->fcal_portno, fpkt, 1);

        switch (rval) {
        case FCAL_ABORTED:
                SF_DEBUG(1, (sf, CE_NOTE, "Command Abort succeeded\n"));
                pkt = cmd->cmd_pkt;
```

```
                cmd->cmd_state = SF_STATE_IDLE;
                pkt->pkt_statistics |= (STAT_TIMEOUT|STAT_ABORTED);
                pkt->pkt_reason = CMD_TIMEOUT;
                if (pkt->pkt_comp != NULL) {
                        (*pkt->pkt_comp)(pkt);
                }
                break;                          /* success */

        case FCAL_ABORT_FAILED:
                SF_DEBUG(1, (sf, CE_NOTE, "Command Abort failed at target\n"));
                pkt = cmd->cmd_pkt;
                cmd->cmd_state = SF_STATE_IDLE;
                pkt->pkt_reason = CMD_TIMEOUT;
                pkt->pkt_statistics |= STAT_TIMEOUT;
                tgt_id = pkt->pkt_address.a_target;
                sf->sf_stats.tstats[tgt_id].abts_failures++;
                if (pkt->pkt_comp != NULL) {
                        (*pkt->pkt_comp)(pkt);
                }
                break;

        case FCAL_BAD_ABORT:
                if (sf_core && (sf_core & SF_CORE_BAD_ABORT)) {
                        sf_token = (int *)(uintptr_t)fpkt->fcal_socal_request.\
                            sr_soc_hdr.sh_request_token;
                        (void) soc_take_core(sf->sf_sochandle, sf->sf_socp);
                        sf_core = 0;
                }
                SF_DEBUG(1, (sf, CE_NOTE, "Command Abort bad abort\n"));
                cmd->cmd_timeout = sf_watchdog_time + cmd->cmd_pkt->pkt_time
                    + 20;
                break;

        case FCAL_TIMEOUT:
                retval = TRUE;
                break;

        default:
                pkt = cmd->cmd_pkt;
                tgt_id = pkt->pkt_address.a_target;
                sf_log(sf, CE_WARN,
                "Command Abort failed target 0x%x, forcing a LIP\n", tgt_id);
                if (sf_core && (sf_core & SF_CORE_ABORT_TIMEOUT)) {
                        sf_token = (int *)(uintptr_t)fpkt->fcal_socal_request.\
                            sr_soc_hdr.sh_request_token;
                        (void) soc_take_core(sf->sf_sochandle, sf->sf_socp);
                        sf_core = 0;
                }
                sf_force_lip(sf);
                retval = TRUE;
                break;
        }

        return (retval);
}


/*
 * an ELS command has timed out
 * return ???
 */
static struct sf_els_hdr *
sf_els_timeout(struct sf *sf, struct sf_els_hdr *privp)
{
        struct fcal_packet *fpkt;
        int rval, dflag, timeout = SF_ELS_TIMEOUT;
	uint_t lip_cnt = privp->lip_cnt;
	uchar_t els_code = privp->els_code;
	struct sf_target *target = privp->target;
	char what[64];

	fpkt = privp->fpkt;
	dflag = privp->delayed_retry;
	/* use as temporary state variable */
	privp->timeout = SF_INVALID_TIMEOUT;
	mutex_exit(&sf->sf_mutex);

	if (privp->fpkt->fcal_pkt_comp == sf_els_callback) {
		/*
		 * take socal core if required. Timeouts for IB and hosts
		 * are not very interesting, so we take socal core only
		 * if the timeout is *not* for a IB or host.
		 */
		if (sf_core && (sf_core & SF_CORE_ELS_TIMEOUT) &&
		    ((sf_alpa_to_switch[privp->dest_nport_id] &
		    0x0d) != 0x0d) && ((privp->dest_nport_id != 1) ||
		    (privp->dest_nport_id != 2) ||
		    (privp->dest_nport_id != 4) ||
		    (privp->dest_nport_id != 8) ||
		    (privp->dest_nport_id != 0xf))) {
			sf_token = (int *)(uintptr_t)fpkt->fcal_socal_request.\
			    sr_soc_hdr.sh_request_token;
			(void) soc_take_core(sf->sf_sochandle, sf->sf_socp);
			sf_core = 0;
		}
		(void) sprintf(what, "ELS 0x%x", privp->els_code);
	} else if (privp->fpkt->fcal_pkt_comp == sf_reportlun_callback) {
		if (sf_core && (sf_core & SF_CORE_REPORTLUN_TIMEOUT)) {
			sf_token = (int *)(uintptr_t)fpkt->fcal_socal_request.\
			    sr_soc_hdr.sh_request_token;
			(void) soc_take_core(sf->sf_sochandle, sf->sf_socp);
			sf_core = 0;
		}
		timeout = SF_FCP_TIMEOUT;
		(void) sprintf(what, "REPORT_LUNS");
	} else if (privp->fpkt->fcal_pkt_comp == sf_inq_callback) {
		if (sf_core && (sf_core & SF_CORE_INQUIRY_TIMEOUT)) {
			sf_token = (int *)(uintptr_t)
			    fpkt->fcal_socal_request.\
			    sr_soc_hdr.sh_request_token;
			(void) soc_take_core(sf->sf_sochandle, sf->sf_socp);
			sf_core = 0;
		}
		timeout = SF_FCP_TIMEOUT;
		(void) sprintf(what, "INQUIRY to LUN 0x%lx",
		    (long)SCSA_LUN(target));
	} else {
		(void) sprintf(what, "UNKNOWN OPERATION");
	}

	if (dflag) {
		/* delayed retry */
		SF_DEBUG(2, (sf, CE_CONT,
		    "!sf%d: %s to target %x delayed retry\n",
		    ddi_get_instance(sf->sf_dip), what,
		    sf_alpa_to_switch[privp->dest_nport_id]));
		privp->delayed_retry = FALSE;
		goto try_again;
	}

	sf_log(sf, CE_NOTE, "!%s to target 0x%x alpa 0x%x timed out\n",
	    what, sf_alpa_to_switch[privp->dest_nport_id],
```

```
	    privp->dest_nport_id);

	rval = soc_abort(sf->sf_sochandle, sf->sf_socp, sf->sf_sochandle
	    ->fcal_portno, fpkt, 1);
	if (rval == FCAL_ABORTED || rval == FCAL_ABORT_FAILED) {
	SF_DEBUG(1, (sf, CE_NOTE, "!%s abort to al_pa %x succeeded\n",
	    what, privp->dest_nport_id));
try_again:

		mutex_enter(&sf->sf_mutex);
		if (privp->prev != NULL) {
			privp->prev->next = privp->next;
		}
		if (sf->sf_els_list == privp) {
			sf->sf_els_list = privp->next;
		}
		if (privp->next != NULL) {
			privp->next->prev = privp->prev;
		}
		privp->prev = privp->next = NULL;
		if (lip_cnt == sf->sf_lip_cnt) {
			privp->timeout = sf_watchdog_time + timeout;
			if ((++(privp->retries) < sf_els_retries) ||
			    (dflag && (privp->retries < SF_BSY_RETRIES))) {
				mutex_exit(&sf->sf_mutex);
				sf_log(sf, CE_NOTE,
				    "!%s to target 0x%x retrying\n",
				    what,
				    sf_alpa_to_switch[privp->dest_nport_id]);
				if (sf_els_transport(sf, privp) == 1) {
					mutex_enter(&sf->sf_mutex);
					return (sf->sf_els_list); /* success */
				}
				mutex_enter(&sf->sf_mutex);
				fpkt = NULL;
			}
			if ((lip_cnt == sf->sf_lip_cnt) &&
			    (els_code != LA_ELS_LOGO)) {
				if (target != NULL) {
					sf_offline_target(sf, target);
				}
				if (sf->sf_lip_cnt == lip_cnt) {
					sf->sf_device_count--;
					ASSERT(sf->sf_device_count >= 0);
					if (sf->sf_device_count == 0) {
						sf_finish_init(sf,
						    sf->sf_lip_cnt);
					}
				}
			}
			privp = sf->sf_els_list;
			mutex_exit(&sf->sf_mutex);
			if (fpkt != NULL) {
				sf_els_free(fpkt);
			}
		} else {
			mutex_exit(&sf->sf_mutex);
			sf_els_free(privp->fpkt);
			privp = NULL;
		}
	} else {
		if (sf_core && (sf_core & SF_CORE_ELS_FAILED)) {
			sf_token = (int *)(uintptr_t)
			    fpkt->fcal_socal_request.\
			    sr_soc_hdr.sh_request_token;
			(void) soc_take_core(sf->sf_sochandle, sf->sf_socp);
```

```
                        sf_core = 0;
                }
                sf_log(sf, CE_NOTE, "%s abort to target 0x%x failed. "
                    "status=0x%x, forcing LIP\n", what,
                    sf_alpa_to_switch[privp->dest_nport_id], rval);
                privp = NULL;
                if (sf->sf_lip_cnt == lip_cnt) {
                        sf_force_lip(sf);
                }
        }

        mutex_enter(&sf->sf_mutex);
        return (privp);
}


/*
 * called by timeout when a reset times out
 */
/*ARGSUSED*/
static void
sf_check_reset_delay(void *arg)
{
        struct sf *sf;
        struct sf_target *target;
        struct sf_reset_list *rp, *tp;
        uint_t lip_cnt, reset_timeout_flag = FALSE;
        clock_t lb;

        lb = ddi_get_lbolt();

        mutex_enter(&sf_global_mutex);

        sf_reset_timeout_id = 0;

        for (sf = sf_head; sf != NULL; sf = sf->sf_next) {

                mutex_exit(&sf_global_mutex);
                mutex_enter(&sf->sf_mutex);

                /* is this type cast needed? */
                tp = (struct sf_reset_list *)&sf->sf_reset_list;

                rp = sf->sf_reset_list;
                while (rp != NULL) {
                        if (((rp->timeout - lb) < 0) &&
                            (rp->lip_cnt == sf->sf_lip_cnt)) {
                                tp->next = rp->next;
                                mutex_exit(&sf->sf_mutex);
                                target = rp->target;
                                lip_cnt = rp->lip_cnt;
                                kmem_free(rp, sizeof (struct sf_reset_list));
                                /* abort all cmds for this target */
                                while (target) {
                                        sf_abort_all(sf, target, FALSE,
                                            lip_cnt, TRUE);
                                        mutex_enter(&target->sft_mutex);
                                        if (lip_cnt == sf->sf_lip_cnt) {
                                                target->sft_state &=
                                                    ~SF_TARGET_BUSY;
                                        }
                                        mutex_exit(&target->sft_mutex);
                                        target = target->sft_next_lun;
                                }
                                mutex_enter(&sf->sf_mutex);
                                tp = (struct sf_reset_list *)
```

```
                                    &sf->sf_reset_list;
                                rp = sf->sf_reset_list;
                                lb = ddi_get_lbolt();
                        } else if (rp->lip_cnt != sf->sf_lip_cnt) {
                                tp->next = rp->next;
                                kmem_free(rp, sizeof (struct sf_reset_list));
                                rp = tp->next;
                        } else {
                                reset_timeout_flag = TRUE;
                                tp = rp;
                                rp = rp->next;
                        }
                }
                mutex_exit(&sf->sf_mutex);
                mutex_enter(&sf_global_mutex);
        }

        if (reset_timeout_flag && (sf_reset_timeout_id == 0)) {
                sf_reset_timeout_id = timeout(sf_check_reset_delay,
                    NULL, drv_usectohz(SF_TARGET_RESET_DELAY));
        }

        mutex_exit(&sf_global_mutex);
}


/*
 * called to "reset the bus", i.e. force loop initialization (and address
 * re-negotiation)
 */
static void
sf_force_lip(struct sf *sf)
{
        int i;
        struct sf_target *target;


        /* disable restart of lip if we're suspended */
        mutex_enter(&sf->sf_mutex);
        if (sf->sf_state & SF_STATE_SUSPENDED) {
                mutex_exit(&sf->sf_mutex);
                SF_DEBUG(1, (sf, CE_CONT,
                    "sf_force_lip, sf%d: lip restart disabled "
                    "due to DDI_SUSPEND\n",
                    ddi_get_instance(sf->sf_dip)));
                return;
        }

        sf_log(sf, CE_NOTE, "Forcing lip\n");

        for (i = 0; i < sf_max_targets; i++) {
                target = sf->sf_targets[i];
                while (target != NULL) {
                        mutex_enter(&target->sft_mutex);
                        if (!(target->sft_state & SF_TARGET_OFFLINE))
                                target->sft_state |= SF_TARGET_BUSY;
                        mutex_exit(&target->sft_mutex);
                        target = target->sft_next_lun;
                }
        }

        sf->sf_lip_cnt++;
        sf->sf_timer = sf_watchdog_time + SF_OFFLINE_TIMEOUT;
        sf->sf_state = SF_STATE_OFFLINE;
        mutex_exit(&sf->sf_mutex);
        sf->sf_stats.lip_count++;               /* no mutex for this? */
```

```
#ifdef DEBUG
        /* are we allowing LIPs ?? */
        if (sf_lip_flag != 0) {
#endif
                /* call the transport to force loop initialization */
                if (((i = soc_force_lip(sf->sf_sochandle, sf->sf_socp,
                    sf->sf_sochandle->fcal_portno, 1,
                    FCAL_FORCE_LIP)) != FCAL_SUCCESS) &&
                    (i != FCAL_TIMEOUT)) {
                        /* force LIP failed */
                        if (sf_core && (sf_core & SF_CORE_LIP_FAILED)) {
                                (void) soc_take_core(sf->sf_sochandle,
                                    sf->sf_socp);
                                sf_core = 0;
                        }
#ifdef DEBUG
                        /* are we allowing reset after LIP failed ?? */
                        if (sf_reset_flag != 0) {
#endif
                                /* restart socal after resetting it */
                                sf_log(sf, CE_NOTE,
                                    "!Force lip failed Status code 0x%x."
                                    " Reseting\n", i);
                                /* call transport to force a reset */
                                soc_force_reset(sf->sf_sochandle, sf->sf_socp,
                                    sf->sf_sochandle->fcal_portno, 1);
#ifdef  DEBUG
                        }
#endif
                }
#ifdef  DEBUG
        }
#endif
}


/*
 * called by the transport when an unsolicited ELS is received
 */
static void
sf_unsol_els_callback(void *arg, soc_response_t *srp, caddr_t payload)
{
        struct sf *sf = (struct sf *)arg;
        els_payload_t   *els = (els_payload_t *)payload;
        struct la_els_rjt *rsp;
        int     i, tgt_id;
        uchar_t dest_id;
        struct fcal_packet *fpkt;
        fc_frame_header_t *hp;
        struct sf_els_hdr *privp;


        if ((els == NULL) || ((i = srp->sr_soc_hdr.sh_byte_cnt) == 0)) {
                return;
        }

        if (i > SOC_CQE_PAYLOAD) {
                i = SOC_CQE_PAYLOAD;
        }

        dest_id = (uchar_t)srp->sr_fc_frame_hdr.s_id;
        tgt_id = sf_alpa_to_switch[dest_id];

        switch (els->els_cmd.c.ls_command) {
```

```
        case LA_ELS_LOGO:
                /*
                 * logout received -- log the fact
                 */
                sf->sf_stats.tstats[tgt_id].logouts_recvd++;
                sf_log(sf, CE_NOTE, "!LOGO recvd from target %x, %s\n",
                    tgt_id,
                    sf_lip_on_plogo ? "Forcing LIP...." : "");
                if (sf_lip_on_plogo) {
                        sf_force_lip(sf);
                }
                break;

        default:  /* includes LA_ELS_PLOGI */
                /*
                 * something besides a logout received -- we don’t handle
                 * this so send back a reject saying its unsupported
                 */

                sf_log(sf, CE_NOTE, "!ELS 0x%x recvd from target 0x%x\n",
                    els->els_cmd.c.ls_command, tgt_id);


                /* allocate room for a response */
                if (sf_els_alloc(sf, dest_id, sizeof (struct sf_els_hdr),
                    sizeof (struct la_els_rjt), sizeof (union sf_els_rsp),
                    (caddr_t *)&privp, (caddr_t *)&rsp) == NULL) {
                        break;
                }

                fpkt = privp->fpkt;

                /* fill in pkt header */
                hp = &fpkt->fcal_socal_request.sr_fc_frame_hdr;
                hp->r_ctl = R_CTL_ELS_RSP;
                hp->f_ctl = F_CTL_LAST_SEQ | F_CTL_XCHG_CONTEXT;
                hp->ox_id = srp->sr_fc_frame_hdr.ox_id;
                hp->rx_id = srp->sr_fc_frame_hdr.rx_id;
                fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_type =
                    CQ_TYPE_OUTBOUND;

                fpkt->fcal_socal_request.sr_soc_hdr.sh_seg_cnt = 1;

                /* fill in response */
                rsp->ls_code = LA_ELS_RJT;      /* reject this ELS */
                rsp->mbz[0] = 0;
                rsp->mbz[1] = 0;
                rsp->mbz[2] = 0;
                ((struct la_els_logi *)privp->rsp)->ls_code = LA_ELS_ACC;
                *((int *)&rsp->reserved) = 0;
                rsp->reason_code = RJT_UNSUPPORTED;
                privp->retries = sf_els_retries;
                privp->els_code = LA_ELS_RJT;
                privp->timeout = (unsigned)0xffffffff;
                (void) sf_els_transport(sf, privp);
                break;
        }
}


/*
 * Error logging, printing, and debug print routines
 */

/*PRINTFLIKE3*/
static void
```

```
sf_log(struct sf *sf, int level, const char *fmt, ...)
{
        char buf[256];
        dev_info_t *dip;
        va_list ap;

        if (sf != NULL) {
                dip = sf->sf_dip;
        } else {
                dip = NULL;
        }

        va_start(ap, fmt);
        (void) vsprintf(buf, fmt, ap);
        va_end(ap);
        scsi_log(dip, "sf", level, buf);
}


/*
 * called to get some sf kstats -- return 0 on success else return errno
 */
static int
sf_kstat_update(kstat_t *ksp, int rw)
{
        struct sf *sf;

        if (rw == KSTAT_WRITE) {
                /* can't write */
                return (EACCES);
        }

        sf = ksp->ks_private;
        sf->sf_stats.ncmds = sf->sf_ncmds;
        sf->sf_stats.throttle_limit = sf->sf_throttle;
        sf->sf_stats.cr_pool_size = sf->sf_cr_pool_cnt;

        return (0);                             /* success */
}


/*
 * Unix Entry Points
 */

/*
 * driver entry point for opens on control device
 */
/* ARGSUSED */
static int
sf_open(dev_t *dev_p, int flag, int otyp, cred_t *cred_p)
{
        dev_t dev = *dev_p;
        struct sf *sf;


        /* just ensure soft state exists for this device */
        sf = ddi_get_soft_state(sf_state, SF_MINOR2INST(getminor(dev)));
        if (sf == NULL) {
                return (ENXIO);
        }

        ++(sf->sf_check_n_close);

        return (0);
}
```

```
/*
 * driver entry point for last close on control device
 */
/* ARGSUSED */
static int
sf_close(dev_t dev, int flag, int otyp, cred_t *cred_p)
{
        struct sf *sf;

        sf = ddi_get_soft_state(sf_state, SF_MINOR2INST(getminor(dev)));
        if (sf == NULL) {
                return (ENXIO);
        }

        if (!sf->sf_check_n_close) { /* if this flag is zero */
                cmn_err(CE_WARN, "sf%d: trying to close unopened instance",
                    SF_MINOR2INST(getminor(dev)));
                return (ENODEV);
        } else {
                --(sf->sf_check_n_close);
        }
        return (0);
}


/*
 * driver entry point for sf ioctl commands
 */
/* ARGSUSED */
static int
sf_ioctl(dev_t dev,
    int cmd, intptr_t arg, int mode, cred_t *cred_p, int *rval_p)
{
        struct sf *sf;
        struct sf_target *target;
        uchar_t al_pa;
        struct sf_al_map map;
        int cnt, i;
        int     retval;                         /* return value */
        struct devctl_iocdata *dcp;
        dev_info_t *cdip;
        struct scsi_address ap;
        scsi_hba_tran_t *tran;


        sf = ddi_get_soft_state(sf_state, SF_MINOR2INST(getminor(dev)));
        if (sf == NULL) {
                return (ENXIO);
        }

        /* handle all ioctls */
        switch (cmd) {

        /*
         * We can use the generic implementation for these ioctls
         */
        case DEVCTL_DEVICE_GETSTATE:
        case DEVCTL_DEVICE_ONLINE:
        case DEVCTL_DEVICE_OFFLINE:
        case DEVCTL_BUS_GETSTATE:
                return (ndi_devctl_ioctl(sf->sf_dip, cmd, arg, mode, 0));

        /*
         * return FC map
```

```
	 */
	case SFIOCGMAP:
		if ((sf->sf_lilp_map->lilp_magic != FCAL_LILP_MAGIC &&
		    sf->sf_lilp_map->lilp_magic != FCAL_BADLILP_MAGIC) ||
		    sf->sf_state != SF_STATE_ONLINE) {
			retval = ENOENT;
			goto dun;
		}
		mutex_enter(&sf->sf_mutex);
		if (sf->sf_lilp_map->lilp_magic == FCAL_BADLILP_MAGIC) {
			int i, j = 0;

			/* Need to generate a fake lilp map */
			for (i = 0; i < sf_max_targets; i++) {
				if (sf->sf_targets[i])
					sf->sf_lilp_map->lilp_alpalist[j++] =
					    sf->sf_targets[i]->
					    sft_hard_address;
			}
			sf->sf_lilp_map->lilp_length = (uchar_t)j;
		}
		cnt = sf->sf_lilp_map->lilp_length;
		map.sf_count = (short)cnt;
		bcopy((caddr_t)&sf->sf_sochandle->fcal_n_wwn,
		    (caddr_t)&map.sf_hba_addr.sf_node_wwn,
		    sizeof (la_wwn_t));
		bcopy((caddr_t)&sf->sf_sochandle->fcal_p_wwn,
		    (caddr_t)&map.sf_hba_addr.sf_port_wwn,
		    sizeof (la_wwn_t));
		map.sf_hba_addr.sf_al_pa = sf->sf_al_pa;
		map.sf_hba_addr.sf_hard_address = 0;
		map.sf_hba_addr.sf_inq_dtype = DTYPE_UNKNOWN;
		for (i = 0; i < cnt; i++) {
			al_pa = sf->sf_lilp_map->lilp_alpalist[i];
			map.sf_addr_pair[i].sf_al_pa = al_pa;
			if (al_pa == sf->sf_al_pa) {
				(void) bcopy((caddr_t)&sf->sf_sochandle
				    ->fcal_n_wwn, (caddr_t)&map.
				    sf_addr_pair[i].sf_node_wwn,
				    sizeof (la_wwn_t));
				(void) bcopy((caddr_t)&sf->sf_sochandle
				    ->fcal_p_wwn, (caddr_t)&map.
				    sf_addr_pair[i].sf_port_wwn,
				    sizeof (la_wwn_t));
				map.sf_addr_pair[i].sf_hard_address =
				    al_pa;
				map.sf_addr_pair[i].sf_inq_dtype =
				    DTYPE_PROCESSOR;
				continue;
			}
			target = sf->sf_targets[sf_alpa_to_switch[
			    al_pa]];
			if (target != NULL) {
				mutex_enter(&target->sft_mutex);
				if (!(target->sft_state &
				    (SF_TARGET_OFFLINE |
				    SF_TARGET_BUSY))) {
					bcopy((caddr_t)&target->
					    sft_node_wwn,
					    (caddr_t)&map.sf_addr_pair
					    [i].sf_node_wwn,
					    sizeof (la_wwn_t));
					bcopy((caddr_t)&target->
					    sft_port_wwn,
					    (caddr_t)&map.sf_addr_pair
					    [i].sf_port_wwn,
```

```
					    sizeof (la_wwn_t));
					map.sf_addr_pair[i].
					    sf_hard_address
					    = target->sft_hard_address;
					map.sf_addr_pair[i].
					    sf_inq_dtype
					    = target->sft_device_type;
					mutex_exit(&target->sft_mutex);
					continue;
				}
				mutex_exit(&target->sft_mutex);
			}
			bzero((caddr_t)&map.sf_addr_pair[i].
			    sf_node_wwn, sizeof (la_wwn_t));
			bzero((caddr_t)&map.sf_addr_pair[i].
			    sf_port_wwn, sizeof (la_wwn_t));
			map.sf_addr_pair[i].sf_inq_dtype =
			    DTYPE_UNKNOWN;
		}
		mutex_exit(&sf->sf_mutex);
		if (ddi_copyout((caddr_t)&map, (caddr_t)arg,
		    sizeof (struct sf_al_map), mode) != 0) {
			retval = EFAULT;
			goto dun;
		}
		break;

	/*
	 * handle device control ioctls
	 */
	case DEVCTL_DEVICE_RESET:
		if (ndi_dc_allochdl((void *)arg, &dcp) != NDI_SUCCESS) {
			retval = EFAULT;
			goto dun;
		}
		if ((ndi_dc_getname(dcp) == NULL) ||
		    (ndi_dc_getaddr(dcp) == NULL)) {
			ndi_dc_freehdl(dcp);
			retval = EINVAL;
			goto dun;
		}
		cdip = ndi_devi_find(sf->sf_dip,
		    ndi_dc_getname(dcp), ndi_dc_getaddr(dcp));
		ndi_dc_freehdl(dcp);

		if (cdip == NULL) {
			retval = ENXIO;
			goto dun;
		}

		if ((target = sf_get_target_from_dip(sf, cdip)) == NULL) {
			retval = ENXIO;
			goto dun;
		}
		mutex_enter(&target->sft_mutex);
		if (!(target->sft_state & SF_TARGET_INIT_DONE)) {
			mutex_exit(&target->sft_mutex);
			retval = ENXIO;
			goto dun;
		}

		/* This is ugly */
		tran = kmem_zalloc(scsi_hba_tran_size(), KM_SLEEP);
		bcopy(target->sft_tran, tran, scsi_hba_tran_size());
		mutex_exit(&target->sft_mutex);
		ap.a_hba_tran = tran;
```

```
                ap.a_target = sf_alpa_to_switch[target->sft_al_pa];
                if (sf_reset(&ap, RESET_TARGET) == FALSE) {
                        retval = EIO;
                } else {
                        retval = 0;
                }
                kmem_free(tran, scsi_hba_tran_size());
                goto dun;

        case DEVCTL_BUS_QUIESCE:
        case DEVCTL_BUS_UNQUIESCE:
                retval = ENOTSUP;
                goto dun;

        case DEVCTL_BUS_RESET:
        case DEVCTL_BUS_RESETALL:
                sf_force_lip(sf);
                break;

        default:
                retval = ENOTTY;
                goto dun;
        }

        retval = 0;                             /* success */

dun:
        return (retval);
}


/*
 * get the target given a DIP
 */
static struct sf_target *
sf_get_target_from_dip(struct sf *sf, dev_info_t *dip)
{
        int i;
        struct sf_target *target;


        /* scan each hash queue for the DIP in question */
        for (i = 0; i < SF_NUM_HASH_QUEUES; i++) {
                target = sf->sf_wwn_lists[i];
                while (target != NULL) {
                        if (target->sft_dip == dip) {
                                return (target); /* success: target found */
                        }
                        target = target->sft_next;
                }
        }
        return (NULL);                          /* failure: target not found */
}


/*
 * called by the transport to get an event cookie
 */
static int
sf_bus_get_eventcookie(dev_info_t *dip, dev_info_t *rdip, char *name,
    ddi_eventcookie_t *event_cookiep)
{
        struct sf *sf;

        sf = ddi_get_soft_state(sf_state, ddi_get_instance(dip));
        if (sf == NULL) {
```

```
                /* can't find instance for this device */
                return (DDI_FAILURE);
        }

        return (ndi_event_retrieve_cookie(sf->sf_event_hdl, rdip, name,
            event_cookiep, NDI_EVENT_NOPASS));

}


/*
 * called by the transport to add an event callback
 */
static int
sf_bus_add_eventcall(dev_info_t *dip, dev_info_t *rdip,
    ddi_eventcookie_t eventid, void (*callback)(dev_info_t *dip,
    ddi_eventcookie_t event, void *arg, void *impl_data), void *arg,
    ddi_callback_id_t *cb_id)
{
        struct sf *sf;

        sf = ddi_get_soft_state(sf_state, ddi_get_instance(dip));
        if (sf == NULL) {
                /* can't find instance for this device */
                return (DDI_FAILURE);
        }

        return (ndi_event_add_callback(sf->sf_event_hdl, rdip,
            eventid, callback, arg, NDI_SLEEP, cb_id));

}


/*
 * called by the transport to remove an event callback
 */
static int
sf_bus_remove_eventcall(dev_info_t *devi, ddi_callback_id_t cb_id)
{
        struct sf *sf;

        sf = ddi_get_soft_state(sf_state, ddi_get_instance(devi));
        if (sf == NULL) {
                /* can't find instance for this device */
                return (DDI_FAILURE);
        }

        return (ndi_event_remove_callback(sf->sf_event_hdl, cb_id));
}


/*
 * called by the transport to post an event
 */
static int
sf_bus_post_event(dev_info_t *dip, dev_info_t *rdip,
    ddi_eventcookie_t eventid, void *impldata)
{
        ddi_eventcookie_t remove_cookie, cookie;

        /* is this a remove event ?? */
        struct sf *sf = ddi_get_soft_state(sf_state, ddi_get_instance(dip));
        remove_cookie = ndi_event_tag_to_cookie(sf->sf_event_hdl,
            SF_EVENT_TAG_REMOVE);

        if (remove_cookie == eventid) {
```

```
		struct sf_target *target;

		/* handle remove event */

		if (sf == NULL) {
			/* no sf instance for this device */
			return (NDI_FAILURE);
		}

		/* get the target for this event */
		if ((target = sf_get_target_from_dip(sf, rdip)) != NULL) {
			/*
			 * clear device info for this target and mark as
			 * not done
			 */
			mutex_enter(&target->sft_mutex);
			target->sft_dip = NULL;
			target->sft_state &= ~SF_TARGET_INIT_DONE;
			mutex_exit(&target->sft_mutex);
			return (NDI_SUCCESS); /* event handled */
		}

		/* no target for this event */
		return (NDI_FAILURE);
	}

	/* an insertion event */
	if (ndi_busop_get_eventcookie(dip, rdip, FCAL_INSERT_EVENT, &cookie)
	    != NDI_SUCCESS) {
		return (NDI_FAILURE);
	}

	return (ndi_post_event(dip, rdip, cookie, impldata));
}


/*
 * the sf hotplug daemon, one thread per sf instance
 */
static void
sf_hp_daemon(void *arg)
{
	struct sf *sf = (struct sf *)arg;
	struct sf_hp_elem *elem;
	struct sf_target *target;
	int tgt_id;
	callb_cpr_t cprinfo;

	CALLB_CPR_INIT(&cprinfo, &sf->sf_hp_daemon_mutex,
	    callb_generic_cpr, "sf_hp_daemon");

	mutex_enter(&sf->sf_hp_daemon_mutex);

	do {
		while (sf->sf_hp_elem_head != NULL) {

			/* save ptr to head of list */
			elem = sf->sf_hp_elem_head;

			/* take element off of list */
			if (sf->sf_hp_elem_head == sf->sf_hp_elem_tail) {
				/* element only one in list -- list now empty */
				sf->sf_hp_elem_head = NULL;
				sf->sf_hp_elem_tail = NULL;
			} else {
				/* remove element from head of list */
```

```
				sf->sf_hp_elem_head = sf->sf_hp_elem_head->next;
			}

			mutex_exit(&sf->sf_hp_daemon_mutex);

			switch (elem->what) {
			case SF_ONLINE:
				/* online this target */
				target = elem->target;
				(void) ndi_devi_online(elem->dip, 0);
				(void) ndi_event_retrieve_cookie(
				    sf->sf_event_hdl,
				    target->sft_dip, FCAL_INSERT_EVENT,
				    &sf_insert_eid, NDI_EVENT_NOPASS);
				(void) ndi_event_run_callbacks(sf->sf_event_hdl,
				    target->sft_dip, sf_insert_eid, NULL);
				break;
			case SF_OFFLINE:
				/* offline this target */
				target = elem->target;
				tgt_id = sf_alpa_to_switch[target->sft_al_pa];
				/* don't do NDI_DEVI_REMOVE for now */
				if (ndi_devi_offline(elem->dip, 0) !=
				    NDI_SUCCESS) {
					SF_DEBUG(1, (sf, CE_WARN, "target %x, "
					    "device offline failed", tgt_id));
				} else {
					SF_DEBUG(1, (sf, CE_NOTE, "target %x, "
					    "device offline succeeded\n",
					    tgt_id));
				}
				break;
			}
			kmem_free(elem, sizeof (struct sf_hp_elem));
			mutex_enter(&sf->sf_hp_daemon_mutex);
		}

		/* if exit is not already signaled */
		if (sf->sf_hp_exit == 0) {
			/* wait to be signaled by work or exit */
			CALLB_CPR_SAFE_BEGIN(&cprinfo);
			cv_wait(&sf->sf_hp_daemon_cv, &sf->sf_hp_daemon_mutex);
			CALLB_CPR_SAFE_END(&cprinfo, &sf->sf_hp_daemon_mutex);
		}
	} while (sf->sf_hp_exit == 0);

	/* sf_hp_daemon_mutex is dropped by CALLB_CPR_EXIT */
	CALLB_CPR_EXIT(&cprinfo);
	thread_exit();			/* no more hotplug thread */
	/* NOTREACHED */
}
```

```
**********************************************************
   99351 Thu Feb 25 15:39:46 2016
new/usr/src/uts/sun4u/io/rmclomv.c
2976 remove useless offsetof() macros
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */


#include <sys/types.h>
#include <sys/stat.h>
#include <sys/conf.h>
#include <sys/modctl.h>
#include <sys/callb.h>
#include <sys/strlog.h>
#include <sys/cyclic.h>
#include <sys/rmc_comm_dp.h>
#include <sys/rmc_comm_dp_boot.h>
#include <sys/rmc_comm_drvintf.h>
#include <sys/rmc_comm.h>
#include <sys/machsystm.h>
#include <sys/sysevent.h>
#include <sys/sysevent/dr.h>
#include <sys/sysevent/env.h>
#include <sys/sysevent/eventdefs.h>
#include <sys/file.h>
#include <sys/disp.h>
#include <sys/reboot.h>
#include <sys/envmon.h>
#include <sys/rmclomv_impl.h>
#include <sys/cpu_sgnblk_defs.h>
#include <sys/utsname.h>
#include <sys/systeminfo.h>
#include <sys/ddi.h>
#include <sys/time.h>
#include <sys/promif.h>
#include <sys/sysmacros.h>
#endif /* ! codereview */

#define offsetof(s, m)  (size_t)(&(((s *)0)->m))
#define RMCRESBUFLEN    1024
#define DATE_TIME_MSG_SIZE      78
#define RMCLOMV_WATCHDOG_MODE   "rmclomv-watchdog-mode"
```

```
#define DELAY_TIME      5000000  /* 5 seconds, in microseconds */
#define CPU_SIGNATURE_DELAY_TIME        5000000  /* 5 secs, in microsecs */

extern void     pmugpio_watchdog_pat();

extern int      watchdog_activated;
static int      last_watchdog_msg = 1;
extern int      watchdog_enable;
extern int      boothowto;

int             rmclomv_watchdog_mode;

/*
 * functions local to this driver.
 */
static int      rmclomv_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg,
    void **resultp);
static int      rmclomv_attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
static int      rmclomv_detach(dev_info_t *dip, ddi_detach_cmd_t cmd);
static uint_t   rmclomv_break_intr(caddr_t arg);
static int      rmclomv_add_intr_handlers(void);
static int      rmclomv_remove_intr_handlers(void);
static uint_t   rmclomv_event_data_handler(char *);
static void     rmclomv_dr_data_handler(const char *, int);
static int      rmclomv_open(dev_t *dev_p, int flag, int otyp, cred_t *cred_p);
static int      rmclomv_close(dev_t dev, int flag, int otyp, cred_t *cred_p);
static int      rmclomv_ioctl(dev_t dev, int cmd, intptr_t arg, int mode,
    cred_t *cred_p, int *rval_p);
static void     rmclomv_checkrmc_start(void);
static void     rmclomv_checkrmc_destroy(void);
static void     rmclomv_checkrmc_wakeup(void *);
static void     rmclomv_refresh_start(void);
static void     rmclomv_refresh_destroy(void);
static void     rmclomv_refresh_wakeup(void);
static void     rmclomv_reset_cache(rmclomv_cache_section_t *new_chain,
    rmclomv_cache_section_t *new_subchain, dp_get_sysinfo_r_t *sysinfo);
static rmclomv_cache_section_t *rmclomv_find_section(
    rmclomv_cache_section_t *start, uint16_t sensor);
static rmclomv_cache_section_t *create_cache_section(int sensor_type, int num);
static int      get_sensor_by_name(const rmclomv_cache_section_t *section,
    const char *name, int *index);
static int      validate_section_entry(rmclomv_cache_section_t *section,
    int index);
static int      add_names_to_section(rmclomv_cache_section_t *section);
static void     free_section(rmclomv_cache_section_t *section);
static void     add_section(rmclomv_cache_section_t **head,
    rmclomv_cache_section_t *section);
static int      rmclomv_do_cmd(int req_cmd, int resp_cmd, int resp_len,
    intptr_t arg_req, intptr_t arg_res);
static void     refresh_name_cache(int force_fail);
static void     set_val_unav(envmon_sensor_t *sensor);
static void     set_fan_unav(envmon_fan_t *fan);
static int      do_psu_cmd(intptr_t arg, int mode, envmon_indicator_t *env_ind,
    dp_get_psu_status_t *rmc_psu, dp_get_psu_status_r_t *rmc_psu_r,
    int detector_type);
static uint_t rmc_set_watchdog_timer(uint_t timeoutval);
static uint_t rmc_clear_watchdog_timer(void);
static void send_watchdog_msg(int msg);
static void plat_timesync(void *arg);

static kmutex_t         timesync_lock;
static clock_t          timesync_interval = 0;
static timeout_id_t     timesync_tid = 0;

/*
 * Driver entry points
 */
static struct cb_ops rmclomv_cb_ops = {
        rmclomv_open,   /* open */
        rmclomv_close,  /* close */
        nodev,          /* strategy() */
        nodev,          /* print() */
        nodev,          /* dump() */
        nodev,          /* read() */
        nodev,          /* write() */
        rmclomv_ioctl,  /* ioctl() */
        nodev,          /* devmap() */
        nodev,          /* mmap() */
        ddi_segmap,     /* segmap() */
        nochpoll,       /* poll() */
        ddi_prop_op,    /* prop_op() */
        NULL,           /* cb_str */
        D_NEW | D_MP    /* cb_flag */
};
```
______***unchanged_portion_omitted_***